

JAVA™
COMO PROGRAMAR

D325j

Deitel, H. M.

Java, como programar / H. M. Deitel e P. J. Deitel; trad. Carlos Arthur Lang Lisboa. – 4.ed. – Porto Alegre : Bookman, 2003.

1. Computação – Programação de computadores – Java. I. Deitel, P. J. II. Título.

CDU 681.3 (Java)

Catalogação na publicação: Mônica Ballejo Canto – CRB 10/1023

ISBN 85-363-0123-6

H. M. Deitel
Deitel & Associates, Inc.

P. J. Deitel
Deitel & Associates, Inc.

JAVATM

COMO PROGRAMAR

4^a Edição

Tradução e revisão técnica:
CARLOS ARTHUR LANG LISBÔA
Professor do Instituto de Informática da UFRGS

Consultoria desta edição:
MARIA LÚCIA BLANCK LISBÔA
Professora do Instituto de Informática da UFRGS



2003

Obra originalmente publicada sob o título

Java How to Program, 4th Edition

© 2002, Prentice Hall, Inc.

*Tradução autorizada a partir do original em língua inglesa publicado por Pearson Education, Inc., através da sua divisão
Prentice Hall.*

ISBN 0-13-034151-7

Capa: Mário Röhnelt

Preparação do original: Daniel Grassi

Supervisão editorial: Arysinha Jacques Affonso

Editoração eletrônica: Laser House

Impressão e Acabamento: R.R. Donnelley América Latina

Reservados todos os direitos de publicação, em língua portuguesa, à

ARTMED® EDITORA S. A.

(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S. A.)

Av. Jerônimo de Ornelas, 670 – Santana

90040-340 – Porto Alegre – RS

Fone: (51) 3330-3444 Fax: (51) 3330-2378

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

SÃO PAULO

Av. Rebouças, 1.073 – Jardins

05401-150 – São Paulo – SP

Fone: (11) 3062-3757* Fax: (011) 3062-2487

SAC 0800 703-3444

IMPRESSO NO BRASIL

PRINTED IN BRAZIL

Para David Nelson e Gary Morin da Sun Microsystems:
Dedicamos este livro a vocês – nossos amigos e nossos
mentores – a quem devemos nosso relacionamento de
uma década com a Sun.

Foi uma honra ter trabalhado com vocês e seus colegas na
Sun quando Java e o ramo de educação em Java
nasceram.

Harvey e Paul Deitel

Prefácio

Nunca mais viver em fragmentos. Apenas se conectar.
Edward Morgan Forster

Bem-vindo a *Java Como Programar, Quarta Edição*, e ao estimulante mundo da programação com a plataforma *Java™ 2, Standard Edition*. Este livro foi escrito por um velho e um jovem. O velho (HMD; Massachusetts Institute of Technology 1967) programa e/ou ensina programação há 40 anos. O jovem (PJD; MIT 1991) programa e/ou ensina programação há 22 anos e é tanto um *Sun Certified Java Programmer* como um *Sun Certified Java Developer*. O velho programa e ensina a partir de sua experiência; o jovem faz isso a partir de uma reserva inesgotável de energia. O velho quer clareza; o jovem quer desempenho. O velho busca elegância e beleza; o jovem quer resultados. Reunimo-nos para produzir um livro o qual esperamos que você ache informativo, desafiante e divertido.

Em novembro de 1995, comparecemos a uma conferência de Internet/World Wide Web em Boston para ouvir sobre Java. Um representante da Sun Microsystems falou sobre Java em um salão de convenções lotado. Tivemos sorte de conseguir assentos. Durante essa apresentação, vimos o futuro da programação se revelar. A primeira edição de *Java Como Programar* nascia naquele momento e foi publicada como o primeiro livro-texto de ciência da computação sobre Java do mundo.

O mundo de Java está desenvolvendo-se tão rápido que *Java Como Programar, Quarta Edição* está sendo publicado menos de cinco anos depois da primeira edição. Isso cria desafios e oportunidades imensas para nós como autores, para nosso editor – a Prentice Hall, para instrutores, alunos e profissionais.

Antes de Java aparecer, estávamos convencidos de que C++ substituiria C como a principal linguagem de desenvolvimento de aplicativos e a principal linguagem de programação de sistemas durante a próxima década. Entretanto, a combinação da World Wide Web com Java agora aumenta a importância da Internet no planejamento e na implementação estratégicos de sistemas de informações. Muitas organizações querem integrar a Internet “transparentemente” em seus sistemas de informações. Java é mais apropriada que C++ para esse propósito.

Novos recursos em *Java Como Programar, Quarta Edição*

Esta edição contém muitos recursos novos e aprimoramentos, incluindo:

- *Apresentação totalmente a cores.* O livro está totalmente a cores. O uso total de cores permite aos leitores ver os exemplos de saída assim como eles aparecem em um monitor colorido. Colorimos todo o código Java de acordo com a sintaxe, como boa parte dos ambientes de desenvolvimento de Java faz hoje em dia.

Nossas convenções para colorir de acordo com a sintaxe são as seguintes:

comentários aparecem em verde
palavras-chave aparecem em azul escuro

constantes e valores literais aparecem em azul claro
nomes de classes, métodos e variáveis aparecem em preto

- “**Limpeza do código**”. Este é o nosso novo termo para o processo que usamos para converter todos os programas do livro para um leiaute mais aberto e com comentários melhorados. Agrupamos o código dos programas em pedaços pequenos e bem documentados. Isto aumenta enormemente a legibilidade do código – um objetivo especialmente importante para nós, já que esta edição tem mais de 25.000 linhas de código.
- **Ajuste.** Realizamos um ajuste substancial no conteúdo do livro, com base em nossas próprias anotações de correntes do uso extensivo no ensino durante nossos seminários de Java para profissionais. Além disso, uma eminent equipe de revisores leu a terceira edição do livro e nos forneceu seus comentários e críticas. Existem literalmente milhares de melhorias de ajuste fino em relação à terceira edição.
- **Pensando em objetos.** Este estudo de caso opcional apresenta um *projeto orientado a objetos* com a *Unified Modeling Language* (a UML). Muitos capítulos nesta edição terminam com uma seção “Pensando em objetos” na qual apresentamos gradualmente uma introdução à orientação a objetos. Nosso objetivo nestas seções é ajudá-lo a desenvolver uma maneira de pensar orientada a objetos, para ser capaz de implementar sistemas mais substanciais. Estas seções também apresentam a *Unified Modeling Language* (UML). A UML é uma linguagem gráfica que permite às pessoas que constroem sistemas (p. ex., arquitetos de *software*, engenheiros de sistemas e programadores) representar seus projetos orientados a objetos usando uma notação comum. A seção “Pensando em objetos” do Capítulo 1 apresenta conceitos básicos e terminologia. Os Capítulos 2 a 13, 15 e 22 (o 22 está no CD) e os Apêndices G, H e I (também no CD) incluem seções “Pensando em objetos” opcionais que apresentam um estudo de caso substancial de um elevador, orientado a objetos, que aplica as técnicas de *projeto orientado a objetos*. Os Apêndices G, H e I implementam integralmente o projeto do estudo de caso em código Java. Este estudo de caso ajudará a prepará-lo para os tipos de projetos substanciais que você provavelmente irá encontrar nas empresas. Se você é um estudante e seu instrutor não planeja incluir este estudo de caso em seu curso, você poderá querer ler o estudo de caso por si mesmo. Acreditamos que valerá a pena o seu esforço de percorrer este projeto grande e desafiador. O material apresentado nas seções do estudo de caso reforça o material abordado nos capítulos correspondentes. Você experimentará uma introdução sólida ao projeto orientado a objetos com a UML. Além disso, você irá aperfeiçoar seus conhecimentos de leitura de código passeando por um programa em Java com 3.465 linhas de código, cuidadosamente escrito e bem documentado, que resolve completamente o problema apresentado no estudo de caso.
- **Descobrindo padrões de projeto.** Estas seções opcionais apresentam padrões populares de projeto orientado a objetos em uso atualmente. A maioria dos exemplos fornecidos neste livro contém menos de 150 linhas de código. Pequenos exemplos como estes normalmente não exigem um processo de projeto muito detalhado. Entretanto, alguns programas, como o nosso estudo de caso opcional de simulação de elevador, são mais complexos – eles podem precisar de milhares de linhas de código. Sistemas maiores, como sistemas de controle de caixas eletrônicos automáticos ou de controle de tráfego aéreo, podem conter milhões, ou até centenas de milhões de linhas de código. O projeto eficaz é crucial para a construção adequada de tais sistemas complexos. Ao longo da última década, o setor de engenharia de *software* fez progressos significativos no campo de *padrões de projeto* – arquiteturas de eficiência comprovada para construir *softwares* orientados a objetos flexíveis e fáceis de manter¹. O uso de padrões de projeto pode reduzir substancialmente a complexidade do processo de projetar. Apresentamos diversos padrões de projeto em Java, mas estes padrões de projeto podem ser implementados em qualquer linguagem orientada a objetos, como C++, C# ou Visual Basic. Descrevemos diversos padrões de projeto usados pela Sun Microsystems na API de Java.. Usamos padrões de projeto em muitos programas neste livro, que identificaremos em nossas seções “Descobrindo padrões de projeto”. Estes programas fornecem exemplos do uso de padrões de projeto para construir *softwares* orientados a objetos confiável e robusto.
- **Capítulo 22 (no CD), Java Media Framework (JMF) e JavaSound.** Este capítulo faz uma introdução aos recursos para áudio e vídeo de Java, aperfeiçoando nossa cobertura de multimídia do Capítulo 18. Com o *Java*

¹ Gamma, Erich, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software*. (Massachussets: Addison-Wesley, 1995)

va Media Framework, um programa Java pode reproduzir mídias de áudio e vídeo e capturar áudio e vídeo a partir de dispositivos como microfones e câmeras de vídeo. O JMF permite criar aplicações com *streaming media*, nas quais um programa Java envia, através da Internet, para outros computadores, áudio ou vídeo ao vivo ou gravados e, então, estes outros computadores reproduzem a mídia à medida que ela é recebida através da rede. As APIs de *JavaSound* permitem manipular sons em MIDI (*Musical Instrument Digital Interface*) e mídia capturada (i.e., mídia recebida de um dispositivo como um microfone). O capítulo termina com uma aplicação de processamento de MIDI substancial, que permite aos usuários gravar arquivos MIDI ou selecionar arquivos MIDI para reproduzir. Os usuários podem criar sua própria música MIDI interagindo com o teclado de sintetizador simulado pela aplicação. A aplicação pode sincronizar a reprodução das notas em um arquivo MIDI com o pressionar das teclas no teclado do sintetizador simulado – de forma semelhante a uma pianola. [Nota: os Capítulos 18 e 22 oferecem um conjunto substancial de exercícios. Cada capítulo também tem uma seção especial que contém projetos de multimídia interessantes e desafiadores. Estas são apenas sugestões para projetos mais importantes. As soluções não são fornecidas para estes exercícios adicionais nem no *Instructor's Manual* nem na *Java 2 Multimedia Cyber Classroom*.]

- **Redes avançadas baseadas em TCP/IP.** Incluímos um novo exemplo de coroamento no Capítulo 17 que apresenta o *multicasting* para enviar informações para grupos de clientes na rede. Este estudo de caso do Deitel Messenger emula muitas das aplicações populares de envio instantâneo de mensagens atuais, as quais permitem aos usuários de computadores se comunicar com amigos, parentes e colegas de trabalho através da Internet. Este programa Java cliente/servidor de 1.130 linhas, com *multithreading*, usa a maioria das técnicas apresentadas até este ponto no livro.
- **Apêndice J (no CD), Oportunidades de carreira.** Este apêndice detalhado apresenta serviços para a carreira profissional disponíveis na Internet. Exploramos os serviços *on-line* para a carreira profissional do ponto de vista do empregador e do empregado. Sugerimos *sites* para os quais você pode enviar pedidos de emprego, procurar trabalho e analisar candidatos (se você está interessado em contratar alguém). Também analisamos serviços que constroem páginas de recrutamento diretamente nas empresas de *e-business*. Um de nossos revisores nos disse que ele recém havia feito uma procura de emprego utilizando amplamente a Internet e que este capítulo teria ampliado em muito sua busca.
- **Apêndice K (no CD), Unicode.** Este apêndice dá uma visão geral do padrão *Unicode*. À medida que os sistemas de computação evoluíram no mundo inteiro, os fabricantes de computadores desenvolveram representações numéricas de conjuntos de caracteres e símbolos especiais para as linguagens locais faladas em países diferentes. Em alguns casos, representações diferentes foram desenvolvidas para as mesmas linguagens. Conjuntos de caracteres tão díspares tornaram difícil a comunicação entre sistemas de computação. Java suporta o padrão Unicode (mantido por uma organização sem fins lucrativos chamada *Unicode Consortium*), que define um conjunto único de caracteres com valores numéricos únicos para caracteres e símbolos especiais da maioria das linguagens faladas. Este apêndice discute o padrão Unicode, dá uma visão geral do *sítio* na Web do *Unicode Consortium* (unicode.org) e mostra um exemplo em Java que exibe “Welcome” em oito línguas diferentes!
- **Java 2 Plug-in transferido para o Capítulo 3.** Os estudantes gostam de ver resultados imediatos quando executam seus programas em Java. Isto é difícil se estes programas são *applets* Java que são executados em navegadores da Web. A maioria dos navegadores atuais (com a exceção do Netscape Navigator 6) não suporta *applets* Java 2 diretamente, de modo que os estudantes precisam testar seus programas *applets* com o utilitário *appletviewer*. A Sun Microsystems fornece o Java 2 Plug-in para permitir que os *applets* Java 2 sejam executados em um navegador que não suporta Java 2. A discussão do Java Plug-in conduz o estudante através das etapas necessárias para executar um *applet* nos navegadores da Web atuais.
- **Capítulo 22 e Apêndices E a K no CD.** Existem tantos assuntos novos abordados nesta nova edição que nós não conseguimos colocar todos eles dentro do livro! No CD que acompanha este livro, você irá encontrar os seguintes capítulos e apêndices: o Capítulo 22, e os Apêndices E, F, G, H, I, J e K.

- **Capítulos transferidos para Advanced Java™ 2 Platform How to Program.** Quatro capítulos de Java Como Programar, Terceira edição, foram transferidos para nosso novo livro *Advanced Java 2 Platform How to Program* e aperfeiçoados enormemente. Estes capítulos são: Conectividade do banco de dados Java (JDBC), *Servlets*, Invocação remota de métodos (RMI) e JavaBeans. *Advanced Java 2 Platform How to Program* aborda cada um destes tópicos em maior profundidade. Apresentamos o índice de *Advanced Java 2 Platform How to Program* em breve.

Alguns comentários para instrutores

Um mundo de orientação a objetos

Quando escrevemos a primeira edição de *Java: Como Programar*, as universidades ainda estavam dando ênfase à programação procedural em linguagens como Pascal e C. Os cursos de ponta estavam usando a linguagem orientada a objetos C++, mas estes cursos em geral estavam misturando uma quantidade substancial de programação procedural com programação orientada a objetos – algo que C++ lhe permite fazer, mas Java não. Por ocasião da terceira edição de *Java: Como Programar*, muitas universidades estavam mudando de C++ para Java em suas disciplinas introdutórias e os instrutores estavam enfatizando uma abordagem de programação orientada a objetos pura. Paralelamente a esta atividade, a comunidade de engenharia de *software* estava padronizando sua abordagem à modelagem de sistemas orientados a objetos com UML e o movimento em direção a padrões de projeto estava criando forma. *Java: Como Programar* tem muitos públicos, de modo que projetamos o livro para ser adaptável. Em particular, incluímos mais de 200 páginas de material opcional que apresenta o projeto orientado a objetos, a UML e os padrões de projeto e apresenta um estudo de caso substancial em projeto e programação orientados a objetos. Este material está cuidadosamente distribuído ao longo do livro para permitir aos instrutores enfatizar o projeto orientado a objetos com padrão empresarial em seus cursos.

Os alunos gostam de Java

Os alunos ficam altamente motivados pelo fato de que estão aprendendo uma linguagem de ponta (Java) e um paradigma da programação de ponta (programação orientada a objetos) que lhes serão imediatamente úteis enquanto estiverem no ambiente universitário e quando entrarem em um mundo em que a Internet e a World Wide Web têm uma importância enorme. Os alunos descobrem rapidamente que eles podem fazer coisas incríveis com Java, de modo que estão dispostos a despender o esforço extra. Java ajuda os programadores a libertar sua criatividade. Vemos isto nos cursos de Java que a Deitel & Associates ministra. Assim que nossos alunos entram no laboratório, não podemos mais segurá-los. Eles experimentam e exploram avidamente partes das bibliotecas de classes de Java que ainda não cobrimos em aula. Produzem aplicativos que vão muito além de qualquer coisa que tenhamos alguma vez tentado fazer em nossos cursos de introdução a C e C++. E eles nos falam sobre projetos que eles “mal podem esperar” para experimentar após o curso.

Enfoque do livro

Nosso objetivo era claro – produzir um livro de Java para os cursos introdutórios de nível universitário em programação de computador para alunos com pouca ou nenhuma experiência em programação, e ainda oferecer a profundidade e o tratamento rigoroso de teoria e prática exigidos por cursos tradicionais de nível superior e que satisfaça as necessidades dos profissionais. Para atender a esses objetivos, produzimos um livro abrangente, porque nosso texto também ensina pacientemente os princípios básicos de programação de computadores e da linguagem Java (i.e., tipos de dados, estruturas de controle, métodos, *arrays*, recursividade e outros tópicos “tradicionais” de programação); apresenta paradigmas de programação fundamentais, incluindo programação baseada em objetos, programação orientada a objetos, programação movida por eventos e programação concorrente; e fornece um tratamento detalhado das bibliotecas de classe Java.

Evolução de Java Como Programar

Java Como Programar (primeira edição) foi o primeiro livro universitário de ciência da computação sobre Java do mundo. Nós o escrevemos logo depois de *C How to Program: Second Edition* e *C++ Como Programar*. Centenas

de milhares de estudantes universitários e profissionais aprenderam C, C++ e Java a partir desses textos. Com a publicação, em agosto de 2001, *Java Como Programar, Quarta Edição* passou a ser utilizado por centenas de universidades e milhares de corporações e organizações governamentais em todo o mundo. A Deitel & Associates, Inc. ministrava cursos de Java internacionalmente para milhares de alunos enquanto estávamos escrevendo as diversas edições de *Java Como Programar*. Monitoramos cuidadosamente a eficácia destes cursos e ajustamos esses materiais de acordo.

Conceitualização de Java

Acreditamos em Java. Sua concepção (e liberação para o público em 1995) pela Sun Microsystems, os criadores de Java, foi brilhante. A Sun baseou a nova linguagem nas duas linguagens de implementação mais amplamente utilizadas do mundo, C e C++. Isso imediatamente deu a Java uma enorme base de programadores altamente qualificados que estavam implementando a maioria dos novos sistemas operacionais, sistemas de comunicações, sistemas de banco de dados, aplicativos de computador pessoal e *software* de sistemas do mundo. A Sun removeu os recursos mais confusos, mais complexos e mais propensos a erro de C/C++ (como ponteiros, sobrecarga de operadores e herança múltipla, entre outros). Eles mantiveram a linguagem concisa, removendo os recursos de uso especial que eram utilizados apenas por pequenos segmentos da comunidade de programação. Tornaram a linguagem verdadeiramente portável de modo que seja apropriada para a implementação de aplicativos baseados na Internet e na World Wide Web e incluíram os recursos de que as pessoas realmente precisam, como *strings*, imagens gráficas, componentes de interface gráfica com o usuário, tratamento de exceções, *multithreading*, multimídia (áudio, imagens, animação e vídeo), processamento de arquivos, processamento de banco de dados, redes cliente/servidor baseadas na Internet e na World Wide Web e computação distribuída, e estruturas de dados pré-empacotadas. Depois, disponibilizaram a linguagem *gratuitamente* para milhões de programadores em potencial no mundo inteiro.

2,5 milhões de desenvolvedores Java

Java foi lançada em 1995 como um meio de adicionar “conteúdo dinâmico” a páginas da World Wide Web. Em vez de páginas da Web apenas com texto e imagens gráficas estáticas, as páginas da Web das pessoas agora podiam “ganhar vida” com áudio, vídeos, animações, interatividade – e, em breve, imagens tridimensionais. Mas vimos muito mais em Java do que isso. Os recursos de Java são precisamente do que as empresas e as organizações precisam para atender aos requisitos de processamento de informações atuais. Assim, imediatamente vimos em Java o potencial para se tornar uma das principais linguagens de programação de uso geral do mundo. Na verdade, Java revolucionou o desenvolvimento de *softwares* com código orientado a objetos que usa intensivamente multimídia, independente de plataforma, para aplicações convencionais, baseadas em Internet, Intranet e Extranet e *applets*. Java agora conta com 2,5 milhões de desenvolvedores no mundo inteiro – uma façanha surpreendente, considerando que ela está disponível publicamente há apenas seis anos. Nenhuma outra linguagem de programação jamais conquistou uma base de desenvolvedores tão grande como esta em tão pouco tempo.

Permitindo aplicações e comunicações baseadas em multimídia

O campo da informática nunca viu qualquer coisa parecida com a “explosão” da Internet/World Wide Web/Java que está ocorrendo atualmente. As pessoas querem se comunicar. As pessoas precisam se comunicar. Com certeza elas têm feito isso desde o início da civilização, mas as comunicações através dos computadores têm sido na maioria limitadas a dígitos, caracteres alfabéticos e caracteres especiais. Atualmente, estamos no meio de uma revolução da multimídia. As pessoas querem transmitir imagens e querem que essas imagens sejam coloridas. Querem transmitir vozes, sons, clipes de áudio e vídeo completos, com movimento e cores (e não querem nada menos do que a qualidade de um DVD). Em algum momento, as pessoas insistirão na transmissão tridimensional com imagens em movimento. Nossas televisões planas e bidimensionais atuais por fim serão substituídas por versões tridimensionais que transformarão nossas salas de estar em “teatros de arena”. Os atores realizarão seus papéis como se estivéssemos assistindo teatro ao vivo. Nossas salas de estar serão transformadas em estádios em miniatura. Nossos escritórios profissionais permitirão videoconferências entre colegas do outro lado do mundo como se estivessem sentados em torno de uma mesa de conferência. As possibilidades são intrigantes e Java está desempenhando um papel fundamental para tornar muitas delas realidade.

A abordagem de ensino

Java Como Programar, Quarta Edição contém uma rica coleção de exemplos, exercícios e projetos trazidos de muitos campos para fornecer ao aluno uma oportunidade de resolver problemas interessantes do mundo real. O livro se concentra nos princípios da boa engenharia de *software* e enfatiza principalmente a clareza do programa. Evitamos terminologia obscura e especificações de sintaxe em favor de ensinar por meio de exemplos. Nossos exemplos de código foram testados em plataformas populares Java. Somos educadores que ensinam tópicos estritamente práticos em salas de aulas de empresas no mundo inteiro. O texto enfatiza a boa pedagogia.

Aprendendo Java através da abordagem de código ativo (Live Code™)

O livro está repleto de exemplos de código ativo (Live Code™). Esse é o foco da maneira como ensinamos e escrevemos sobre programação e também o foco de cada uma de nossas *Cyber Classrooms* multimídia e dos cursos de treinamento baseados na Web. Cada novo conceito é apresentado no contexto de um programa (*applet* ou aplicativo) Java completo, que funciona, imediatamente seguido por uma ou mais capturas de tela para mostrar a saída do programa. Chamamos esse estilo de ensinar e escrever de nossa **abordagem baseada em código ativo (live-code™)**. Utilizamos a linguagem para ensinar a linguagem. Ler esses programas (mais de 25.000 linhas de código) é muito parecido com instalá-los e executá-los em um computador.

Java e Swing já no Capítulo Dois!

Java Como Programar, Quarta Edição “vai direto ao assunto”, com programação orientada a objetos, aplicativos e componentes GUI no estilo do Swing já a partir do Capítulo 2! As pessoas nos dizem que isso é um passo arriscado, mas os alunos de Java realmente querem “ir direto à luta”. Há muita coisa a ser feita em Java; portanto, vamos direto a elas! Java não é de modo algum trivial, mas é divertido programar com ela e os alunos podem ver resultados imediatos. Os alunos podem pôr rapidamente em execução programas gráficos, com animações, baseados em multimídia, com uso intensivo de áudio, *multithreading*, com uso intensivo de bancos de dados e baseados em redes, por meio das extensas bibliotecas de classes de “componentes reutilizáveis” de Java. Eles podem implementar projetos impressionantes. Eles normalmente são mais criativos e produtivos em um curso de um ou dois semestres do que é possível em cursos introdutórios de C e C++.

Acesso à World Wide Web

Todo o código de *Java Como Programar* está no CD que acompanha este livro e está disponível na Internet no *site* da Deitel & Associates, Inc., www.deitel.com. Execute cada programa à medida que for lendo o texto. Faça alterações nos exemplos de código e veja o que acontece. Veja como o compilador Java “reclama” quando você faz vários tipos de erros. Veja imediatamente os efeitos de se fazer alterações no código. Uma excelente maneira de aprender programação é programando. [Este material possui direitos autorais. Sinta-se à vontade para utilizá-lo enquanto você estuda Java, mas você não pode republicar qualquer parte dele sem permissão explícita dos autores e da Prentice Hall.]

Objetivos

Cada capítulo inicia com uma declaração de objetivos. Essa declaração diz ao aluno o que esperar e dá ao aluno a oportunidade de, após a leitura do capítulo, determinar se alcançou esses objetivos. Isso dá confiança ao aluno e é uma fonte de reforço positivo.

Citações

Após os objetivos de aprendizagem há citações. Algumas são humorísticas, algumas são filosóficas e outras oferecem idéias interessantes. Nossos alunos gostam de relacionar as citações com o material do capítulo. As citações merecem uma “segunda leitura” depois de você ler cada capítulo.

Visão geral

A seção *Visão geral* ajuda o aluno a ter um primeiro contato com o material “de cima para baixo.” Isto também ajuda os alunos a antecipar o que está por vir e definir um ritmo de aprendizagem confortável e eficaz.

25.576 linhas de código em 197 programas de exemplo (com saídas de programa)

Apresentamos os recursos de Java no contexto de programas Java completos e que funcionam. Os programas variam de apenas algumas linhas de código a exemplos substanciais com várias centenas de linhas de código (e 3.465 linhas

de código para o exemplo opcional de simulador de elevador orientado a objetos). Os alunos devem usar o código do programa do CD que acompanha este livro ou baixar o código de nosso site (www.deitel.com) e executar cada programa enquanto estuda esse programa no texto.

545 ilustrações/figuras

Incluímos grande número de gráficos, desenhos e saídas de programas. A discussão de estruturas de controle, por exemplo, apresenta fluxogramas cuidadosamente desenhados. [Nota: não ensinamos fluxogramas como uma ferramenta de desenvolvimento de programas, mas utilizamos uma breve apresentação baseada em fluxogramas para especificar a operação precisa de cada uma das estruturas de controle de Java.]

605 dicas de programação

Incluímos dicas de programação para ajudar os alunos a focalizar aspectos importantes do desenvolvimento de programas. Destacamos centenas dessas dicas na forma de *Boas práticas de programação*, *Erros comuns de programação*, *Dicas de teste e depuração*, *Dicas de desempenho*, *Dicas de portabilidade*, *Observações de engenharia de software* e *Observações de aparência e comportamento*. Essas dicas e práticas representam o melhor daquilo que recolhemos em seis décadas de experiência em programação e ensino. Uma de nossas alunas – estudante de matemática – disse-nos que ela sente que essa abordagem é como o destaque que recebem os axiomas, teoremas e corolários nos livros de matemática; fornece uma base sobre a qual construir um bom *software*.

97 Boas práticas de programação



Quando ensinamos cursos introdutórios, afirmamos que o “termo da moda” de cada curso é “clareza” e destacamos como Boas práticas de programação técnicas para escrever programas que sejam mais claros, mais compreensíveis, mais depuráveis e de manutenção mais fácil.

199 Erros comuns de programação



Os alunos que estão aprendendo uma linguagem tendem a cometer certos erros com freqüência. Focalizar a atenção dos alunos nesses Erros comuns de programação ajuda os alunos a não cometer os mesmos erros. Também ajuda a reduzir as longas filas ao lado das salas dos instrutores fora do horário de aula!

46 Dicas de teste e depuração



Quando projetamos pela primeira vez esse “tipo de dica”, achamos que as utilizariamos estritamente para dizer às pessoas como testar e depurar programas Java. De fato, muitas das dicas descrevem aspectos de Java que reduzem a probabilidade de “bugs” e assim simplificam o processo de teste e depuração.

67 Dicas de desempenho



Em nossa experiência, ensinar os alunos a escrever programas claros e compreensíveis é, de longe, o objetivo mais importante para um primeiro curso de programação. Mas os alunos querem escrever programas que rodem mais rápido, utilizem menos memória, exijam o menor número de acionamentos de teclas, ou impressionem por outros truques inteligentes. Os alunos realmente se preocupam com o desempenho. Eles querem saber o que podem fazer para “turbinar” seus programas. Então incluímos 67 Dicas de desempenho que destacam as oportunidades para melhorar o desempenho dos programas – fazendo com que os programas sejam executados mais rapidamente ou minimizando a quantidade de memória que eles ocupam.

24 Dicas de portabilidade



Uma das “razões para a fama” de Java é a portabilidade “universal”, de modo que alguns programadores supõem que, se eles implementam um aplicativo em Java, o aplicativo será automática e “perfeitamente” portável para todas as plataformas Java. Infelizmente, esse nem sempre é o caso. Incluímos as Dicas de portabilidade para ajudar os alunos a escrever código portável e fornecer sugestões sobre como Java alcança seu alto grau de portabilidade. Tínhamos muito mais dicas de portabilidade em nossos livros, C How to Program e C++ How to Program. Precisamos de menos Dicas de portabilidade em Java Como Programar porque Java foi projetada para ser portável de cima para baixo (em sua maior parte) – muito menos esforço é exigido por parte do programador de Java para alcançar a portabilidade do que com C ou C++.

134 Observações de engenharia de software



O paradigma de programação orientada a objetos exige uma completa reavaliação da maneira como construímos sistemas de software. Java é uma linguagem eficaz para realizar boa engenharia de software. As Observações de engenharia de software destacam questões arquitetônicas e de projeto que afetam a construção de sistemas de soft-

ware, especialmente sistemas de larga escala. Muito do que o aluno aprende aqui será útil em cursos de nível superior e nas empresas, quando o aluno começar a trabalhar com sistemas grandes e complexos do mundo real.



38 Observações de aparência e comportamento

Fornecemos Observações de aparência e comportamento para destacar convenções da interface gráfica com o usuário. Essas observações ajudam os alunos a projetar suas próprias interfaces gráficas com o usuário em conformidade com as normas da indústria.

Resumo (983 marcadores de resumo)

Cada capítulo termina com dispositivos pedagógicos adicionais. Apresentamos um resumo completo do capítulo, no estilo de lista de marcadores. Em média, há 42 itens de resumo por capítulo. Isso ajuda os alunos a revisar e reforçar os conceitos fundamentais.

Terminologia (2171 termos)

Incluímos na seção *Terminologia* uma lista alfabética dos termos importantes definidos no capítulo – novamente, mais reforço. Em média, há 95 termos por capítulo.

397 exercícios de auto-revisão e respostas (a contagem inclui partes separadas)

Extensos exercícios e respostas de auto-revisão são incluídos para auto-aprendizagem. Isso oferece ao aluno a oportunidade de adquirir confiança com o material e preparar-se para os exercícios regulares. Os alunos devem ser incentivados a fazer todos os exercícios de auto-revisão e verificar suas respostas.

779 exercícios (a contagem inclui partes separadas)

Cada capítulo termina com um conjunto de exercícios que inclui revisões simples de terminologia e conceitos importantes; escrever comandos de Java isolados; escrever pequenas partes de métodos e classes de Java; escrever métodos, classes, *applets* e aplicativos Java completos; e escrever projetos de diplomação importantes. O grande número de exercícios relacionados com uma ampla variedade de áreas permite aos instrutores montar seus cursos de acordo com as necessidades específicas de seu público e variar os trabalhos do curso em cada semestre. Os instrutores podem utilizar esses exercícios para formar deveres de casa, pequenos questionários e exames importantes. As soluções para a maioria dos exercícios estão incluídas no CD *Instructor's Manual*, que está *disponível somente para instrutores* através de seus representantes da Prentice-Hall. [Nota: não nos escreva solicitando o manual do instrutor. A distribuição desta publicação está limitada estritamente a professores universitários que estejam ensinando a partir do livro. Os instrutores podem obter o manual de soluções somente com seus representantes normais da Prentice-Hall. Lamentamos não poder fornecer as soluções para profissionais.] As soluções para aproximadamente metade dos exercícios encontram-se no CD *Java Multimedia Cyber Classroom: Fourth Edition*, que também faz parte de *The Complete Java 2 Training Course*. Para saber como fazer pedidos, visite o endereço www.deitel.com.

Aproximadamente 5300 entradas de índice (com aproximadamente 9500 referências a páginas)

Incluímos um extenso *Índice* no final do livro. Isso ajuda o aluno a localizar qualquer termo ou conceito por palavra-chave. O *Índice* é útil para as pessoas que estão lendo o livro pela primeira vez e é especialmente útil para os programadores profissionais que utilizam o livro como referência. Os termos nas seções *Terminologia* geralmente aparecem no *Índice* (junto com vários outros itens de cada capítulo). Os alunos podem utilizar o *Índice* junto com as seções *Terminologia* para certificar-se de que abordaram o material fundamental de cada capítulo.

“Dupla indexação” de todos os exemplos e exercícios de código ativo Java

Java Como Programar tem 197 exemplos de código ativo (*live code™*) e 1176 exercícios (incluindo partes). Muitos dos exercícios são problemas ou projetos desafiadores que exigem esforço substancial. Fornecemos “dupla indexação” de cada um dos exemplos de código ativo e da maioria dos projetos mais desafiadores. Para cada programa de código-fonte de Java no livro, colocamos o nome do arquivo com a extensão **.java**, como em **LoadAudioAndPlay.java** e o indexamos tanto alfabeticamente (nesse caso sob “L”) como um item de subíndice em “Exemplos”. Isso facilita a localização dos exemplos que utilizam recursos específicos. Os exercícios mais substanciais, como “Gerando e Percorrendo Labirintos”, são indexados alfabeticamente (neste caso sob “G”) e como um item de subíndice em “Exercícios”.

Bibliografia

Uma extensa bibliografia de livros, artigos e documentação sobre Java 2 da Sun Microsystems é incluída para incentivar leituras adicionais.

Software incluído com *Java Como Programar, Quarta Edição*

Existem diversos produtos Java disponíveis à venda. Entretanto, você não precisa deles para começar com Java. Escrevemos *Java Como Programar, Quarta Edição* usando somente o *Java 2 Software Development Kit* (J2SDK). Para sua conveniência, a versão 1.3.1 do J2SDK da Sun está incluída no CD que acompanha este livro. O J2SDK também pode ser baixado do site sobre Java da Sun Microsystems, java.sun.com. Com a cooperação da Sun, também pudemos incluir no CD um poderoso ambiente de desenvolvimento integrado (IDE – *integrated development environment*) para Java – o *Forté for Java Community Edition* da Sun Microsystems.

Forté for Java Community Edition é um IDE profissional escrito em Java que inclui uma ferramenta para projeto de interface gráfica com o usuário, editor de código, compilador, depurador visual e mais. J2SDK 1.3.1 deve ser instalado antes de reinstalar *Forté for Java Community Edition*. Se você tem qualquer dúvida sobre o uso deste software, leia a documentação de introdução ao *Forté* que está no CD. Forneceremos informações adicionais em nosso site da Web www.deitel.com.

O CD também contém os exemplos do livro e uma página da Web em HTML com *links* para o site na Web da Deitel & Associates, Inc., o site da Prentice Hall e os muitos sites listados nos apêndices. Se você tem acesso à Internet, esta página pode ser carregada em seu navegador para lhe dar acesso rápido a todos os recursos. Finalmente, o CD contém o Capítulo 22 e os Apêndices E a K.

Pacote complementar para *Java Como Programar, Quarta Edição*

Java Como Programar, Quarta Edição tem uma grande quantidade de material complementar para os instrutores que ensinam a partir do livro. O CD Instructor's Manual contém soluções para a grande maioria dos exercícios dos fins de capítulos e um banco de questões de múltipla escolha (aproximadamente duas por seção do livro). Além disso, fornecemos eslaides em PowerPoint® contendo todo o código e as figuras que estão no texto. Você tem a liberdade de modificar estes eslaides para atender às suas necessidades em sala de aula. A Prentice Hall fornece um Companion Web Site (www.prenhall.com/deitel) que inclui recursos para instrutores e alunos. Para os instrutores, o site tem um Administrador de Semestre para planejamento de cursos, *links* para os eslaides em PowerPoint e material de referência dos apêndices do livro (tais como a tabela de precedência dos operadores, conjuntos de caracteres e recursos na Web). Para os alunos, o site oferece objetivos do capítulo, exercícios do tipo verdadeiro/falso com realimentação instantânea, destaque do capítulo e materiais de referência. [Nota: não nos escreva solicitando o manual do instrutor. A distribuição desta publicação está limitada estritamente a professores universitários que estejam ensinando a partir do livro. Os instrutores podem obter o manual de soluções somente com seus representantes da Prentice-Hall.* Lamentamos não poder fornecer as soluções para profissionais.]

Java 2 Multimedia Cyber Classroom: Fourth Edition* (versões no CD e treinamento baseado na Web) e *The Complete Java 2 Training Course: Fourth Edition

Preparamos uma versão interativa em software, baseada em CD-ROM, de *Java Como Programar, Quarta Edição*, chamada *Java 2 Multimedia Cyber Classroom: Fourth Edition*. Ela está repleta de recursos de aprendizagem e referência. A *Cyber Classroom* é distribuída num mesmo pacote com o livro com um desconto de preço no *The Complete Java 2 Training Course: Fourth Edition*. Se você já possui o livro e gostaria de comprar o *Java 2 Multimedia Cyber Classroom: Fourth Edition* separadamente, visite o site www.informit.com/cyberclassrooms. O número ISBN para a *Java 2 Multimedia Cyber Classroom: Fourth Edition* é 0-13-064935-x. Todas as *Cyber Classrooms* da Deitel geralmente estão disponíveis nos formatos CD e treinamento baseado na Web.

O CD tem uma introdução com os autores dando uma visão geral dos recursos da *Cyber Classroom*. Os 197 exemplos de programas Java em código ativo do livro realmente “ganham vida” na *Cyber Classroom*. Se você estiver vendo um programa e quiser executá-lo, simplesmente clique no ícone de relâmpago e o programa será executado. Você imediatamente verá – e ouvirá, no caso de programas multimídia baseados em áudio – as saídas do programa. Se quiser modificar um programa e ver e ouvir os efeitos de suas alterações, simplesmente clique no ícone de disquete que faz com que o código-fonte seja “arras-

* N. de R. No Brasil, escreva para a Bookman Companhia Editora.

tado” do CD e “jogado” em um de seus próprios diretórios, de modo que você possa editar o texto, recompilar o programa e experimentar sua nova versão. Clique no ícone de áudio e Paul Deitel falará sobre o programa e irá “conduzi-lo” através do código.

A *Cyber Classroom* também fornece ajudas para navegação, incluindo extensos recursos de *hyperlink*. A *Cyber Classroom* baseia-se em um navegador, de modo que ela se lembra das seções que você visitou recentemente e permite que você se move para a frente ou para trás nestas seções. As milhares de entradas de índice contêm *hyperlinks* para suas ocorrências no texto. Você pode digitar um termo utilizando o recurso “*find*” e a *Cyber Classroom* localizará suas ocorrências por todo o texto. As entradas do Sumário são “ativas”, de modo que clicar em um nome de capítulo leva-o até este capítulo.

Os alunos nos dizem que eles gostam particularmente das centenas de problemas do livro resolvidos que estão incluídos com a *Cyber Classroom*. Estudar e executar esses programas extras é uma excelente maneira de os alunos aprimorarem sua experiência de aprendizagem.

Os alunos e usuários profissionais de nossa *Cyber Classroom* nos informam que gostam da interatividade e que a *Cyber Classroom* é uma referência eficaz por causa dos extensos recursos de *hyperlink* e outros recursos de navegação. Recebemos uma mensagem por correio eletrônico de uma pessoa dizendo que, por não poder arcar com os custos de cursar uma universidade, a *Cyber Classroom* foi a solução para suas necessidades educacionais.

Os professores nos dizem que seus alunos gostam de usar a *Cyber Classroom*, dedicam mais tempo ao curso e dominam mais o material que em cursos baseados somente em texto. Publicamos (e iremos publicar) muitos outros produtos *Cyber Classroom* e *Complete Training Course*. Visite também os endereços www.deitel.com ou www.prenhall.com/deitel para obter mais informações.

Advanced Java™ 2 Platform How to Program

Nosso livro complementar – *Advanced Java 2 Platform How to Program* – focaliza a *Java 2 Platform, Enterprise Edition (J2EE)*, apresenta recursos avançados da *Java 2 Platform Standard Edition* e apresenta a *Java 2 Platform, Micro Edition (J2ME)*. Este livro é destinado a desenvolvedores e universitários em cursos avançados que já conhecem Java e querem um tratamento e um entendimento mais profundos da linguagem. O livro apresenta nossa abordagem característica de “código ativo” (*live code™*) para programas completos que funcionam e contém mais de 37.000 linhas de código. Os programas são mais substanciais que os apresentados em *Java Como Programar, Quarta Edição*. O livro expande a cobertura de *Java Database Connectivity (JDBC)*, invocação remota de método (RMI), *servlets* e *JavaBeans* de *Java Como Programar, Quarta Edição*. O livro também aborda tecnologias emergentes e mais avançadas de Java, de interesse para desenvolvedores de aplicações corporativas. O Sumário de *Advanced Java 2 Platform How to Program* é: Capítulos – Introduction; Advanced Swing Graphical User Interface Components; Model-View-Controller; Graphics Programming with Java 2D and Java 3D; Case Study: A Java 2D Application; JavaBeans Component Model; Security; Java Database Connectivity (JDBC); Servlets; Java Server Pages (JSP); Case Study: Servlet and JSP Bookstore; Java 2 Micro Edition (J2ME) and Wireless Internet; Remote Method Invocation (RMI); Session Enterprise JavaBeans (EJBs) and Distributed Transactions; Entity EJBs; Java Message Service (JMS) and Message-Driven EJBs; Enterprise Java Case Study: Architectural Overview; Enterprise Java Case Study: Presentation and Controller Logic; Enterprise Java Case Study: Business Logic Part 1; Enterprise Java Case Study: Business Logic Part 2; Application Servers; Jini; JavaSpaces; Jiro; Java Management Extensions (JMX); Common Object Request Broker Architecture (CORBA): Part 1; Common Object Request Broker Architecture (CORBA): Part 2; Peer-to-Peer Networking; Apêndices – Creating Markup with XML; XML Document Type Definitions; XML Document Object Model (DOM); XSL: Extensible Style Sheet Transformations; Downloading and Installing J2EE 1.2.1; Java Community Process (JCP); Java Native Interface (JNI); Carrer Opportunities; Unicode.

Agradecimentos

Um dos grandes prazeres de escrever um livro é reconhecer os esforços das muitas pessoas cujos nomes não podem aparecer na capa, mas cujo trabalho duro, cuja cooperação, amizade e compreensão foram cruciais à produção do livro.

Outras pessoas na Deitel & Associates, Inc. dedicaram longas horas a esse projeto. Gostaríamos de reconhecer os esforços de nossos colegas em tempo integral na Deitel & Associates, Inc., Tem Nieto, Sean Santry, Jonathan Gadzik, Kate Steinbuhler, Rashmi Jayaprakash e Laura Treibick.

- Tem Nieto é graduado pelo Massachusetts Institute of Technology. Tem ministra seminários sobre XML, Java, Internet e Web, C, C++ e Visual Basic e trabalha conosco na redação de livros, desenvolvimento de cur-

sos e trabalhos de autoria em multimídia. É co-autor de *Internet & World Wide Web How to Program (Segunda Edição)*, *XML How to Program*, *Perl How to Program* e *Visual Basic 6 How to Program*. Em *Java Como Programar, Quarta Edição*, Tem foi co-autor dos Capítulos 11, 12, 13 e 21 e da Seção Especial no Capítulo 19.

- Sean Santry, graduado pelo Boston College (ciência da computação e filosofia) e co-autor de *Advanced Java 2 Platform How to Program*, editou o Capítulo 22, ajudou a atualizar os programas no Capítulo 15, projetou e implementou a aplicação para redes Deitel Messenger no Capítulo 17, ajudou a projetar o estudo de caso opcional sobre OOP/UML, revisou o estudo de caso opcional sobre padrões de projeto e revisou a implementação da simulação de elevador para o estudo de caso de OOP/UML.
- Jonathan Gadzik, graduado pela Columbia University School of Engineering and Applied Science (BS em ciência da computação) foi co-autor do estudo de caso opcional sobre OOP/UML e das seções opcionais. Também implementou o programa Java de 3.465 linhas que resolve completamente o exercício de simulação de elevador orientado a objetos apresentado no estudo de caso sobre OOP/UML.
- Kate Steinbuhler, graduada pelo Boston College em inglês e comunicações, foi co-autora do Apêndice J e administrou o processo de autorizações. Kate está se mudando para cursar a faculdade de direito na University of Pittsburgh – boa sorte, Kate! Obrigado por suas contribuições a três publicações da Deitel.
- Rashmi Jayaprakash, graduado pela Boston University com diploma em ciência da computação, foi co-autor do apêndice K, Unicode.
- Laura Treibick, graduada pela University of Colorado at Boulder em fotografia e multimídia, criou o encantador personagem do inseto animado para a implementação do estudo de caso sobre OOP/UML.

Também gostaríamos de agradecer aos participantes de nosso programa de estágios para estudantes universitários na Deitel & Associates, Inc.²

- Susan Warren, estudante (*Junior*) de ciência da computação na Brown University, e Eugene Izumo, estudante (*Sophomore*) de ciência da computação na Brown University, revisaram a *Quarta Edição* inteira; revisaram e atualizaram o Capítulo 22 e atualizaram o Apêndice A e o Apêndice B. Susan e Eugene também trabalharam em muitos dos materiais complementares do livro, incluindo as soluções dos exercícios, questões verdadeiro/falso para a *Java 2 Multimedia Cyber Classroom* e questões de múltipla escolha para o banco de questões do instrutor.
- Vincent He, estudante (*Senior*) de administração e ciência da computação no Boston College, foi co-autor do Capítulo 22 – um dos capítulos mais emocionantes e divertidos do livro! Temos a certeza de que você irá gostar da profusão de multimídia que Vincent criou para você.
- Liz Rockett, uma estudante (*Senior*) de inglês na Princeton University editou e atualizou o Capítulo 22.
- Chris Henson, graduado pela Brandeis University (ciência da computação e história), revisou o Capítulo 22.
- Christina Carney, uma estudante (*Senior*) de psicologia e comércio no Framingham State College, pesquisou e atualizou a bibliografia, ajudou a preparar o Prefácio e fez a pesquisa de URLs para o estudo de caso de OOP/UML e padrões de projeto.

² [O Programa de Estágios para Estudantes Universitários da Deitel & Associates, Inc. oferece um número limitado de vagas remuneradas para estudantes universitários da região de Boston que cursam ciência da computação, Tecnologia de Informação ou marketing. Os estudantes trabalham na matriz da nossa corporação em Sudbury, Massachusetts, em tempo integral durante os verões e em tempo parcial durante o ano letivo. Há vagas em tempo integral disponíveis para graduados. Para obter mais informações sobre este programa competitivo, entre em contato com Abey Deitel pelo endereço deitel@deitel.com e verifique nosso site na Web, www.deitel.com]

- Amy Gips, estudante (*Sophomore*) de *marketing* e finanças no Boston College, atualizou e adicionou URLs para *applets*, gráficos, Java 2D e Multimídia nos Apêndices A e B. Amy também pesquisou citações para o Capítulo 22 e ajudou a preparar o Prefácio.
- Varun Ganapathi, uma estudante (*Sophomore*) de ciência da computação e engenharia elétrica da Cornell University, atualizou o Apêndice F.
- Reshma Khilnani, um estudante (*Junior*) de ciência da computação e matemática no Massachusetts Institute of Technology, trabalhou com Rashmi no Apêndice sobre Unicode.

Tivemos a felicidade de trabalhar nesse projeto com a talentosa e dedicada equipe de profissionais de publicação da Prentice Hall. Apreciamos especialmente os esforços extraordinários de nossa editora de ciência da computação, Petra Recter, e de sua chefe – nossa mentora em publicações – Marcia Horton, editora-chefe da divisão de engenharia e de ciência da computação da Prentice-Hall. Camille Trentacoste fez um trabalho maravilhoso como gerente de produção.

A *Java 2 Multimedia Cyber Classroom: Fourth Edition* foi desenvolvida em paralelo com *Java Como Programar, Quarta Edição*. Apreciamos sinceramente o conhecimento técnico, o entendimento e o insaite sobre a “nova mídia” de nosso editor Mark Taub. Ele e nossa editora de mídia eletrônica, Karen Mclean, fizeram um trabalho notável trazendo *Java 2 Multimedia Cyber Classroom: Fourth Edition* para publicação sob um cronograma apertado. Michael Ruel fez um trabalho maravilhoso como gerente de projeto da *Cyber Classroom*.

Devemos agradecimentos especiais à criatividade de Tamara Newnam Cavallo (smart_art@earthlink.net), que fez o trabalho artístico para nossos ícones de dicas de programação e para a capa. Ela criou a encantadora criatura que compartilha com você as dicas de programação do livro.

Estamos sinceramente agradecidos pelos esforços de nossos revisores da quarta edição:

Revisores de Java Como Programar, Quarta Edição

Dibyendu Baksi (Sun Microsystems)

Tim Boudreau (Sun Microsystems)

Michael Bundschuh (Sun Microsystems)

Gary Ginstling (Sun Microsystems)

Tomas Pavek (Sun Microsystems)

Rama Roberts (Sun Microsystems)

Terry Hull (Sera Nova)

Ralph Johnson (co-autor em “quarteto” do livro, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995)

Cameron Skinner (Embarcadero Technologies; OMG)

Michael Chonoles (Lockheed Martin Adv. Concepts; OMG)

Brian Cook (The Technical Resource Connection; OMG)

Akram Al-Rawi (Zayed University)

Charley Bay (Fronte Range Community College)

Clint Bickmore (Fronte Range Community College)

Ron Braithwaite (Nutriware)

Columbus Brown (IBM)

Larry Brown (co-autor de *Core Web Programming*)

Dan Corkum (Trillium Software)

Jonathan Earl (Technical Training and Consulting)

Karl Frank (togethersoft.com)

Charles Fry (thesundancekid.org)

Kyle Gabhart (Objective Solutions)

Felipe Gaucho (Softexport)

Rob Gordon (SuffolkSoft, Inc.)

Michelle Guy (XOR)

Christopher Green (Cobrado Springs Technical Consulting Group)

Kevlin Henney (Curbralan Limited)

Ethan Henry (Sitraka Software)

Faisal Kaleem (Florida International University)
Rob Keily (SUNY)
Scott Kendall (Consultor, autor de UML)
Sachin Khana (Programador Java *free lancer*)
Michael-Franz Mannion (Desenvolvedor Java)
Julie McVicar (Oakland Community College)
Matt Mitton (Consultor)
Dan Moore (XOR)
Simon North (Synopsys)
Chetan Patel (Lexisnexis)
Brian Pontarelli (Consultor)
Kendall Scott (Consultor, autor de UML)
Craig Shofding (CAS Training Corp)
Spencer Roberts (Titus Corporation)
Toby Steel (CertaPay)
Stephen Tockey (Construx Software)
Kim Topley (Autor de *Core Java Foundation Classes* e de *Core Swing: Advanced Programming*, ambos publicados pela Prentice Hall)
Gustavo Toretti (Programador Java; Campinas University)
Michael Van Kleeck (Diretor de Tecnologia, Learning.com)
Dave Wagstaff (Sungard)

***Revisores de Java How to Program, Third Edition* após a publicação:**

Jonathan Earl (Technical Training and Consulting)
Harry Foxwell (Sun Microsystems)
Terry Hull (Sera Nova)
Ron McCarty (Penn State University Behrend Campus)
Bina Ramamurthy (SUNY Buffalo)
Vadim Tkachenko (Sera Nova)

Sob uma agenda apertada, eles examinaram minuciosamente cada aspecto do texto e fizeram incontáveis sugestões para aprimorar a exatidão e a completude da apresentação.

Apreciamos sinceramente seus comentários, críticas, correções e sugestões para aprimorar o texto. Envie toda a correspondência para:

deitel@deitel.com

Responderemos imediatamente. Bem, isso é tudo por enquanto. Bem-vindo ao mundo emocionante da programação em Java. Esperamos que você desfrute esse exame do desenvolvimento de aplicativos de computador de ponta. Boa sorte!

*Dr. Harvey M. Deitel
Paul J. Deitel*

Sobre os autores

Dr. Harvey M. Deitel, executivo-chefe da Deitel & Associates, Inc., tem 40 anos de experiência no campo de computação incluindo extensa experiência acadêmica e empresarial. É um dos principais instrutores e conferencistas de ciência da computação do mundo. Dr. Deitel obteve os graus de B.S. e M.S. do Massachusetts Institute of Technology e um Ph.D. da Boston University. Ele tem 20 anos de experiência em ensino universitário, conquistando estabilidade e servindo como chefe do Departamento de Ciência da Computação do Boston College antes de fundar a Deitel & Associates, Inc. com seu filho Paul J. Deitel. É autor ou co-autor de várias dezenas de livros e pacotes de multimídia e atualmente está escrevendo vários outros. Com traduções publicadas em japonês, russo, espanhol, italiano, chinês básico, chinês tradicional, coreano, francês, polonês e português, os textos do Dr. Deitel ganharam re-

conhecimento internacional, tendo proferido seminários profissionais internacionalmente para importantes corporações, organizações do governo e várias áreas militares.

Paul J. Deitel, executivo-chefe de tecnologia da Deitel & Associates, Inc., é graduado pela Sloan School of Management do Massachusetts Institute of Technology, onde estudou Tecnologia da Informação. Por meio da Deitel & Associates, Inc. proferiu cursos sobre Internet e World Wide Web e aulas de linguagens de programação para clientes empresariais, incluindo Sun Microsystems, MCE², IBM, BEA Systems, Visa International, Progress Software, Boeing, Fidelity, Hitachi, Cap Gemini, Compaq, Art Technology, White Sands Missile Range, Digital Equipment Corporation, NASA no Kennedy Space Center, National Severe Storm Laboratory, Rogue Wave Software, Lucent Technologies, Computervision, Cambridge Technology Partners, Adra Systems, Entergy, CableData Systems, Banyan, Stratus, Concord Communications e muitas outras organizações. Fez conferências sobre Java e C++ para o Boston Chapter da Association for Computing Machinery e ministrou cursos via satélite por meio de um empreendimento cooperativo entre a Deitel & Associates, Inc., a Prentice Hall e a Technology Education Network. Ele e seu pai, o Dr. Harvey M. Deitel, são os autores de livros-texto sobre ciência da computação mais vendidos.

Sobre a Deitel & Associates, Inc.

A Deitel & Associates, Inc. é uma organização internacionalmente reconhecida de treinamento corporativo e criação de conteúdo especializada em tecnologia de *software* para Internet/World Wide Web, tecnologia de *software* para *e-business* e *e-commerce* e educação em linguagens de programação de computadores. A Deitel & Associates, Inc. é membro do World Wide Web Consortium. A empresa oferece cursos sobre programação da Internet e da World Wide Web, tecnologia de objetos e das principais linguagens de programação. Os fundadores da Deitel & Associates, Inc. são o Dr. Harvey M. Deitel e Paul J. Deitel. Entre os clientes da empresa incluem-se algumas das maiores empresas de computadores do mundo, órgãos do governo, áreas do serviço militar e organizações comerciais. Por meio de sua parceria editorial com a Prentice Hall, a Deitel & Associates, Inc. publica livros de programação de ponta, livros profissionais, *Cyber Classrooms* multimídia baseadas em CD-ROM interativo, cursos via satélite e cursos baseados na Web. A Deitel & Associates, Inc. e os autores podem ser contatados por correio eletrônico em:

deitel@deitel.com

Para aprender mais sobre a Deitel & Associates, Inc., suas publicações e seu currículo corporativo mundial *on-site*, visite:

www.deitel.com

Aqueles que quiserem comprar livros, *Cyber Classrooms*, *Complete Training Courses* e cursos de treinamento baseado na Web podem fazer isso através da página:

www.deitel.com

Os pedidos em grandes volumes feitos por corporações e instituições acadêmicas devem ser feitos diretamente à Prentice Hall.

O World Wide Web Consortium (W3C)



A Deitel & Associates, Inc. é sócia do *World Wide Web Consortium (W3C)*. O W3C foi fundado em 1994 “para desenvolver protocolos comuns para a evolução da World Wide Web”. Como sócios do W3C, participamos do *W3C Advisory Committee* (nossa representante no *Advisory Committee* é nosso executivo-chefe de tecnologia, Paul J. Deitel). Os membros do *Advisory Committee* ajudam a definir “direção estratégica” para o W3C através de reuniões em todo o mundo. As organizações associadas também ajudam a desenvolver recomendações de padrões para tecnologias da Web (tais como HTML, XML e muitas outras) através da participação em atividades e grupos de trabalho do W3C. A participação como sócias no W3C é reservada a empresas e grandes organizações. Para obter informações sobre como se tornar sócio do W3C, visite o endereço www.w3.org/Consortium/Prospectus/Joining.

Sumário

1	Introdução aos computadores, à Internet e à Web	49
1.1	Introdução	50
1.2	O que é um computador?	54
1.3	A organização de um computador	54
1.4	Evolução dos sistemas operacionais	55
1.5	Computação pessoal distribuída e computação cliente/servidor	56
1.6	Linguagens de máquina, linguagens <i>assembler</i> e linguagens de alto nível	56
1.7	A história de C++	57
1.8	A história de Java	58
1.9	Bibliotecas de classes Java	59
1.10	Outras linguagens de alto nível	60
1.11	Programação estruturada	60
1.12	A Internet e a World Wide Web	61
1.13	Princípios básicos de um ambiente Java típico	62
1.14	Notas gerais sobre Java e este livro	64
1.15	Pensando em objetos: introdução à tecnologia de objetos e à Unified Modeling Language	66
1.16	Descobrindo padrões de projeto: introdução	70
1.17	Um passeio pelo livro	71
1.18	(Opcional) Um passeio pelo estudo de caso sobre projeto orientado a objetos com a UML	82
1.19	(Opcional) Um passeio pelas seções “Descobrindo padrões de projeto”	85
2	Introdução a aplicativos Java	93
2.1	Introdução	94
2.2	Um primeiro programa em Java: imprimindo uma linha de texto	94
2.2.1	Compilando e executando seu primeiro aplicativo Java	99

22 SUMÁRIO

2.3	Modificando nosso primeiro programa em Java	99
2.3.1	Exibindo uma única linha de texto com múltiplas instruções	99
2.3.2	Exibindo múltiplas linhas de texto com uma única instrução	100
2.4	Exibindo texto em uma caixa de diálogo	100
2.5	Outro aplicativo Java: adicionando inteiros	105
2.6	Conceitos de memória	110
2.7	Aritmética	111
2.8	Tomada de decisão: operadores de igualdade e operadores relacionais	114
2.9	(Estudo de caso opcional) Pensando em objetos: examinando a definição do problema	120

3 Introdução a applets Java 135

3.1	Introdução	136
3.2	<i>Applets</i> de exemplo do Java 2 Software Development Kit	137
3.2.1	O <i>applet</i> TicTacToe	137
3.2.2	O <i>applet</i> DrawTest	140
3.2.3	O <i>applet</i> Java2D	141
3.3	Um <i>applet</i> Java simples: desenhando um <i>string</i>	141
3.3.1	Compilando e executando o <i>WelcomeApplet</i>	146
3.4	Dois <i>applets</i> mais simples: desenhando <i>strings</i> e linhas	148
3.5	Outro <i>applet</i> Java: adicionando números de ponto flutuante	150
3.6	Visualizando <i>applets</i> em um navegador da Web	156
3.6.1	Visualizando <i>applets</i> no Netscape Navigator 6	156
3.6.2	Visualizando <i>applets</i> em outros navegadores usando o Java Plug-in	157
3.7	Recursos para <i>applets</i> Java na Internet e na World Wide Web	159
3.8	(Estudo de caso opcional) Pensando em objetos: identificando as classes em uma definição de problema	
	160	

4 Estruturas de controle: parte 1 172

4.1	Introdução	173
4.2	Algoritmos	173
4.3	Pseudocódigo	174
4.4	Estruturas de controle	174
4.5	A estrutura de seleção if	176
4.6	A estrutura de seleção if/else	178
4.7	A estrutura de repetição while	181
4.8	Formulando algoritmos: estudo de caso 1 (repetição controlada por contador)	182
4.9	Formulando algoritmos com refinamento passo a passo de cima para baixo: estudo de caso 2 (repetição controlada por sentinelas)	187
4.10	Formulando algoritmos com refinamento passo a passo de cima para baixo: estudo de caso 3 (estruturas de controle aninhadas)	194
4.11	Operadores de atribuição	199
4.12	Operadores de incremento e decremento	199
4.13	Tipos de dados primitivos	202
4.14	(Estudo de caso opcional) Pensando em objetos: identificando os atributos das classes	203

5	Estruturas de controle: parte 2	215
5.1	Introdução	216
5.2	Princípios básicos da repetição controlada por contador	216
5.3	A estrutura de repetição for	218
5.4	Exemplos com a estrutura for	221
5.5	A estrutura de seleção múltipla switch	226
5.6	A estrutura de repetição do/while	231
5.7	As instruções break e continue	233
5.8	As instruções rotuladas break e continue	235
5.9	Operadores lógicos	236
5.10	Resumo de programação estruturada	242
5.11	(Estudo de caso opcional) Pensando em objetos: identificando estados e atividades dos objetos	248
6	Métodos	259
6.1	Introdução	260
6.2	Módulos de programas em Java	260
6.3	Os métodos da classe Math	261
6.4	Métodos	262
6.5	Definições de métodos	263
6.6	Promoção de argumentos	269
6.7	Pacotes da Java API	270
6.8	Geração de números aleatórios	272
6.9	Exemplo: um jogo de azar	275
6.10	Duração dos identificadores	282
6.11	Regras de escopo	283
6.12	Recursão	285
6.13	Exemplo que utiliza recursão: a série de Fibonacci	288
6.14	Recursão <i>versus</i> iteração	293
6.15	Sobrecarga de métodos	295
6.16	Métodos da classe JApplet	297
6.17	(Estudo de caso opcional) Pensando em objetos: identificando operações de classes	298
7	Arrays	316
7.1	Introdução	317
7.2	<i>Arrays</i>	317
7.3	Declarando e alocando <i>arrays</i>	319
7.4	Exemplos com <i>arrays</i>	320
7.4.1	Alocando um <i>array</i> e inicializando seus elementos	320
7.4.2	Utilizando uma lista de inicializadores para inicializar os elementos de um <i>array</i>	321
7.4.3	Calculando o valor a armazenar em cada elemento de um <i>array</i>	322
7.4.4	Somando os elementos de um <i>array</i>	324
7.4.5	Utilizando histogramas para exibir dados de <i>arrays</i> graficamente	325
7.4.6	Utilizando os elementos de um <i>array</i> como contadores	326

24 SUMÁRIO

7.4.7	Utilizando <i>arrays</i> para analisar resultados de pesquisas	327
7.5	Referências e parâmetros por referência	330
7.6	Passando <i>arrays</i> para métodos	330
7.7	Classificando <i>arrays</i>	333
7.8	Pesquisando <i>arrays</i> : pesquisa linear e pesquisa binária	335
7.8.1	Pesquisando um <i>array</i> com pesquisa linear	335
7.8.2	Pesquisando um <i>array</i> com pesquisa binária	337
7.9	Arrays multidimensionais	342
7.10	(Estudo de caso opcional) Pensando em objetos: colaboração entre objetos	347

8 Programação baseada em objetos 372

8.1	Introdução	373
8.2	Implementando um tipo abstrato de dados Time com uma classe	374
8.3	Escopo de classe	381
8.4	Controlando o acesso a membros	381
8.5	Criando pacotes	382
8.6	Inicializando objetos de classe: construtores	386
8.7	Utilizando construtores sobrecarregados	387
8.8	Utilizando os métodos <i>set</i> e <i>get</i>	391
8.8.1	Executando um <i>applet</i> que usa pacotes definidos pelo programador	399
8.9	Reutilização de <i>software</i>	400
8.10	Variáveis de instância final	401
8.11	Composição: objetos como variáveis de instância de outras classes	402
8.12	Acesso de pacote	405
8.13	Utilizando a referência this	407
8.14	Finalizadores	413
8.15	Membros de classe static	414
8.16	Abstração de dados e encapsulamento	418
8.16.1	Exemplo: tipo abstrato de dados fila	419
8.17	(Estudo de caso opcional) Pensando em objetos: começando a programar as classes para a simulação do elevador	420

9 Programação orientada a objetos 430

9.1	Introdução	431
9.2	Superclasses e subclasses	433
9.3	Membros protected	436
9.4	Relacionamento entre objetos de superclasse e objetos de subclasse	436
9.5	Construtores e finalizadores em subclasses	442
9.6	Conversão implícita de objeto de subclasse para objeto de superclasse	445
9.7	Engenharia de <i>software</i> com herança	446
9.8	Composição <i>versus</i> herança	447
9.9	Estudo de caso: ponto, círculo, cilindro	447
9.10	Introdução ao polimorfismo	454
9.11	Campos de tipo e instruções switch	454
9.12	Vinculação dinâmica de método	454

9.13	Métodos e classes final	455
9.14	Superclasses abstratas e classes concretas	455
9.15	Exemplos de polimorfismo	456
9.16	Estudo de caso: um sistema de folha de pagamento utilizando polimorfismo	457
9.17	Novas classes e vinculação dinâmica	465
9.18	Estudo de caso: herança de interface e implementação	465
9.19	Estudo de caso: criando e utilizando interfaces	472
9.20	Definições de classe interna	478
9.21	Notas sobre definições de classe interna	488
9.22	Classes invólucro de tipo para tipos primitivos	488
9.23	(Estudo de caso opcional) Pensando em objetos: incorporando herança à simulação do elevador	488
9.24	(Opcional) Descobrindo padrões de projeto: apresentando os padrões de criação, estruturais e comportamentais de projeto	494
9.24.1	Padrões de criação de projeto	495
9.24.2	Padrões estruturais de projeto	497
9.24.3	Padrões comportamentais de projeto	498
9.24.4	Conclusão	499
9.24.5	Recursos na Internet e na World Wide Web	500
10	Strings e caracteres	508
10.1	Introdução	509
10.2	Fundamentos de caracteres e <i>strings</i>	509
10.3	Construtores de String	510
10.4	Os métodos length , charAt e getChars de String	512
10.5	Comparando Strings	513
10.6	O método hashCode de String	518
10.7	Localizando caracteres e <i>substrings</i> em Strings	519
10.8	Extraindo <i>substrings</i> a partir de Strings	521
10.9	Concatenando Strings	522
10.10	Métodos diversos de String	523
10.11	Utilizando o método valueOf de String	525
10.12	O método intern de String	526
10.13	A classe StringBuffer	528
10.14	Construtores de StringBuffer	528
10.15	Os métodos length , capacity , setLength e ensureCapacity de StringBuffer	529
10.16	Os métodos charAt , setCharAt , getChars e reverse de StringBuffer	531
10.17	Os métodos append de StringBuffer	532
10.18	Métodos de inserção e de exclusão de StringBuffer	534
10.19	Exemplos da classe Character	535
10.20	A classe StringTokenizer	542
10.21	Simulação de embaralhamento e distribuição de cartas	544
10.22	(Estudo de caso opcional) Pensando em objetos: tratamento de eventos	548
11	Imagens gráficas e Java2D	563
11.1	Introdução	564
11.2	Contextos gráficos e objetos gráficos	566

11.3	Controle de cor	567
11.4	Controle de fontes	572
11.5	Desenhando linhas, retângulos e elipses	578
11.6	Desenhando arcos	582
11.7	Desenhando polígonos e polilinhas	584
11.8	A API Java2D	587
11.9	Formas de Java2D	587
11.10	(Estudo de caso opcional) Pensando em objetos: projetando interfaces com a UML	592

12 Componentes da interface gráfica com o usuário: parte 1 603

12.1	Introdução	604
12.2	Visão geral do Swing	605
12.3	JLabel	607
12.4	Modelo de tratamento de eventos	610
12.5	JTextField e JPasswordField	612
12.5.1	Como funciona o tratamento de eventos	616
12.6	JButton	617
12.7	JCheckBox e JRadioButton	619
12.8	JComboBox	624
12.9	JList	627
12.10	Listas de seleção múltipla	629
12.11	Tratamento de eventos de mouse	631
12.12	Classes adaptadoras	635
12.13	Tratamento de eventos de teclado	640
12.14	Gerenciadores de leiaute	643
12.14.1	FlowLayout	644
12.14.2	BorderLayout	646
12.14.3	GridLayout	649
12.15	Painéis	650
12.16	(Estudo de caso opcional) Pensando em objetos: casos de uso	652

13 Componentes da interface gráfica com o usuário: parte 2 667

13.1	Introdução	668
13.2	JTextArea	669
13.3	Criando uma subclasse personalizada de JPanel	672
13.4	Criando uma subclasse autocontida de JPanel	675
13.5	JSlider	680
13.6	Janelas	684
13.7	Projetando programas que podem ser executados como <i>applets</i> ou aplicativos	685
13.8	Utilizando menus com <i>frames</i>	690
13.9	Utilizando JPopupMenu	697
13.10	Aparência e comportamento plugável	700
13.11	Utilizando JDesktopPane e JInternalFrame	704

13.12	Gerenciadores de leiaute	707
13.13	O gerenciador de leiaute BoxLayout	707
13.14	O gerenciador de leiaute CardLayout	711
13.15	O gerenciador de leiaute GridBagLayout	714
13.16	As constantes RELATIVE e REMAINDER de GridBagConstraints	719
13.17	(Estudo de caso opcional) Pensando em objetos: <i>Model-View-Controller</i>	722
13.18	(Opcional) Descobrindo padrões de projeto: padrões de projeto usados nos pacotes java.awt e javax.swing	726
13.18.1	Padrões de criação de projeto	726
13.18.2	Padrões estruturais de projeto	727
13.18.3	Padrões comportamentais de projeto	729
13.18.4	Conclusão	732
14	Tratamento de exceções	739
14.1	Introdução	740
14.2	Quando deve ser utilizado o tratamento de exceções	742
14.3	Outras técnicas de tratamento de erros	742
14.4	Princípios básicos de tratamento de exceções em Java	742
14.5	Blocos <i>try</i>	743
14.6	Disparando uma exceção	744
14.7	Capturando uma exceção	744
14.8	Um exemplo de tratamento de exceções: divisão por zero	746
14.9	Disparando novamente uma exceção	751
14.10	Cláusula throws	751
14.11	Construtores, finalizadores e tratamento de exceções	756
14.12	Exceções e herança	757
14.13	O bloco finally	757
14.14	Utilizando printStackTrace e getMessage	761
15	Multithreading	768
15.1	Introdução	769
15.2	Classe Thread : uma visão geral dos métodos de Thread	771
15.3	Estatos de <i>threads</i> : ciclo de vida de uma <i>thread</i>	772
15.4	Prioridades de <i>threads</i> e escalonamento de <i>threads</i>	773
15.5	Sincronização de <i>threads</i>	777
15.6	Relacionamento produtor/consumidor sem sincronização de <i>threads</i>	778
15.7	Relacionamento produtor/consumidor com sincronização de <i>threads</i>	783
15.8	Relacionamento produtor/consumidor: o <i>buffer</i> circular	787
15.9	<i>Threads daemon</i>	795
15.10	A interface Runnable	796
15.11	Grupos de <i>threads</i>	801
15.12	(Estudo de caso opcional) Pensando em objetos: <i>multithreading</i>	802
15.13	(Opcional) Descobrindo padrões de projeto: padrões de projeto simultâneos	809

16	Arquivos e fluxos	817
16.1	Introdução	818
16.2	Hierarquia de dados	818
16.3	Arquivos e fluxos	820
16.4	Criando um arquivo de acesso seqüencial	824
16.5	Lendo dados de um arquivo de acesso seqüencial	834
16.6	Atualizando arquivos de acesso seqüencial	845
16.7	Arquivos de acesso aleatório	846
16.8	Criando um arquivo de acesso aleatório	846
16.9	Gravando dados aleatoriamente em um arquivo de acesso aleatório	851
16.10	Lendo dados seqüencialmente de um arquivo de acesso aleatório	856
16.11	Exemplo: um programa de processamento de transações	860
16.12	A classe <code>File</code>	875
17	Redes	890
17.1	Introdução	891
17.2	Manipulando URIs	892
17.3	Lendo um arquivo em um servidor da Web	897
17.4	Estabelecendo um servidor simples com soquetes de fluxo	900
17.5	Estabelecendo um cliente simples com soquetes de fluxo	901
17.6	Interação cliente/servidor com conexões de soquete de fluxo	902
17.7	Interação cliente/servidor sem conexão com datagramas	912
17.8	Tic-Tac-Toe cliente/servidor utilizando um servidor com múltiplas <i>threads</i>	918
17.9	Segurança e a rede	931
17.10	Servidor e cliente de bate-papo DeitelMessenger	931
17.10.1	<code>DeitelMessengerServer</code> e classes de suporte	931
17.10.2	Cliente DeitelMessenger e classes de suporte	940
17.11	(Opcional) Descobrindo padrões de projeto: padrões de projeto utilizados nos pacotes <code>java.io</code> e <code>java.net</code>	956
17.11.1	Padrões de criação de projeto	956
17.11.2	Padrões estruturais de projeto	957
17.11.3	Padrões de arquitetura de projeto	958
17.11.4	Conclusão	960
18	Multimídia: imagens, animação, áudio e vídeo	966
18.1	Introdução	967
18.2	Carregando, exibindo e dimensionando imagens	968
18.3	Animando uma série de imagens	970
18.4	Personalizando <code>LogoAnimator</code> através de parâmetros de <i>applet</i>	974
18.5	Mapas de imagens	977
18.6	Carregando e reproduzindo clipes de áudio	980

18.7	Recursos na Internet e na World Wide Web	982
19	Estruturas de dados	988
19.1	Introdução	989
19.2	Classes auto-referenciais	989
19.3	Alocação dinâmica de memória	990
19.4	Listas encadeadas	991
19.5	Pilhas	1001
19.6	Filas	1004
19.7	Árvores	1007
20	Pacote de utilitários Java e manipulação de bits	1034
20.1	Introdução	1035
20.2	A classe <code>Vector</code> e a interface <code>Enumeration</code>	1035
20.3	A classe <code>Stack</code>	1042
20.4	A classe <code>Dictionary</code>	1046
20.5	A classe <code>Hashtable</code>	1046
20.6	A classe <code>Properties</code>	1052
20.7	A classe <code>Random</code>	1057
20.8	Manipulação de bits e os operadores sobre bits	1058
20.9	A classe <code>BitSet</code>	1071
21	Coleções	1081
21.1	Introdução	1082
21.2	Visão geral das coleções	1083
21.3	A classe <code>Arrays</code>	1083
21.4	A interface <code>Collection</code> e a classe <code>Collections</code>	1087
21.5	Listas	1087
21.6	Algoritmos	1093
21.6.1	O algoritmo <code>sort</code>	1093
21.6.2	O algoritmo <code>shuffle</code>	1095
21.6.3	Os algoritmos <code>reverse</code> , <code>fill</code> , <code>copy</code> , <code>max</code> e <code>min</code>	1097
21.6.4	O algoritmo <code>binarySearch</code>	1099
21.7	Conjuntos	1100
21.8	Mapeamentos	1103
21.9	Empacotadores de sincronização	1104
21.10	Empacotadores não-modificáveis	1105
21.11	Implementações abstratas	1105
21.12	(Opcional) Descobrindo padrões de projeto: padrões de projeto usados no pacote <code>java.util</code>	1106
21.12.1	Padrões de criação de projeto	1106
21.12.2	Padrões comportamentais de projeto	1106
21.12.3	Conclusão	1107

22	Java Media Framework e Java Sound (no CD)	1112
22.1	Introdução	1113
22.2	Reproduzindo mídia	1114
22.3	Formatando e salvando mídia capturada	1123
22.4	<i>Streaming</i> com RTP	1135
22.5	Java Sound	1148
22.6	Reproduzindo amostras de áudio	1148
22.7	Musical Instrument Digital Interface (MIDI)	1154
22.7.1	Reprodução de MIDI	1155
22.7.2	Gravação de MIDI	1160
22.7.3	Síntese de MIDI	1163
22.7.4	A classe <code>MidiDemo</code>	1166
22.8	Recursos na Internet e na World Wide Web	1181
22.9	(Estudo de caso opcional) Pensando em objetos: animação e som na visão	1181
A	Demos de Java	1206
A.1	Introdução	1206
A.2	Os <i>sites</i>	1206
B	Recursos para Java	1208
B.1	Recursos	1208
B.2	Produtos	1209
B.3	FAQs	1210
B.4	Tutoriais	1210
B.5	Revistas	1210
B.6	<i>Applets</i> Java	1210
B.7	Multimídia	1211
B.8	Grupos de notícias	1211
C	Tabela de precedência de operadores	1213
D	Conjunto de caracteres ASCII	1215
E	Sistemas de numeração (no CD)	1216
E.1	Introdução	1217
E.2	Abreviando números binários como números octais e números hexadecimais	1219
E.3	Convertendo números octais e números hexadecimais em números binários	1221
E.4	Convertendo de binário, octal ou hexadecimal para decimal	1221
E.5	Convertendo de decimal para binário, octal ou hexadecimal	1221
E.6	Números binários negativos: notação em complemento de dois	1223
F	Criando documentação em HTML com javadoc (no CD)	1229
F.1	Introdução	1230
F.2	Comentários de documentação	1231

F.3	Documentando o código-fonte Java	1231
F.4	<code>javadoc</code>	1238
F.5	Arquivos produzidos por <code>javadoc</code>	1238
G	Eventos e interfaces <i>listener</i> do elevador (no CD)	1242
G.1	Introdução	1242
G.2	Eventos	1242
G.3	<i>Listeners</i>	1245
G.4	Diagramas de componentes revisitados	1248
H	Modelo do elevador (no CD)	1251
H.1	Introdução	1251
H.2	A classe <code>ElevatorModel</code>	1251
H.3	As classes <code>Location</code> e <code>Floor</code>	1258
H.4	A classe <code>Door</code>	1261
H.5	A classe <code>Button</code>	1264
H.6	A classe <code>ElevatorShaft</code>	1265
H.7	As classes <code>Light</code> e <code>Bell</code>	1271
H.8	A classe <code>Elevator</code>	1274
H.9	A classe <code>Person</code>	1282
H.10	Diagramas de componentes revisitados	1288
H.11	Conclusão	1288
I	Visão do elevador (no CD)	1290
I.1	Introdução	1290
I.2	Objetos da classe	1305
I.3	Constantes da classe	1307
I.4	Construtor da classe	1308
I.5	Tratamento de eventos	1309
I.5.1	Tipos de eventos <code>ElevatorMoveEvent</code>	1310
I.5.2	Tipos de eventos <code>PersonMoveEvent</code>	1310
I.5.3	Tipos de eventos <code>DoorEvent</code>	1311
I.5.4	Tipos de eventos <code>ButtonEvent</code>	1311
I.5.5	Tipos de eventos <code>BellEvent</code>	1311
I.5.6	Tipos de eventos <code>LightEvent</code>	1311
I.6	Diagrama de componentes revisitado	1311
I.7	Conclusão	1312
J	Oportunidades para a carreira profissional (no CD)	1313
J.1	Introdução	1314
J.2	Recursos para quem procura trabalho	1315
J.3	Oportunidades <i>on-line</i> para empregadores	1316
J.3.1	Anunciando trabalho <i>on-line</i>	1318
J.3.2	Problemas com o recrutamento através da Web	1319
J.3.3	Diversidade no local de trabalho	1320

32 SUMÁRIO

J.4	Serviços de recrutamento	1320
J.4.1	Testando funcionários em potencial <i>on-line</i>	1321
J.5	<i>Sites</i> para a carreira profissional	1322
J.5.1	<i>Sites</i> abrangentes para a carreira profissional	1322
J.5.2	Cargos técnicos	1322
J.5.3	Cargos “sem-fio”	1324
J.5.4	Empreitando <i>on-line</i>	1324
J.5.5	Cargos executivos	1326
J.5.6	Estudantes e profissionais jovens	1326
J.5.7	Outros serviços <i>on-line</i> para a carreira profissional	1327
J.6	Recursos na Internet e na World Wide Web	1327
K	Unicode® (no CD)	1336
K.1	Introdução	1337
K.2	Formatos de transformação Unicode	1338
K.3	Caracteres e hieróglifos	1338
K.4	Vantagens/desvantagens de Unicode	1339
K.5	<i>Site</i> do Consórcio Unicode na Web	1340
K.6	Usando Unicode	1340
K.7	Intervalos de caracteres	1342
	Bibliografia	1347
	Índice	1351

Ilustrações

1	Introdução aos computadores, à Internet e à Web	
1.1	Ambiente Java típico.	63
2	Introdução a aplicativos Java	
2.1	Um primeiro programa em Java.	95
2.2	Executando <code>Welcome1</code> em um Command Prompt do Microsoft Windows 2000.	99
2.3	Imprimindo em uma mesma linha com múltiplas instruções.	100
2.4	Imprimindo em múltiplas linhas de texto com uma única instrução.	101
2.5	Algumas seqüências comuns de escape.	101
2.6	Exibindo múltiplas linhas em uma caixa de diálogo.	102
2.7	Uma janela de exemplo do Netscape Navigator com os componentes GUI.	103
2.8	Caixa de diálogo de mensagem.	104
2.9	Um programa de adição “em ação”.	105
2.10	Caixa de diálogo de entrada.	108
2.11	Caixa de diálogo de mensagem personalizada com a versão de quatro argumentos do método <code>showMessageDialog</code> .	109
2.12	Constantes de <code>JOptionPane</code> para o diálogo de mensagem.	110
2.13	Posição da memória que mostra o nome e o valor da variável <code>number1</code> .	110
2.14	Posições da memória depois que os valores para <code>number1</code> e <code>number2</code> foram armazenados.	111
2.15	Posições da memória após calcular a soma de <code>number1</code> e <code>number2</code> .	111
2.16	Operadores aritméticos.	112
2.17	Precedência de operadores aritméticos.	113
2.18	Ordem em que um polinômio de segundo grau é avaliado.	114
2.19	Operadores de igualdade e operadores relacionais.	115
2.20	Utilizando operadores de igualdade e operadores relacionais.	115
2.21	Precedência e associatividade dos operadores discutidos até agora.	120
2.22	Pessoa caminhando em direção ao elevador no primeiro andar.	122
2.23	Pessoa andando no elevador para o segundo andar.	122
2.24	Pessoa se afastando do elevador.	123

3 Introdução a applets Java

3.1	Os exemplos do diretório applets .	138
3.2	Exemplo de execução do <i>applet tictaetoe</i> .	139
3.3	Selecionando Reload no menu Applet do appletviewer .	139
3.4	Exemplo de execução do <i>applet DrawTest</i> .	140
3.5	Exemplo de execução do <i>applet Java2D</i> .	142
3.6	Um primeiro <i>applet</i> em Java e a saída na tela do <i>applet</i> .	142
3.7	WelcomeApplet.html carrega a classe WelcomeApplet da Fig. 3.6 no appletviewer .	146
3.8	<i>Applet</i> que exibe múltiplos <i>strings</i> .	148
3.9	WelcomeApplet2.html carrega a classe WelcomeApplet2 da Fig. 3.8 no appletviewer .	148
3.10	Desenhando <i>strings</i> e linhas.	149
3.11	O arquivo WelcomeLines.html , que carrega a classe WelcomeLines da Fig. 3.10 no appletviewer .	150
3.12	Um programa de adição “em ação”.	150
3.13	AdditionApplet.html carrega a classe AdditionApplet da Fig. 3.12 no appletviewer .	151
3.14	<i>Applet</i> da 3.10 sendo executado no Netscape Navigator 6.	157
3.15	Janela do Java Plug-in HTML Converter .	158
3.16	Selecionando o diretório que contém os arquivos HTML a converter.	158
3.17	Selecionando o gabarito usado para converter os arquivos HTML.	159
3.18	Diálogo de confirmação após a conversão ter terminado.	159
3.19	Lista de substantivos na definição do problema.	160
3.20	Representando uma classe na UML.	162
3.21	Diagrama de classes mostrando as associações entre classes.	163
3.22	Tipos de multiplicidade.	163
3.23	Diagrama de classes para o modelo do elevador.	164
3.24	Diagrama de objetos de um prédio vazio em nosso modelo de elevador.	165

4 Estruturas de controle: parte 1

4.1	Fluxograma da estrutura de seqüência de Java.	175
4.2	Palavras-chave de Java.	176
4.3	Fluxograma da estrutura de seleção única if .	177
4.4	Fluxograma da estrutura de seleção dupla if/else .	178
4.5	Fluxograma da estrutura de repetição while .	183
4.6	Algoritmo em pseudocódigo que utiliza repetição controlada por contador para resolver o problema de média da turma.	183
4.7	Programa de média da turma com repetição controlada por contador.	183
4.8	Algoritmo em pseudocódigo que utiliza repetição controlada por sentinelas para resolver o problema da média da turma.	189
4.9	Programa de média da turma com repetição controlada por sentinelas.	190
4.10	Pseudocódigo para o problema dos resultados de um exame.	196
4.11	Programa Java para o problema dos resultados de um exame.	196
4.12	Operadores aritméticos de atribuição.	199
4.13	Operadores de incremento e decremento.	200
4.14	A diferença entre pré-incrementar e pós-incrementar.	200
4.15	Precedência e associatividade dos operadores discutidos até agora.	202

4.16	Os tipos de dados primitivos em Java.	202
4.17	Palavras e frases desritivas da definição do problema.	204
4.18	Classes com atributos.	206

5 Estruturas de controle: parte 2

5.1	Repetição controlada por contador.	217
5.2	Repetição controlada por contador com a estrutura for .	219
5.3	Componentes de um cabeçalho de estrutura for típico.	220
5.4	Fluxograma de uma estrutura de repetição for típica.	222
5.5	Soma com a estrutura for .	223
5.6	Cálculo de juros compostos com a estrutura for .	224
5.7	Um exemplo que utiliza switch .	227
5.8	A estrutura de seleção múltipla switch .	230
5.9	Utilizando a estrutura de repetição do/while .	231
5.10	Fluxograma da estrutura de repetição do/while .	232
5.11	Utilizando a instrução break em uma estrutura for .	233
5.12	Utilizando a instrução continue em uma estrutura for .	234
5.13	Utilizando uma instrução rotulada break em uma estrutura aninhada for .	235
5.14	Utilizando uma instrução rotulada continue em uma estrutura aninhada for .	236
5.15	Tabela-verdade para o operador && (E lógico).	238
5.16	Tabela-verdade para o operador (OU lógico).	239
5.17	Tabela-verdade para o operador lógico booleano exclusivo OU (^).	240
5.18	Tabela-verdade para o operador ! (NÃO lógico).	240
5.19	Demonstrando os operadores lógicos.	240
5.20	Precedência e associatividade dos operadores discutidos até agora.	242
5.21	As estruturas de controle com entrada única/saída única de Java.	243
5.22	Regras para formar programas-estruturados.	244
5.23	O fluxograma mais simples.	244
5.24	Aplicando repetidamente a Regra 2 da Fig. 5.22 ao fluxograma mais simples.	245
5.25	Aplicando a Regra 3 da Fig. 5.22 ao fluxograma mais simples.	245
5.26	Blocos de construção empilhados, aninhados e sobrepostos.	246
5.27	Um fluxograma não-estruturado.	246
5.28	Diagrama de mapa de estados para objetos FloorButton e ElevatorButton	248
5.29	Diagrama de atividades para um objeto Person .	248
5.30	Diagrama de atividades para um objeto Elevator .	249

6 Métodos

6.1	Relacionamento hierárquico método do trabalhador/método do patrão.	261
6.2	Métodos da classe Math .	262
6.3	Utilizando o método square definido pelo programador.	264
6.4	O método maximum definido pelo programador.	268
6.5	Promoções permitidas para tipos de dados primitivos.	270
6.6	Os pacotes da Java API.	271
6.7	Inteiros aleatórios deslocados e escalonados.	272
6.8	Lançando um dado de seis faces 6000 vezes.	273
6.9	Programa que simula o jogo <i>craps</i> .	275

6.10	Um exemplo de escopo.	284
6.11	Avaliação recursiva de 5!	287
6.12	Calculando fatoriais com um método recursivo.	287
6.13	Gerando números de Fibonacci recursivamente.	289
6.14	Conjunto de chamadas recursivas para o método fibonacci (f neste diagrama).	292
6.15	Resumo de exemplos e exercícios com recursão no texto.	294
6.16	Utilizando métodos sobrecarregados.	295
6.17	Mensagens de erro do compilador geradas por métodos sobrecarregados com listas de parâmetros idênticas e tipos diferentes de valores devolvidos.	297
6.18	Métodos de JApplet que o contêiner de <i>applets</i> chama durante a execução de um <i>applet</i> .	298
6.19	Frases com verbos para cada classe no simulador	299
6.20	Classe com atributos e operações	300
6.21	Valores para os lados dos triângulos no Exercício 6.15.	310
6.22	Torres de Hanói para o caso com quatro discos.	314

7 Arrays

7.1	Um <i>array</i> de 12 elementos.	318
7.2	Precedência e associatividade dos operadores discutidos até agora.	319
7.3	Inicializando os elementos de um <i>array</i> com zero.	320
7.4	Inicializando os elementos de um <i>array</i> com uma declaração.	322
7.5	Gerando valores para serem colocados em elementos de um <i>array</i> .	323
7.6	Calculando a soma dos elementos de um <i>array</i> .	324
7.7	Programa que imprime histogramas.	325
7.8	Programa de jogo de dados que utiliza <i>arrays</i> em vez de switch .	326
7.9	Um programa simples de análise de pesquisa de alunos.	328
7.10	Passando <i>arrays</i> e elementos individuais de <i>arrays</i> para métodos.	331
7.11	Classificando um <i>array</i> com o algoritmo <i>bubble sort</i> .	333
7.12	Pesquisa linear de um <i>array</i> .	336
7.13	Pesquisa binária de um <i>array</i> ordenado.	338
7.14	Um <i>array</i> bidimensional com três linhas e quatro colunas.	342
7.15	Inicializando <i>arrays</i> multidimensionais.	343
7.16	Exemplo de utilização de <i>arrays</i> bidimensionais.	345
7.17	Frases com verbos para cada classe, mostrando comportamentos na simulação.	348
7.18	Colaborações no sistema do elevador.	349
7.19	Diagrama de colaborações para uma pessoa pressionando o botão de andar.	350
7.20	Diagrama de colaborações para passageiros entrando no elevador e saindo dele.	351
7.21	Os 36 resultados possíveis ao se lançar dois dados.	356
7.22	Determine o que faz este programa.	356
7.23	Determine o que faz este programa.	358
7.24	Comandos para o gráfico de tartaruga.	359
7.25	Os oito movimentos possíveis do cavalo.	360
7.26	Os 22 quadrados eliminados ao se colocar a rainha no canto esquerdo superior.	362
7.27	Regras para ajustar as posições da tartaruga e da lebre.	366
7.28	Códigos de operação da Simpletron Machine Language (SML).	367
7.29	Programa em SML que lê dois inteiros e calcula sua soma	367
7.30	Programa em SML que lê dois inteiros e determina qual é o maior	368

7.31	Comportamento de diversas instruções SML no Simpletron.	370
7.32	Um exemplo de <i>dump</i> .	370
8	Programação baseada em objetos	
8.1	Implementação do tipo abstrato de dados Time1 como uma classe.	374
8.2	Usando um objeto da classe Time1 em um programa.	378
8.3	Tentativa incorreta de acessar membros privados da classe Time1 .	382
8.4	Colocando a classe Time1 em um pacote para reutilização.	383
8.5	Usando a classe definida pelo programador Time1 num pacote.	385
8.6	Classe Time2 com construtores sobrecarregados.	387
8.7	Usando construtores sobrecarregados para inicializar objetos da classe Time2 .	390
8.8	Classe Time3 com métodos <i>set</i> e <i>get</i> .	392
8.9	Usando os métodos <i>set</i> e <i>get</i> da classe Time3 .	395
8.10	Conteúdo de TimeTest5.jar .	399
8.11	Inicializando uma variável final .	401
8.12	Mensagem de erro do compilador como resultado da não-inicialização de INCREMENT .	402
8.13	Classe Date .	403
8.14	Classe Employee com referências para objetos-membros.	404
8.15	Demonstrando um objeto com uma referência para objetos-membros.	405
8.16	Acesso de pacote a membros de uma classe.	406
8.17	Usando a referência this implícita e explicitamente.	407
8.18	Classe Time4 usando this para permitir chamadas de métodos encadeadas.	409
8.19	Concatenando chamadas a métodos.	412
8.20	Classe Employee que usa uma variável de classe static para manter uma contagem do número de objetos Employee presentes na memória.	415
8.21	Usando uma variável de classe static para manter a contagem do número de objetos de uma classe.	416
8.22	Diagrama de classes completo com notações de visibilidade.	421
9	Programação orientada a objetos	
9.1	Alguns exemplos simples de herança nos quais a subclasse “é uma” superclasse.	434
9.2	Hierarquia de herança para MembroDaComunidade em uma universidade.	435
9.3	Parte de uma hierarquia de classes Forma .	435
9.4	Definição da classe Point .	437
9.5	Definição da classe Circle .	438
9.6	Atribuindo referências para subclasse a referências para superclasse.	440
9.7	Definição da classe Point para demonstrar quando os construtores e finalizadores são chamados.	443
9.8	Definição da classe Circle para demonstrar quando os construtores e finalizadores são chamados.	443
9.9	Ordem na qual os construtores e finalizadores são chamados.	445
9.10	Definição da classe Point .	447
9.11	Testando a classe Point .	448
9.12	Definição da classe Circle .	449
9.13	Testando a classe Circle .	450
9.14	Definição da classe Cylinder .	451
9.15	Testando a classe Cylinder .	453
9.16	Superclasse abstract Employee .	458

9.17	Boss estende a classe <code>abstract Employee</code> .	459
9.18	CommissionWorker estende a classe <code>abstract Employee</code> .	460
9.19	PieceWorker estende a classe <code>abstract Employee</code> .	461
9.20	HourlyWorker estende a classe <code>abstract Employee</code> .	462
9.21	Testando a hierarquia de classes <code>Employee</code> usando uma superclasse <code>abstract</code> .	463
9.22	Superclasse abstrata <code>Shape</code> para a hierarquia <code>Point</code> , <code>Circle</code> e <code>Cylinder</code> .	466
9.23	Subclasse <code>Point</code> da classe abstrata <code>Shape</code> .	466
9.24	Subclasse <code>Circle</code> de <code>Point</code> – subclasse indireta da classe <code>abstract Shape</code> .	467
9.25	Subclasse <code>Cylinder</code> de <code>Circle</code> – subclasse indireta da classe abstrata <code>Shape</code> .	469
9.26	Hierarquia <code>Shape</code> , <code>Point</code> , <code>Circle</code> , <code>Cylinder</code> .	470
9.27	Hierarquia ponto, círculo, cilindro com uma interface <code>Shape</code> .	472
9.28	Implementação de <code>Point</code> com a interface <code>Shape</code> .	473
9.29	Subclasse <code>Circle</code> de <code>Point</code> – implementação indireta da interface <code>Shape</code> .	474
9.30	Subclasse <code>Cylinder</code> de <code>Circle</code> – implementação indireta da interface <code>Shape</code> .	475
9.31	Hierarquia <code>Shape</code> , <code>Point</code> , <code>Circle</code> , <code>Cylinder</code> .	476
9.32	Classe <code>Time</code> .	478
9.33	Demonstrando uma classe interna em um aplicativo com janela.	480
9.34	Demonstrando classes internas anônimas.	484
9.35	Atributos e operações das classes <code>FloorButton</code> e <code>ElevatorButton</code> .	489
9.36	Atributos e operações das classes <code>FloorDoor</code> e <code>ElevatorDoor</code> .	489
9.37	Diagrama de generalização da superclasse <code>Location</code> e das subclasses <code>Elevator</code> e <code>Floor</code> .	491
9.38	Diagrama de classes de nosso simulador (incorporando herança).	492
9.39	Diagrama de classes com atributos e operações (incorporando herança).	493
9.40	Os 18 padrões de projeto da gangue dos quatro discutidos em <i>Java Como Programar 4a edição</i> .	494
9.41	Padrões simultâneos e de arquitetura de projeto discutidos em <i>Java Como Programar 4a edição</i> .	495
9.42	Classe <code>Singleton</code> assegura que somente um objeto de sua classe é criado.	496
9.43	Classe <code>SingletonExample</code> tenta criar um objeto <code>Singleton</code> mais de uma vez.	496
9.44	Saída da classe <code>SingletonExample</code> mostra que o objeto <code>Singleton</code> só pode ser criado uma vez.	497
9.45	A hierarquia <code>MyShape</code> .	505
9.46	A hierarquia <code>MyShape</code> .	507

10 Strings e caracteres

10.1	Demonstrando os construtores da classe <code>String</code> .	510
10.2	Os métodos de manipulação de caracteres da classe <code>String</code> .	512
10.3	Demonstrando comparações de <code>Strings</code> .	514
10.4	Métodos <code>startsWith</code> e <code>endsWith</code> da classe <code>String</code> .	516
10.5	Método <code>hashCode</code> da classe <code>String</code> .	518
10.6	Os métodos de pesquisa da classe <code>String</code> .	519
10.7	Métodos <code>substring</code> da classe <code>String</code> .	521
10.8	Método <code>concat</code> de <code>String</code> .	522
10.9	Métodos diversos de <code>String</code> .	523
10.10	Métodos <code>valueOf</code> da classe <code>String</code> .	525
10.11	Método <code>intern</code> da classe <code>String</code> .	526
10.12	Construtores da classe <code>StringBuffer</code> .	529
10.13	Métodos <code>length</code> e <code>capacity</code> de <code>StringBuffer</code> .	529

10.14	Métodos de manipulação de caracteres da classe StringBuffer .	531
10.15	Métodos append da classe StringBuffer .	532
10.16	Métodos insert e delete da classe StringBuffer .	534
10.17	Métodos static da classe Character para testar caracteres e converter para maiúsculas/minúsculas.	536
10.18	Métodos de conversão static da classe Character .	538
10.19	Métodos não static da classe Character .	541
10.20	Separando <i>strings</i> em <i>tokens</i> com um objeto StringTokenizer .	542
10.21	Programa que distribui cartas.	544
10.22	A classe ElevatorModelEvent é a superclasse para todas as outras classes de eventos em nosso modelo.	549
10.23	Diagrama de classes que modela a generalização entre ElevatorModelEvent e suas subclasses.	550
10.24	Disparando ações dos eventos de subclasses de ElevatorModelEvent .	550
10.25	Diagrama de colaborações modificado para passageiros que entram no Elevator e saem dele no primeiro Floor .	551
10.26	A classe ElevatorMoveEvent , subclasse de ElevatorModelEvent , é enviada quando o Elevator chegou no Floor ou dele partiu.	553
10.27	Interface ElevatorMoveListener fornece os métodos necessários para escutar eventos de partida e chegada do Elevator .	553
10.28	Diagrama de classes de nosso simulador (incluindo o tratamento de eventos).	554
10.29	Contagem para o string "Whether 'tis nobler in the mind to suffer"	559
10.30	As letras do alfabeto segundo o código Morse internacional.	561

11 Imagens gráficas e Java2D

11.1	Algumas classes e interfaces utilizadas neste capítulo a partir dos recursos gráficos originais de Java e da API Java2D.	565
11.2	Sistema de coordenadas Java. As unidades são medidas em <i>pixels</i> .	565
11.3	Constantes static da classe Color e valores de RGB.	567
11.4	Métodos Color e métodos Graphics relacionados com cores.	567
11.5	Demonstrando como configurar e obter uma Color .	568
11.6	Demonstrando o diálogo JColorChooser .	570
11.7	As guias HSB e RGB do diálogo JColorChooser .	573
11.8	Métodos e constantes de Font e métodos de Graphics relacionados com fontes.	573
11.9	Usando o método setFont de Graphics para mudar Fonts .	575
11.10	Medidas de fontes.	576
11.11	Os métodos FontMetrics e Graphics para obter medidas de fontes.	577
11.12	Obtendo informações sobre métricas da fonte.	577
11.13	Métodos de Graphics que desenham linhas, retângulos e elipses.	579
11.14	Demonstrando o método drawLine da classe Graphics .	580
11.15	A largura e a altura do arco para retângulos arredondados.	581
11.16	Elipse delimitada por um retângulo.	582
11.17	Ângulos de arcos positivo e negativo.	582
11.18	Métodos de Graphics para desenhar arcos.	582
11.19	Demonstrando drawArc e fillArc .	583
11.20	Métodos de Graphics para desenhar polígonos e a classe de construtores Polygon .	584

11.21	Demonstrando <code>drawPolygon</code> e <code>fillPolygon</code> .	585
11.22	Demonstrando algumas formas de Java2D.	587
11.23	Demonstrando algumas formas de Java2D.	590
11.24	Diagrama de classes que modela a classe <code>Person</code> que realiza a interface <code>DoorListener</code> .	593
11.25	Diagrama elidido de classes que modela a classe <code>Person</code> que realiza a interface <code>DoorListener</code> .	593
11.26	A classe <code>Person</code> é gerada a partir da Fig. 11.24.	594
11.27	Diagrama de classes que modela as realizações no modelo do elevador.	595
11.28	Diagrama de classes para as interfaces <i>listener</i> .	595

12 Componentes da interface gráfica com o usuário: parte 1

12.1	Uma janela de exemplo do Netscape Navigator com componentes GUI.	605
12.2	Alguns componentes GUI básicos.	605
12.3	Superclasses comuns de muitos dos componentes Swing.	607
12.4	Demonstrando a classe <code>JLabel</code> .	608
12.5	Algumas classes de eventos do pacote <code>java.awt.event</code> .	610
12.6	Interfaces <i>listeners</i> de eventos do pacote <code>java.awt.event</code> .	611
12.7	Demonstrando <code>JTextFields</code> e <code>JPasswordFields</code> .	612
12.8	Registro de evento para <code>textField1 JTextField</code> .	616
12.9	A hierarquia de botões.	617
12.10	Demonstrando botões de comando e eventos de ação.	617
12.11	Programa que cria dois botões <code>JCheckBox</code> .	620
12.12	Criando e manipulando botões de opção.	622
12.13	Programa que usa uma <code>JComboBox</code> para selecionar um ícone.	625
12.14	Selecionando cores a partir de uma <code>JList</code> .	627
12.15	Usando uma <code>JList</code> de seleção múltipla.	629
12.16	Métodos das interfaces <code>MouseListener</code> e <code>MouseMotionListener</code> .	631
12.17	Demonstrando o tratamento de eventos do mouse.	632
12.18	As classes adaptadoras de eventos e as interfaces que elas implementam.	635
12.19	Programa que demonstra classes adaptadoras.	636
12.20	Distinguindo o clique do botão esquerdo, do meio e direito do mouse.	638
12.21	Métodos <code>InputEvent</code> que ajudam a distinguir entre cliques com o botão esquerdo, do meio e direito do mouse.	640
12.22	Demonstrando tratamento de eventos de teclas.	640
12.23	Gerenciadores de leiaute.	643
12.24	Programa que demonstra componentes em um <code>FlowLayout</code> .	644
12.25	Demonstrando componentes em <code>BorderLayout</code> .	646
12.26	Programa que demonstra componentes em um <code>GridLayout</code> .	649
12.27	Um <code>JPanel</code> com cinco <code>JButtons</code> em um <code>GridLayout</code> anexado à região <code>SOUTH</code> de um <code>BorderLayout</code> .	651
12.28	Diagrama de caso de uso para a simulação de elevador da perspectiva do usuário.	653
12.29	Diagrama de caso de uso para a simulação de elevador da perspectiva de uma pessoa.	654
12.30	Classe <code>ElevatorController</code> processa dados de entrada do usuário.	654
12.31	Interface <code>ElevatorConstants</code> fornece constantes com nome das <code>Locations</code> .	657
12.32	Diagrama de classes modificado mostrando a generalização da superclasse <code>Location</code> e das subclasses <code>Elevator</code> e <code>Floor</code> .	657

13 Componentes da interface gráfica com o usuário: parte 2

13.1	Copiando o texto selecionado de uma área de texto para outra.	669
13.2	Definindo uma área de desenho personalizada com uma subclasse de <code>JPanel</code> .	673
13.3	Desenhando em uma subclasse personalizada da classe <code>JPanel</code> .	674
13.4	Subclasse personalizada de <code>JPanel</code> que processa eventos do mouse.	676
13.5	Capturando eventos do mouse com um <code>JPanel</code> .	678
13.6	Um componente <code>JSlider</code> horizontal.	680
13.7	Subclasse personalizada de <code>JPanel</code> para desenhar círculos de um diâmetro especificado.	681
13.8	Usando um <code>JSlider</code> para determinar o diâmetro de um círculo.	682
13.9	Criando um aplicativo baseado em GUI a partir de um <i>applet</i> .	686
13.10	Usando <code>JMenus</code> e mnemônicos.	691
13.11	Usando um objeto <code>PopupMenu</code> .	698
13.12	Mudando a aparência e o comportamento de uma GUI baseada em Swing.	701
13.13	Criando uma interface com múltiplos documentos.	704
13.14	Gerenciadores de layout adicionais.	707
13.15	Demonstrando o gerenciador de layout <code>BoxLayout</code> .	708
13.16	Demonstrando o gerenciador de layout <code>CardLayout</code> .	711
13.17	Projetando uma GUI que utilizará <code>GridBagLayout</code> .	714
13.18	Variáveis de instância de <code>GridBagConstraints</code> .	714
13.19	<code>GridBagLayout</code> com <code>weightx</code> e <code>weighty</code> configurados como zero.	715
13.20	Demonstrando o gerenciador de layout <code>GridBagLayout</code> .	716
13.21	Demonstrando as constantes <code>RELATIVE</code> e <code>REMAINDER</code> de <code>GridBagConstraints</code> .	719
13.22	Diagrama de classes da simulação do elevador.	723
13.23	Diagrama de componentes para a simulação do elevador.	724
13.24	A classe <code>ElevatorSimulation</code> é o aplicativo para a simulação do elevador.	725
13.25	Hierarquia de herança para a classe <code>JPanel</code> .	728
13.26	Base para o padrão de projeto Observer.	730

14 Tratamento de exceções

14.1	Classe de exceção <code>DivideByZeroException</code> .	747
14.2	Um exemplo simples de tratamento de exceções com divisão por zero.	748
14.3	Erros do pacote <code>java.lang</code> .	753
14.4	Exceções do pacote <code>java.lang</code> .	754
14.5	Exceções do pacote <code>java.util</code> .	755
14.6	Exceções do pacote <code>java.io</code> .	755
14.7	Exceções do pacote <code>java.awt</code>	756
14.8	Exceções do pacote <code>java.net</code> .	756
14.9	Demonstração do mecanismo de tratamento de exceção <code>try-catch-finally</code>	757
14.10	Demonstração de um desempilhamento.	760
14.11	Usando <code>getMessage</code> e <code>printStackTrace</code> .	762

15 Multithreading

15.1	Ciclo de vida de uma <i>thread</i> .	772
15.2	O escalonamento por prioridade de <i>threads</i> em Java.	774
15.3	Múltiplas <i>threads</i> imprimindo a intervalos de tempo aleatórios.	775

42 ILUSTRAÇÕES

15.4	Classe ProduceInteger representa o produtor em um relacionamento produtor/consumidor.	779
15.5	Classe ConsumeInteger representa o consumidor em um relacionamento produtor/consumidor.	780
15.6	Classe HoldIntegerUnsynchronized mantém os dados compartilhados entre as <i>threads</i> produtora e consumidora.	781
15.7	<i>Threads</i> modificando um objeto compartilhado sem sincronização.	781
15.8	Classe ProduceInteger representa o produtor em um relacionamento produtor/consumidor.	783
15.9	Classe ConsumeInteger representa o consumidor em um relacionamento produtor/consumidor.	784
15.10	Classe HoldIntegerSynchronized monitora o acesso a um inteiro compartilhado.	785
15.11	<i>Threads</i> modificando um objeto compartilhado com sincronização.	786
15.12	UpdateThread usada pelo método invokeLater de SwingUtilities para assegurar que a GUI seja atualizada apropriadamente.	788
15.13	Classe ProduceInteger representa o produtor em um relacionamento produtor/consumidor.	789
15.14	Classe ConsumeInteger representa o consumidor em um relacionamento produtor/consumidor.	789
15.15	Classe HoldIntegerSynchronized monitora o acesso a um <i>array</i> de inteiros compartilhado.	791
15.16	<i>Threads</i> modificando um <i>array</i> de células compartilhado.	794
15.17	Demonstrando a interface Runnable , suspensendo <i>threads</i> e retomando <i>threads</i> .	797
15.18	Diagrama de colaborações modificado com as classes ativas para passageiros que entram no Elevator e saem dele.	803
15.19	Diagrama de seqüência para uma única Person trocando de andar no sistema.	806
15.20	Diagrama de classes final da simulação de elevador.	808
15.21	Diagrama de classes final com atributos e operações.	809

16 Arquivos e fluxos

16.1	A hierarquia de dados.	820
16.2	Visão de Java de um arquivo de <i>n bytes</i> .	821
16.3	Parte da hierarquia de classes do pacote java.io .	821
16.4	BankUI contém uma GUI reusável para diversos programas.	825
16.5	Classe AccountRecord mantém as informações sobre uma conta.	827
16.6	Cria um arquivo seqüencial.	829
16.5	Exemplo de dados para o programa da Fig. 16.4.	835
16.8	Lendo um arquivo seqüencial.	835
16.9	Programa de consulta de crédito.	840
16.10	A visão de Java de um arquivo de acesso aleatório.	847
16.11	Classe RandomAccessAccountRecord usada nos programas com arquivo de acesso aleatório.	847
16.12	Criando um arquivo de acesso aleatório seqüencialmente.	849
16.13	Gravando dados aleatoriamente em um arquivo de acesso aleatório.	851
16.14	Lendo um arquivo de acesso aleatório seqüencialmente.	856
16.15	A janela inicial do Transaction Processor .	861
16.16	Carregando um registro para a <i>frame</i> interna Update Record .	861

16.17	Lendo uma transação na <i>frame</i> interna Update Record .	862
16.18	A <i>frame</i> interna New Record .	862
16.19	A <i>frame</i> interna Delete Record .	862
16.20	Programa de processamento de transações.	862
16.21	Alguns métodos de File comumente utilizados.	876
16.22	Demonstrando a classe File .	877
16.23	Exemplo de dados para o arquivo-mestre.	887
16.24	Exemplo de dados para o arquivo de transações.	888
16.25	Registros de transações adicionais.	888
16.26	Dados para o Exercício 16.10.	888
16.17	Dígitos e letras do teclado do telefone.	889
17	Redes	
17.1	Documento HTML para carregar o <i>applet</i> SiteSelector .	893
17.2	Carregando um documento a partir de um URL para um navegador.	894
17.3	Lendo um arquivo ao se abrir uma conexão através de um URL.	897
17.4	A parte do servidor de uma conexão cliente/servidor com soquete de fluxo.	902
17.5	Demonstrando a parte do cliente de uma conexão com soquete de fluxo entre um cliente e um servidor.	907
17.6	Demonstrando o lado do servidor da computação cliente/servidor sem conexão com datagramas.	912
17.7	Demonstrando o lado do cliente da computação cliente/servidor sem conexão com datagramas.	915
17.8	O lado do servidor do programa cliente/servidor do jogo-da-velha.	919
17.9	O lado do cliente do programa cliente/servidor do jogo-da-velha.	924
17.10	Exemplos de saídas do programa cliente/servidor do jogo-da-velha.	930
17.11	Aplicativo DeitelMessengerServer para administrar uma sala de bate-papo.	932
17.12	SocketMessengerConstants declara constantes para uso ao longo de todo o DeitelMessengerServer e o DeitelMessenger	934
17.13	Interface MessageListener que define o método messageReceived para receber novas mensagens de bate-papo.	934
17.14	ReceivingThread para esperar por novas mensagens de clientes do DeitelMessengerServer em Threads separadas.	935
17.15	MulticastSendingThread para entregar mensagens enviadas para um grupo de <i>multicast</i> através de DatagramPackets .	938
17.16	Interface MessageManager que define métodos para se comunicar com um DeitelMessengerServer .	940
17.17	Implementação da interface MessageManager em SocketMessageManager para comunicação através de Sockets e DatagramPackets de <i>multicast</i> .	941
17.18	SendingThread para entregar mensagens enviadas para o DeitelMessengerServer .	943
17.19	PacketReceivingThread para esperar por novas mensagens de <i>multicast</i> do DeitelMessengerServer em uma Thread separada.	944
17.20	Subclasse ClientGUI de JFrame que apresenta uma GUI para visualizar e enviar mensagens de bate-papo.	948
17.21	Aplicativo DeitelMessenger para participar de uma sessão de bate-papo do DeitelMessengerServer .	954
17.22	Arquitetura <i>Model-View-Controller</i> .	958
17.23	Modelo de aplicativo em três camadas.	959

18	Multimídia: imagens, animação, áudio e vídeo	
18.1	Carregando e exibindo uma imagem em um <i>applet</i> .	968
18.2	Animando uma série de imagens.	970
18.3	Subclasse <code>LogoAnimator2</code> de <code>LogoAnimator</code> (Fig. 18.2) adiciona um construtor para personalizar a quantidade de imagens, o retardo na animação e o nome básico das imagens.	974
18.4	Personalizando um <i>applet</i> animado através da marca de HTML <code>param</code> .	975
18.5	Demonstrando um mapa de imagens.	977
18.6	Carregando e reproduzindo um <code>AudioClip</code> .	980
19	Estruturas de dados	
19.1	Dois objetos de classe auto-referencial encadeados.	990
19.2	Uma representação gráfica de uma lista encadeada.	992
19.3	Definições da classe <code>ListNode</code> e da classe <code>List</code> .	992
19.4	Definição da classe <code>EmptyListException</code> .	995
19.5	Manipulando uma lista encadeada.	996
19.6	A operação <code>insertAtFront</code> .	998
19.7	Uma representação gráfica da operação <code>insertAtBack</code> .	998
19.8	Uma representação gráfica da operação <code>removeFromFront</code> .	999
19.9	Uma representação gráfica da operação <code>removeFromBack</code> .	1000
19.10	Classe <code>StackInheritance</code> estende a classe <code>List</code> .	1001
19.11	Um programa simples com pilha.	1002
19.12	Uma classe de pilha comum usando composição.	1004
19.13	Classe <code>QueueInheritance</code> que estende a classe <code>List</code> .	1005
19.14	Processando uma fila.	1006
19.15	Uma representação gráfica de uma árvore binária.	1008
19.16	Uma árvore de pesquisa binária que contém 12 valores.	1008
19.17	Definições de <code>TreeNode</code> e <code>Tree</code> para uma árvore de pesquisa binária.	1008
19.18	Criando e percorrendo uma árvore de pesquisa binária.	1011
19.19	Uma árvore de pesquisa binária.	1012
19.20	Uma árvore de pesquisa binária de 15 nós.	1017
19.21	Exemplo de saída do método recursivo <code>outputTree</code> .	1022
19.22	Comandos de Simple.	1022
19.23	Programa em Simple que determina a soma de dois inteiros.	1023
19.24	Programa em Simple que determina o maior de dois inteiros.	1024
19.25	Calcula os quadrados de vários inteiros.	1024
19.26	Gravando, compilando e executando um programa da linguagem Simple.	1025
19.27	Instruções de SML produzidas depois da primeira passagem do compilador.	1026
19.28	Tabela de símbolos para o programa da Fig. 19.27.	1027
19.29	Código não-otimizado do programa da Fig. 19.25.	1031
19.30	Código otimizado para o programa da Fig. 19.27.	1031
20	Pacote de utilitários Java e manipulação de bits	
20.1	Demonstrando a classe <code>Vector</code> do pacote <code>java.util</code> .	1036
20.2	Demonstrando a classe <code>Stack</code> do pacote <code>java.util</code> .	1042
20.3	Demonstrando a classe <code>Hashtable</code> .	1047
20.4	Demonstrando a classe <code>Properties</code> .	1053

20.5	Operadores sobre <i>bits</i> .	1059
20.6	Exibindo a representação dos <i>bits</i> de um inteiro.	1060
20.7	Resultados da combinação de dois <i>bits</i> com o operador E sobre <i>bits</i> (&).	1062
20.8	Demonstrando os operadores E sobre <i>bits</i> , OU inclusivo sobre <i>bits</i> , OU exclusivo sobre <i>bits</i> e o complemento sobre <i>bits</i> .	1062
20.9	Resultados da combinação de dois <i>bits</i> com o operador OU inclusivo sobre <i>bits</i> ().	1066
20.10	Resultados da combinação de dois <i>bits</i> com o operador OU exclusivo sobre <i>bits</i> (^).	1067
20.11	Demonstrando os operadores de deslocamento sobre <i>bits</i> .	1067
20.12	Os operadores de atribuição sobre <i>bits</i> .	1071
20.13	Demonstrando o Crivo de Eratóstenes com um BitSet .	1072

21 Coleções

21.1	Utilizando métodos da classe Arrays .	1083
21.2	Utilizando o método static asList .	1086
21.3	Utilizando uma ArrayList para demonstrar a interface Collection .	1088
21.4	Utilizando Lists e ListIterators .	1089
21.5	Utilizando o método toArray .	1091
21.6	Utilizando o algoritmo sort .	1093
21.7	Utilizando um objeto Comparator em sort .	1094
21.8	Exemplo de embaralhamento e distribuição de cartas.	1095
21.9	Utilizando os algoritmos reverse , fill , copy , max e min .	1097
21.10	Utilizando o algoritmo binarySearch .	1099
21.11	Utilizando um HashSet para remover duplicatas.	1101
21.12	Utilizando SortedSets e TreeSets .	1102
21.13	Utilizando HashMaps e Maps .	1103
21.14	Métodos empacotadores de sincronização.	1105
21.15	Métodos empacotadores não-modificáveis.	1105

22 Java Media Framework e Java Sound (no CD)

22.1	Reproduzindo mídia com a interface Player .	1114
22.2	Formatando e salvando mídia a partir de dispositivos de captura.	1124
22.3	Servindo mídia de <i>streaming</i> com gerenciadores de sessão RTP.	1136
22.4	Aplicativo para testar a classe RTPServer da Fig. 22.3.	1143
22.5	ClipPlayer reproduz um arquivo de áudio.	1148
22.6	ClipPlayerTest permite especificar o nome e o endereço do áudio a ser reproduzido com ClipPlayer .	1152
22.7	MidiData carrega arquivos MIDI para reprodução.	1155
22.8	MidiRecord permite a um programa gravar uma seqüência MIDI.	1160
22.9	MidiSynthesizer pode gerar notas e enviá-las para um outro dispositivo MIDI.	1163
22.10	MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo.	1167
22.11	Diagrama de classes da visão da simulação do elevador.	1183
22.12	A classe ImagePanel representa e exibe um objeto imóvel do modelo.	1183
22.13	A classe MovingPanel representa e exibe um objeto do modelo em movimento.	1185
22.14	A classe AnimatedPanel representa e exibe um objeto animado do modelo.	1187
22.15	Relacionamento entre o array imageIcons e a List frameSequences .	1191
22.16	A classe SoundEffects devolve objetos AudioClip .	1191

22.17 A classe `ElevatorMusic` toca música quando uma `Person` anda no `Elevator`. 1192

C Tabela de precedência de operadores

C.1 Tabela de precedência de operadores. 1213

D Conjunto de caracteres ASCII

D.1 Conjunto de caracteres ASCII. 1215

E Sistemas de numeração (no CD)

E.1	Os dígitos dos sistemas de numeração binária, octal decimal e hexadecimal.	1218
E.2	Comparação dos sistemas de numeração binária, octal, decimal e hexadecimal.	1218
E.3	Valores posicionais no sistema de numeração decimal.	1218
E.4	Valores posicionais no sistema de numeração binária.	1219
E.5	Valores posicionais no sistema de numeração octal.	1219
E.6	Valores posicionais no sistema de numeração hexadecimal.	1219
E.7	Equivalentes decimais, binários, octais e hexadecimais.	1220
E.8	Convertendo um número binário em decimal.	1221
E.9	Convertendo um número octal em decimal.	1221
E.10	Convertendo um número hexadecimal em decimal.	1222

F Criando documentação em HTML com javadoc (no CD)

F.1	A documentação da API Java.	1230
F.2	Um arquivo de código-fonte em Java contendo comentários de documentação.	1231
F.3	A documentação em HTML para a classe <code>Time3</code> .	1235
F.4	A nota Parameters: gerada por <code>javadoc</code> .	1236
F.5	A documentação em HTML para o método <code>setTime</code> .	1236
F.6	A documentação em HTML para <code>getHour</code> .	1237
F.7	Marcas comuns de <code>javadoc</code> .	1237
F.8	Utilizando a ferramenta <code>javadoc</code> .	1238
F.9	<code>index.html</code> da classe <code>Time3</code> .	1239
F.10	A página <code>Tree</code> .	1239
F.11	A página <code>index-all.html</code> de <code>Time3</code> .	1240
F.12	A página <code>helpdoc.html</code> de <code>Time3</code> .	1240

G Eventos e interfaces *listener* do elevador (no CD)

G.1	Superclasse <code>ElevatorModelEvent</code> para eventos no modelo da simulação de elevador.	1242
G.2	Subclasse <code>BellEvent</code> de <code>ElevatorModelEvent</code> indicando que a <code>Bell</code> tocou.	1243
G.3	Subclasse <code>ButtonEvent</code> de <code>ElevatorModelEvent</code> indicando que um <code>Button</code> mudou de estado.	1244
G.4	Subclasse <code>DoorEvent</code> de <code>ElevatorModelEvent</code> indicando que uma <code>Door</code> mudou de estado.	1244
G.5	Subclasse <code>ElevatorMoveEvent</code> de <code>ElevatorModelEvent</code> indicando em que <code>Floor</code> o <code>Elevator</code> chegou ou do qual ele partiu.	1244

G.6	Subclasse LightEvent de ElevatorModelEvent indicando qual o Floor cuja Light mudou de estado.	1245
G.7	Subclasse PersonMoveEvent de ElevatorModelEvent indicando que uma Person se moveu.	1245
G.8	Método da interface BellListener para quando a Bell tocou.	1246
G.9	Métodos da interface ButtonListener para quando o Button foi pressionado ou desligado.	1246
G.10	Métodos da interface DoorListener para quando a Door abriu ou fechou.	1246
G.11	Métodos da interface ElevatorMoveListener para quando o Elevator partiu de um Floor ou chegou em um.	1247
G.12	Método da interface LightListener para quando a Light ligou ou desligou.	1247
G.13	Métodos da interface PersonMoveListener para quando a Person se moveu.	1247
G.14	A interface ElevatorModelListener permite que o modelo envie todos os eventos para a visão.	1248
G.15	Diagrama de componentes para o pacote event .	1249

H Modelo do elevador (no CD)

H.1	Classe ElevatorModel representa o modelo em nossa simulação de elevador.	1251
H.2	Diagramas de classes mostrando as realizações no modelo do elevador.	1257
H.3	Diagramas de classes mostrando as realizações no modelo do elevador.	1258
H.4	Classes e interfaces <i>listener</i> implementadas da Fig. H.2.	1258
H.5	Superclasse abstrata Location que representa um lugar na simulação.	1259
H.6	A classe Floor – uma subclasse de Location – representa um Floor através do qual a Person caminha para o Elevator .	1259
H.7	A classe Door , que representa uma Door no modelo, informa aos ouvintes quando uma Door abriu ou fechou.	1261
H.8	A classe Button , que representa um Button no modelo, informa aos ouvintes quando o Button foi pressionado ou desligado.	1264
H.9	A classe ElevatorShaft , que representa o ElevatorShaft , que envia eventos do Elevator para o ElevatorModel .	1265
H.10	A classe Light representa uma Light no Floor no modelo.	1271
H.11	A classe Bell representa a Bell no modelo.	1273
H.12	A classe Elevator representa o Elevator se movendo entre dois Floors , operando assincronamente com outros objetos.	1275
H.13	A classe Person representa a Person que anda no Elevator . A Person opera assincronamente com outros objetos.	1282
H.14	Diagrama de componentes para o pacote model .	1289

I Visão do elevador (no CD)

I.1	ElevatorView exibe o modelo de simulação do elevador.	1290
I.2	Objetos na ElevatorView que representam objetos no modelo.	1306
I.3	Objetos na ElevatorView não representados no modelo.	1306
I.4	Diagrama de objetos para a ElevatorView após a inicialização.	1309
I.5	Diagrama de componentes para o pacote view .	1312

J Oportunidades de Carreira Profissional (no CD)

J.1	A homepage de Monster.com (cortesia de Monster.com).	1316
J.2	A pesquisa de trabalho do FlipDog.com (cortesia de FlipDog.com).	1317
J.3	Lista de critérios das pessoas que procuram emprego.	1319
J.4	O serviço Net-Interview™ da Advantage Hiring, Inc. (cortesia de Advantage Hiring, Inc.).	1321
J.5	Os serviços on-line para a carreira profissional do Cruel World. (cortesia de Cruel World).	1323
J.6	Exemplo de solicitação de proposta (RFP) do eLance.com (cortesia de eLance, Inc.).	1325

K Unicode® (no CD)

K.1	Correlação entre as três formas de codificação.	1339
K.2	Vários hieróglifos do caractere A.	1339
K.3	Programa Java que usa a codificação Unicode.	1340
K.4	Alguns intervalos de caracteres.	1343

Introdução aos computadores, à Internet e à Web

Objetivos

- Entender conceitos básicos da ciência da computação.
- Familiarizar-se com diferentes tipos de linguagens de programação.
- Apresentar o ambiente de desenvolvimento de programas Java.
- Entender o papel que Java desempenha no desenvolvimento de aplicativos cliente/servidor distribuídos para a Internet e a World Wide Web.
- Apresentar o projeto orientado a objetos com a UML e padrões de projeto.
- Visualizar os capítulos restantes do livro.

*Nossa vida é desperdiçada pelos detalhes ... Simplifique,
simplifique.*

Henry Thoreau

Pensamentos elevados devem ter uma linguagem elevada.

Aristófanes

O principal mérito da língua é a clareza.

Galen

*My object all sublime.
I shall achieve in time.*

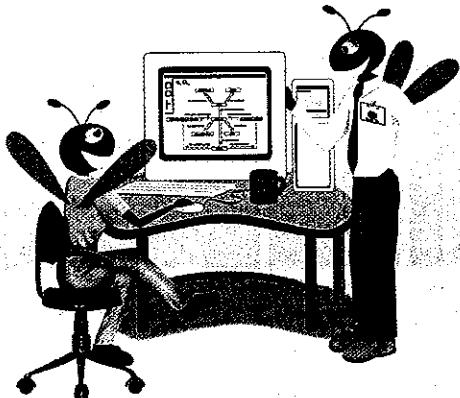
W. S. Gilbert

*Ele tinha um talento maravilhoso para empacotar bem os
pensamentos e torná-los portáteis.*

Thomas Babington Macaulay

*Meu Deus, entre os dois, acho que o mais difícil de entender
é o intérprete!*

Richard Brinsley Sheridan



Sumário do capítulo

- 1.1 Introdução
- 1.2 O que é um computador?
- 1.3 A organização de um computador
- 1.4 Evolução dos sistemas operacionais
- 1.5 Computação pessoal, distribuída e computação cliente/servidor
- 1.6 Linguagens de máquina, linguagens assembly e linguagens de alto nível
- 1.7 A história de C++
- 1.8 A história de Java
- 1.9 Bibliotecas de classes Java
- 1.10 Outras linguagens de alto nível
- 1.11 Programação estruturada
- 1.12 A Internet e a World Wide Web
- 1.13 Princípios básicos de um ambiente Java típico
- 1.14 Notas gerais sobre Java e este livro
- 1.15 Pensando em objetos: introdução à tecnologia de objetos e à Unified Modeling Language
- 1.16 Descobrindo padrões de projeto: introdução
- 1.17 Um passeio pelo livro
- 1.18 (Opcional) Um passeio pelo estudo de caso sobre projeto orientado a objetos com a UML
- 1.19 (Opcional) Um passeio pelas seções “Descobrindo padrões de projeto”

Resumo e Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão • Exercícios

1.1 Introdução

Bem-vindo a Java! Trabalhamos muito para criar o que esperamos ser uma experiência de aprendizagem desafiadora, informativa e divertida para você. Java é uma poderosa linguagem de programação de computador que é divertida de usar para os iniciantes e adequada para os programadores experientes construir poderosos sistemas de informações. *Java Como Programar: Quarta Edição* é projetado para ser uma ferramenta de aprendizagem eficiente para cada um desses públicos.

Como o livro pode servir para ambos os grupos? A resposta é que o núcleo comum do livro enfatiza a obtenção de *clareza* no programa por meio das técnicas já experimentadas de *programação estruturada* e *programação orientada a objetos*. Os que não são programadores aprenderão programação da maneira certa desde o início. Tentamos escrever de forma direta e clara. O livro é ricamente ilustrado. Talvez mais importante que tudo seja o fato de este livro apresentar centenas de programas Java prontos e funcionando e mostrar as saídas produzidas quando esses programas são executados em um computador. Ensinamos todos os recursos de Java no contexto de programas de Java completos, prontos e funcionando. Chamamos isso de *abordagem de código ativo (live-code™)*. Esses exemplos estão disponíveis a partir de três fontes – estão no CD que acompanha este livro, podem ser baixados de nosso site www.deitel.com e estão disponíveis em nosso produto em CD interativo, *Java 2 Multimedia Cyber Classroom: Fourth Edition*. As características do *Cyber Classroom* e as informações sobre como encomendá-lo estão no final deste livro. O *Cyber Classroom* também contém respostas de aproximadamente metade dos exercícios resolvidos deste livro, incluindo respostas curtas, programas pequenos e muitos projetos completos. Se você comprou *The Complete Java 2 Training Course: Fourth Edition*, você já possui o *Cyber Classroom*.

Os primeiros capítulos apresentam os princípios básicos dos computadores, da programação de computadores e da linguagem de programação Java. Os iniciantes que fizeram nossos cursos costumam dizer que o material nesses capítulos apresenta uma base sólida para o tratamento mais aprofundado de Java nos capítulos posteriores. Os programadores mais experientes tendem a ler os primeiros capítulos rapidamente e acham que o tratamento de Java nos últimos capítulos é rigoroso e desafiador.

Muitos programadores experientes nos disseram que gostam da nossa abordagem de programação estruturada. Com freqüência, eles já programam em linguagens estruturadas como C ou Pascal, mas nunca foram apresentados formalmente à programação estruturada, de modo que conseguem escrever o melhor código possível nessas linguagens. À medida que esses programadores revisarem a programação estruturada nos capítulos “Estruturas de controle: parte 1” e “Estruturas de controle: parte 2”, eles serão capazes de aprimorar seu estilo de programar em C e em Pascal. Então, se você for um iniciante ou um programador experiente, aqui há muito para informá-lo, entrevê-lo e desafiá-lo.

A maioria das pessoas conhece as tarefas emocionantes que os computadores executam. Utilizando este livro, você aprenderá como comandar os computadores para que executem essas tarefas. É o *software* (isto é, as instruções que você escreve para comandar os computadores e fazer com que executem *ações* e tomem *decisões*) que controla os computadores (freqüentemente chamados de *hardware*), e Java é atualmente uma das linguagens mais populares de desenvolvimento de *software*. Java foi desenvolvida pela Sun Microsystems e uma implementação dela está disponível gratuitamente na Internet a partir do *site* da Sun.

java.sun.com/j2se

Este livro baseia-se na *Java 2 Platform, Standard Edition*, que descreve a linguagem Java, bibliotecas e ferramentas. Outros fornecedores podem implementar *kits de desenvolvimento Java* baseados na *Java 2 Platform*. A Sun fornece uma implementação do *Java 2 Platform, Standard Edition* chamada *Java 2 Software Development Kit, Standard Edition (J2SDK)*, que inclui o conjunto mínimo de ferramentas necessárias para escrever *softwares* em Java. Na época da publicação deste livro, a versão mais recente era o J2SDK 1.3.1. Você pode baixar atualizações gratuitas da J2SDK do *site* da Sun, java.sun.com/j2se.

O uso dos computadores está aumentando em quase todos os campos de atividade. Em uma era de custos de crescimento constante, os custos de computação têm diminuído drasticamente por causa do rápido desenvolvimento em tecnologia de *hardware* e *software*. Os computadores que ocupavam grandes salas e custavam milhões de dólares há duas décadas agora podem ser gravados na superfície de chips de silício menores que uma unha, ao custo de apenas alguns dólares. Ironicamente, o silício é um dos materiais mais abundantes na Terra – é um ingrediente da areia comum. A tecnologia do chip de silício tornou a computação tão econômica que aproximadamente 200 milhões de computadores de uso geral estão em utilização em todo o mundo, auxiliando pessoas no comércio, na indústria, no governo e em suas vidas pessoais. O número de computadores em todo o mundo pode facilmente dobrar nos próximos anos.

Este livro o desafiará por várias razões. Por muitos anos, os alunos aprenderam C ou Pascal como sua primeira linguagem de programação. Eles provavelmente aprenderam a metodologia de programação conhecida como *programação estruturada*. Você aprenderá tanto a programação estruturada como a mais nova e estimulante metodologia, a *programação orientada a objetos*. Por que ensinamos ambas? Acreditamos que a orientação a objetos é a principal metodologia de programação do futuro. Você construirá e trabalhará com muitos *objetos* nesse curso. Entretanto, você descobrirá que a estrutura interna desses objetos é construída com as técnicas de programação estruturada. Além disso, a lógica de manipular objetos é ocasionalmente melhor expressa através da programação estruturada.

Outra razão de apresentarmos ambas as metodologias é a contínua migração dos sistemas baseados em C (construídos principalmente com técnicas de programação estruturada) para sistemas baseados em C++ e em Java (construídos principalmente com técnicas de programação orientada a objetos). Há uma quantidade enorme do chamado “código legado de C” implantado, porque C foi intensamente utilizada por mais de três décadas. Uma vez que as pessoas aprendem C++ ou Java, elas acham que essas linguagens são mais poderosas que C. Estas pessoas freqüentemente decidem mudar seus projetos de programação para C++ ou Java. Elas começam convertendo seus sistemas legados e passam a empregar os recursos de programação orientada a objetos de C++ ou de Java para aproveitar integralmente os benefícios dessas linguagens. Freqüentemente a escolha entre C++ e Java é feita com base na simplicidade de Java comparada com C++.

Java se tornou a linguagem preferida para implementar aplicativos baseados em Intranet e Internet e *softwares* para dispositivos que se comunicam através de uma rede. Não se surpreenda quando seu novo estéreo e outros dispositivos em sua casa estiverem conectados em rede utilizando tecnologia Java! Além disso, não se surprenda quan-

do seus dispositivos sem-fio, como telefones celulares, *pgers* e assistentes digitais pessoais (PDAs) se comunicam através da assim chamada Internet sem-fio, usando o tipo de protocolos para redes baseados em Java que você aprenderá neste livro e no seu companheiro *Advanced Java 2 Platform How to Program*.

Java é particularmente atraente como primeira linguagem de programação. No evento JavaOne™, em junho de 2001, foi anunciado que Java agora é parte obrigatória do currículo de linguagens de programação em 56% das faculdades e universidades dos EUA. Ademais, 87% das faculdades e universidades americanas oferecem cursos sobre Java. Java também é atraente para as escolas de segundo grau. Em 2003, o College Board vai estabelecer Java como padrão para cursos de ciência da computação de colocação antecipada.

Java evoluiu rapidamente para a arena de aplicativos de grande porte. Não é mais uma linguagem usada simplesmente para fazer as páginas da World Wide Web “ganharem vida”. Java se tornou a linguagem preferida para atender às necessidades de programação de uma organização.

Durante muitos anos, linguagens como C e C++ foram atraentes para as universidades por causa de sua portabilidade. Cursos introdutórios podiam ser oferecidos nessas linguagens em qualquer combinação de sistema operacional/*hardware* desde que um compilador C/C++ estivesse disponível. Entretanto, o mundo da programação tornou-se mais complexo e mais exigente. Os usuários atuais querem aplicativos com interfaces gráficas com o usuário (GUI – *graphical user interfaces*). Querem aplicativos que utilizem as capacidades de multimídia como gráficos, imagens, animação, áudio e vídeo. Querem aplicativos que possam rodar na Internet e na World Wide Web e que se comuniquem com outros aplicativos. Os usuários querem aplicativos que tirem proveito dos aprimoramentos na flexibilidade e no desempenho proporcionados pelo *multithreading ou multiescalonamento* (o *multithreading* permite que os programadores especifiquem que várias atividades ocorram paralelamente). Querem aplicativos com processamento de arquivos mais rico do que o oferecido por C ou C++. Querem aplicativos que não sejam limitados a computadores *desktop* ou mesmo a alguma rede local de computadores, mas que também possam integrar componentes de Internet e bancos de dados remotos. Os usuários querem aplicativos que possam ser escritos rápida e corretamente de maneira a tirar proveito de componentes de *softwares* preexistentes. Querem acesso fácil a um crescente universo de componentes de *softwares* reutilizáveis. Os programadores querem todos esses benefícios de uma maneira verdadeiramente portável, de modo que aplicativos rodem sem modificações sobre uma variedade de *plataformas* (isto é, diferentes tipos de computadores executando diferentes sistemas operacionais). Java oferece todos esses benefícios à comunidade de programação.

Outra razão de Java ser atraente para os cursos universitários é que ela é completamente orientada a objetos. Uma razão que tem aumentado a utilização de C++ tão rapidamente é que ela estende a programação em C para a arena de orientação a objetos. Para a enorme comunidade de programadores C, esta foi uma vantagem poderosa. C++ inclui ANSI/ISO C e também oferece a capacidade de fazer programação orientada a objetos (ANSI é o American National Standards Institute, e ISO é o International Standards Organization). Uma quantidade enorme de código C foi escrita nas empresas nas várias últimas décadas. C++ é um superconjunto de C, de modo que muitas organizações acham que ele é um próximo passo ideal. Os programadores podem pegar seu código C, compilá-lo em um compilador C++, freqüentemente com alterações nominais, e continuar escrevendo código semelhante a C enquanto domina o paradigma de objetos. Então os programadores podem gradualmente migrar partes do código legado de C para C++ conforme o tempo permitir. Novos sistemas podem ser inteiramente escritos em C++ orientado a objetos. Tais estratégias têm atraído muitas organizações. A desvantagem é que, mesmo depois de adotar essa estratégia, as empresas tendem a continuar produzindo código semelhante a C durante muitos anos. Isso, naturalmente, significa que eles não percebem rapidamente os benefícios da programação orientada a objetos e podem produzir programas que são confusos e difíceis de manter como resultado de seu projeto híbrido. Muitas organizações prefeririam poder mergulhar 100% no desenvolvimento orientado a objetos, mas a realidade de montanhas de código legado e a tentação de adotar uma abordagem de programação em C freqüentemente impede que isso ocorra.

Java é uma linguagem completamente orientada a objetos com forte suporte para técnicas adequadas de engenharia de *software*. É difícil escrever programas semelhantes a C e chamadas procedurais em Java. Você deve criar e manipular objetos. O processamento de erro é embutido na linguagem. Muitos dos complexos detalhes da programação em C e C++ que impedem que os programadores tenham uma “visão geral” não são incluídos em Java. Para as universidades, esses recursos são poderosamente atraentes. Os alunos aprenderão programação orientada a objetos desde o início. Eles simplesmente pensarão de uma maneira orientada a objetos.

Aqui, também, há uma relação de troca entre aquilo que se ganha e aquilo que se perde. As organizações que se voltaram para Java a fim de desenvolver novos aplicativos não querem converter todo seu código legado para Java. Dessa forma, Java permite o chamado *código nativo*. Significa que o código existente em C e C++ pode ser in-

tegrado com o código Java. Embora possa parecer um pouco inconveniente (e certamente pode sê-lo), isso apresenta uma solução pragmática para um problema com que a maioria das organizações se defronta.

O fato de Java poder ser gratuitamente baixado do site da Sun, java.sun.com/j2se, é atraente para as universidades que se defrontam com orçamentos curtos e ciclos de planejamento de orçamento longos. Igualmente, à medida que os *bugs* forem corrigidos e novas versões de Java forem disponibilizadas, estas se tornarão imediatamente disponíveis pela Internet, permitindo que as universidades mantenham seu *software* em Java atualizado.

Java pode ser utilizada em um curso destinado àqueles que desejam aprender sua primeira linguagem de programação – seria este o público-alvo deste livro? Achamos que sim. Antes de escrever este livro, os instrutores da Deitel & Associates, Inc. ministraram centenas de cursos de Java para milhares de pessoas em todos os níveis de conhecimento, incluindo muitos cursos para não-programadores. Descobrimos que os não-programadores tornam-se produtivos mais rapidamente com Java do que com C ou C++. Os não-programadores estão ansiosos para experimentar recursos poderosos de Java como imagens, interfaces gráficas com o usuário, multimídia, animação, *multithreading*, rede e recursos semelhantes – e são bem-sucedidos na construção de programas Java substanciais mesmo em seus cursos iniciais.

Durante muitos anos, a linguagem de programação Pascal era o veículo preferido para utilização em cursos de programação introdutórios e intermediários. Muitas pessoas nos disseram que achavam C uma linguagem muito difícil para esses cursos. Em 1992, publicamos a primeira edição de *C How to Program*, para incentivar as universidades a experimentar C em vez de Pascal nesses cursos. Utilizamos a mesma abordagem pedagógica que vínhamos utilizando em nossos cursos universitários havia 12 anos, mas embalávamos os conceitos em C em vez de em Pascal. Descobrimos que os alunos eram capazes de tratar C no mesmo nível que Pascal. Porém, havia uma diferença notável – os alunos gostavam do fato de estar aprendendo uma linguagem (C) que provavelmente seria valiosa para eles na indústria. Nossos clientes industriais apreciavam a disponibilidade de graduados com conhecimentos em C que poderiam imediatamente trabalhar em projetos substanciais em vez de precisar primeiro cursar programas de treinamento caros e demorados.

A primeira edição de *C How to Program* incluiu uma introdução de 60 páginas a C++ e à programação orientada a objetos. Vimos o C++ crescendo e se tornando cada vez mais poderoso, porém sentímos que passariam ainda alguns anos antes de as universidades estarem prontas para ensinar C++ e a programação orientada a objetos (OOP – *object-oriented programming*) em cursos introdutórios.

Durante 1993, presenciamos um surto no interesse por C++ e OOP entre nossos clientes da indústria. Mas ainda não achávamos que as universidades estivessem prontas para mudar para C++ e a OOP em massa. Então publicamos, em janeiro de 1994, a segunda edição de *C How to Program* com uma seção de 300 páginas sobre C++ e OOP. Em maio de 1994, publicamos a primeira edição de *C++ How to Program*, um livro de 950 páginas baseado na ideia de que C++ e OOP estavam agora prontos para o horário nobre em cursos universitários introdutórios para muitas faculdades que queriam estar na liderança do ensino de linguagens de programação.

Em 1995, estávamos seguindo cuidadosamente a introdução de Java. Em novembro de 1995, assistimos a uma conferência de Internet em Boston. Um representante da Sun Microsystems fez uma apresentação sobre Java que preencheu um dos grandes salões de baile no Hynes Convention Center. Enquanto a apresentação prosseguia, tornou-se claro para nós que Java iria desempenhar um papel significativo no desenvolvimento de páginas da Web interativas, com multimídia. Vimos imediatamente um potencial muito maior para a linguagem. Vimos Java como a linguagem adequada para as universidades ensinarem aos alunos de linguagem de programação do primeiro ano nesse mundo moderno de imagens gráficas, imagens, animação, áudio, vídeo, banco de dados, rede e computação *multithreading* e colaborativa. Na época, estávamos ocupados escrevendo a segunda edição de *C++ How to Program*. Discutimos com nossa editora, Prentice Hall, nossa visão de que Java causaria um forte impacto no currículo universitário. Todos concordamos em atrasar um pouco a segunda edição de *C++ How to Program* de modo que pudéssemos colocar a primeira edição de *Java How to Program* (baseada em Java 1.0.2) no mercado em tempo para os cursos do segundo semestre de 1996.

À medida que Java rapidamente evoluía para a versão 1.1, escrevemos em 1997 *Java How to Program: Second Edition*, menos de um ano depois da primeira edição chegar às livrarias. Centenas de universidades e programas de treinamento corporativo em todo o mundo utilizaram a segunda edição. Para acompanhar o ritmo das melhorias de Java, publicamos *Java How to Program: Third Edition* em 1999. A terceira edição foi uma revisão significativa, para atualizar o livro para a *Java 2 Platform*.

Java continua a evoluir rapidamente, de modo que escrevemos esta quarta edição de *Java How to Program* – nosso primeiro livro a chegar à quarta edição – apenas cinco anos depois da primeira edição ter sido publicada. Es-

ta edição se baseia na *Java 2 Platform, Standard Edition (J2SE)*. Java cresceu tão rapidamente durante os últimos anos que ela agora tem duas outras edições. A *Java 2 Platform, Enterprise Edition (J2EE)*, é voltada para o desenvolvimento de aplicações de grande porte, distribuídas em redes, e aplicações baseadas na Web. A *Java 2 Platform, Micro Edition (J2ME)* é voltada ao desenvolvimento de aplicações para dispositivos pequenos (como telefones celulares, *pdagers* e assistentes pessoais digitais) e outras aplicações com restrições de memória. A quantidade de tópicos cobertos em Java se tornou grande demais para um livro. Assim, paralelamente a *Java How to Program, Fourth Edition*, estamos publicando *Advanced Java 2 Platform How to Program*, que enfatiza o desenvolvimento de aplicações com J2EE e aborda diversos tópicos de alto nível da J2SE. Além disto, este livro também inclui uma quantidade substancial de material relacionado à J2ME e o desenvolvimento de aplicações para dispositivos sem-fio.

Então aí está ele! Você está prestes a iniciar um caminho desafiante e recompensador. À medida que avançar, compartilhe conosco suas idéias e opiniões sobre Java e *Java Como programar, Quarta Edição* por correio eletrônico para deitel@deitel.com. Responderemos prontamente.

A Prentice Hall mantém www.prenhall.com/deitel – um site dedicado às nossas publicações feitas através da Prentice Hall, incluindo livros-texto, livros profissionais, *Cyber Classrooms* interativas com multimídia em CD, *Complete Training Courses* (produtos combinados que incluem tanto uma *Cyber Classroom* quanto o livro correspondente), treinamento baseado na Web, artigos eletrônicos, livros eletrônicos e materiais complementares para todos estes produtos. Para cada um de nossos livros, o site contém sites da Web associados, que incluem perguntas feitas com freqüência (ou FAQs – *frequently asked questions*), códigos para *download*, erratas, atualizações, textos e exemplos adicionais, perguntas adicionais de auto-avaliação e novos desenvolvimentos em linguagens de programação e tecnologias de programação orientada a objetos. Se você quiser saber mais sobre os autores ou a Deitel & Associates, Inc., visite o endereço www.deitel.com. Boa sorte!

1.2 O que é um computador?

O *computador* é um dispositivo capaz de realizar cálculos e de tomar decisões lógicas a velocidades de milhões e até mesmo bilhões de vezes mais rápidas do que os seres humanos. Por exemplo, muitos dos computadores pessoais de hoje podem realizar centenas de milhões, até bilhões, de somas por segundo. Uma pessoa operando uma calculadora de mesa pode precisar de décadas para completar o mesmo número de cálculos que um computador pessoal poderoso pode realizar em um segundo. (*Aspectos a ponderar:* como você saberia se a pessoa adicionou os números corretamente? Como você saberia se o computador somou os números corretamente?) Os *supercomputadores* mais rápidos atualmente podem realizar centenas de bilhões de operações por segundo – cálculos que centenas de milhares de pessoas precisariam de um ano para fazer! E computadores de um trilhão de instruções por segundo já estão funcionando em laboratórios de pesquisa!

Os computadores processam *dados* sob o controle de conjuntos de instruções chamados *programas de computador*. Esses programas de computador orientam o computador por conjuntos ordenados de ações especificadas por pessoas chamadas de *programadores de computador*.

Os vários dispositivos que compõem um sistema de computação (como teclado, tela, discos, memória e unidades de processamento) são conhecidos como *hardware*. Os programas de computador que rodam em um computador são conhecidos como *software*. Os custos de *hardware* têm caído significativamente nos últimos anos, a ponto de os computadores pessoais terem-se tornado um bem de consumo popular. Infelizmente, os custos de desenvolvimento de *software* têm aumentado constantemente, à medida que os programadores desenvolvem aplicativos cada vez mais complexos e poderosos sem serem capazes de aprimorar significativamente a tecnologia de desenvolvimento de *software*. Neste livro, você aprenderá métodos comprovados de desenvolvimento de *software* que podem reduzir os custos de desenvolvimento de *software* – refinamento passo a passo de cima para baixo) decomposição funcional e programação orientada a objetos. A programação orientada a objetos é amplamente considerada o avanço significativo que pode aprimorar enormemente a produtividade do programador.

1.3 A organização de um computador

Independentemente das diferenças na aparência física, praticamente todos os computadores podem ser considerados como divididos em seis seções ou *unidades lógicas*. São as seguintes:

1. *Unidade de entrada.* É a seção “receptora” do computador. Ela obtém as informações (dados e programas de computador) a partir de *dispositivos de entrada* e coloca essas informações à disposição de outras unidades de modo que possam ser processadas. A maioria das informações é inserida nos computadores atuais por teclados semelhantes aos de máquina de escrever, dispositivos do tipo “mouse” e discos. Futuramente, a maioria das informações será inserida falando-se com os computadores, varrendo-se imagens eletronicamente e por registro de vídeo.
2. *Unidade de saída.* É a seção de “despacho” do computador. Ela pega as informações processadas pelo computador e coloca-as em vários *dispositivos de saída* de forma a tornar as informações disponíveis para serem utilizadas fora do computador. As informações enviadas para a saída do computador são exibidas em telas, impressas em papel, reproduzidas por alto-falantes, registradas magneticamente em discos e fitas ou utilizadas para controlar outros dispositivos.
3. *Unidade de memória.* É a seção de armazenamento de capacidade relativamente baixa e acesso rápido do computador. Ela retém as informações que foram inseridas pela unidade de entrada para que possam tornar-se imediatamente disponíveis para processamento quando necessário. A unidade de memória também armazena as informações que já foram processadas até que possam ser colocadas em dispositivos de saída pela unidade de saída. A unidade de memória é freqüentemente chamada de *memória, memória principal* ou *memória de acesso aleatório (RAM – random access memory)*.
4. *Unidade de aritmética e de lógica (ALU – arithmetic and logic unit).* É a seção de “fabricação” do computador. É responsável pela realização de cálculos como adição, subtração, multiplicação e divisão. Essa seção contém os mecanismos de decisão que permitem ao computador, por exemplo, comparar dois itens da unidade de memória para determinar se são iguais ou não.
5. *Unidade central de processamento (CPU – central processing unit).* É a seção “administrativa” do computador. Trata-se do coordenador do computador e é responsável pela supervisão das operações das outras seções. A CPU diz à unidade de entrada quando as informações devem ser lidas na unidade de memória, informa à ALU quando as informações da unidade de memória devem ser utilizadas em cálculos e avisa a unidade de saída sobre quando enviar as informações da unidade de memória para certos dispositivos de saída.
6. *Unidade secundária de armazenamento.* É a seção de armazenamento de alta capacidade e de longo prazo do computador. Normalmente, os programas ou dados que não estão sendo utilizados por outras unidades são colocados em dispositivos de armazenamento secundários (como discos) até que sejam necessários, possivelmente horas, dias, meses ou mesmo anos mais tarde. As informações na unidade de armazenamento secundária demoram mais para serem acessadas que as informações na memória principal. O custo por unidade secundária de armazenamento é muito menor que o custo por unidade da memória principal.

1.4 Evolução dos sistemas operacionais

Os computadores antigos eram capazes de realizar somente um *trabalho* ou uma *tarefa* por vez. Essa forma de operação do computador é freqüentemente chamada de *processamento em lotes* de um único usuário. O computador executa um único programa por vez enquanto está processando dados em grupos ou *lotes*. Nesses sistemas antigos, geralmente os usuários submetiam seus trabalhos ao centro de processamento de dados em maços de cartões perfurados. Os usuários freqüentemente tinham de esperar horas ou mesmo dias antes de as impressões serem retornadas para suas mesas.

Os sistemas de *software* chamados *sistemas operacionais* foram desenvolvidos com o intuito de ajudar a tornar mais conveniente o uso dos computadores. Os sistemas operacionais anteriores gerenciavam a transição suave entre os trabalhos. Isso minimizou o tempo necessário para que os operadores de computador alternassem entre trabalhos e, portanto, aumentou a quantidade de trabalho, ou *throughput*, que os computadores poderiam processar.

Quando os computadores se tornaram mais poderosos, tornou-se evidente que o processamento em lotes de um único usuário raramente utilizava os recursos de computador de maneira eficiente. Em vez disso, imaginou-se que muitos trabalhos ou tarefas poderiam *compartilhar* os recursos do computador para obterem melhor utilização. Is-

so é chamado de *multiprogramação*. A multiprogramação envolve a operação “simultânea” de muitos trabalhos no computador – o computador compartilha seus recursos entre os trabalhos que competem por sua atenção. Com os sistemas operacionais anteriores de multiprogramação, os usuários ainda submetiam trabalhos em cartões perfurados e esperavam horas ou dias pelos resultados.

Na década de 60, vários grupos na indústria e nas universidades foram os pioneiros dos sistemas operacionais de *compartilhamento de tempo*. O compartilhamento de tempo é um caso especial da multiprogramação em que os usuários acessam o computador através de *terminais*, em geral dispositivos com teclados e telas. Em um típico sistema de computador baseado em compartilhamento de tempo, pode haver dezenas ou mesmo centenas de usuários compartilhando o computador de cada vez. O computador não executa de fato todos os trabalhos dos usuários simultaneamente. Em vez disso, ele executa uma pequena parte de trabalho de um usuário e segue para atender o próximo usuário. O computador faz isso tão rapidamente que pode fornecer serviço para cada usuário várias vezes por segundo. Assim, os programas dos usuários *parecem* estar rodando simultaneamente. Uma vantagem do compartilhamento de tempo é que o usuário recebe respostas quase imediatas às solicitações, em vez de ter de esperar muito tempo pelos resultados, como ocorria com os modos de computação anteriores. Da mesma forma, se um usuário particular estiver atualmente desocupado, o computador pode continuar a servir outros usuários em vez de esperar por um usuário específico.

1.5 Computação pessoal distribuída e computação cliente/servidor

Em 1977, a Apple Computer popularizou o fenômeno da *computação pessoal*. Inicialmente, era um sonho de aficionados. Os computadores tornaram-se suficientemente econômicos para as pessoas comprá-los para seu próprio uso pessoal. Em 1981, a IBM, o maior fornecedor de computadores do mundo, lançou o IBM Personal Computer. Quase da noite para o dia a computação pessoal tornou-se uma realidade no comércio, na indústria e nos órgãos do governo.

Mas esses computadores eram unidades “independentes” (“*stand-alone*”) – as pessoas faziam seus trabalhos em suas próprias máquinas e então transportavam discos de um lado para o outro para compartilhar informações. Embora os computadores pessoais antigos não fossem suficientemente poderosos para compartilhamento de tempo de vários usuários, essas máquinas podiam ser interconectadas em redes de computadores, às vezes por linhas telefônicas e às vezes em *redes locais (LAN – local area network)* dentro de uma organização. Isso levou ao fenômeno da *computação distribuída*, em que a computação de uma organização, em vez de ocorrer estritamente nas instalações de algum computador central, é distribuída através das redes entre os locais em que o trabalho real da organização é realizado. Os computadores pessoais eram suficientemente poderosos tanto para tratar as necessidades de computação dos usuários como tratar as tarefas básicas de comunicação na transmissão de informações de um lado para o outro eletronicamente.

Os computadores pessoais atuais são tão poderosos quanto as máquinas de milhões de dólares de apenas uma década atrás. As máquinas *desktop* mais poderosas – chamadas *estações de trabalho* – fornecem enormes capacidades aos usuários individuais. As informações são facilmente compartilhadas através das redes de computadores nas quais alguns computadores, chamados de *servidores de arquivos*, oferecem um armazenamento comum de programas e dados que pode ser utilizado por computadores *clientes* distribuídos por toda a rede (daí o termo *computação cliente/servidor*). C e C++ tornaram-se e continuam sendo as linguagens de programação preferidas para escrever sistemas operacionais. Elas também continuam populares para escrever aplicações para redes de computadores, aplicações distribuídas cliente/servidor e aplicações para a Internet e a Web, embora Java seja agora a linguagem predominante em cada uma destas áreas. Muitos programadores descobriram que a programação em Java os ajuda a serem mais produtivos que a programação em C ou C++. Sistemas operacionais populares atuais como UNIX, Linux, Mac OS, Windows e Windows 2000 fornecem os tipos de recursos discutidos nesta seção.

1.6 Linguagens de máquina, linguagens assembler e linguagens de alto nível

Os programadores escrevem instruções em várias linguagens de programação, algumas diretamente compreensíveis pelo computador e outras que exigem passos intermediários de *tradução*. Centenas de linguagens de computador estão atualmente em uso. Essas linguagens podem ser divididas em três tipos gerais:

1. Linguagens de máquina

2. Linguagens *assembler*

3. Linguagens de alto nível

Qualquer computador pode entender diretamente somente sua própria *linguagem de máquina*. A linguagem de máquina é a “linguagem natural” de um computador específico. Ela é definida pelo projeto de *hardware* desse computador. As linguagens de máquina consistem geralmente em seqüências de números (em última instância reduzidas a 1s e 0s) que instruem os computadores a realizar suas operações mais elementares uma de cada vez. As linguagens de máquina são *dependentes de máquina* (isto é, uma linguagem particular de máquina pode ser utilizada apenas em um tipo de computador). As linguagens de máquina são incômodas para os seres humanos, como se pode ver pela seguinte seção de um programa de linguagem de máquina que soma pagamento de horas-extras ao salário básico e armazena o resultado no salário bruto.

```
+1300042774
+1400593419
+1200274027
```

À medida que os computadores se tornaram mais populares, tornou-se evidente que a programação em linguagem de máquina era simplesmente muito lenta e tediosa para a maioria dos programadores. Em vez de utilizar seqüências de números que os computadores poderiam entender diretamente, os programadores começaram a utilizar abreviações semelhantes ao inglês para representar as operações elementares do computador. Essas abreviações semelhantes ao inglês formaram a base das *linguagens assembler*. *Programas-tradutores* chamados de *montadores* foram desenvolvidos para converter programas de linguagem *assembler* em linguagem de máquina a velocidades de computador. A seguinte seção de um programa em linguagem *assembler* também soma pagamento de horas-extras (*overpay*) ao salário básico (*basepay*) e armazena o resultado no salário bruto (*grosspay*), mas um pouco mais claramente que seu equivalente de linguagem de máquina.

```
LOAD BASEPAY
ADD OVERPAY
STORE GROSSPAY
```

Embora tal código seja mais claro para os seres humanos, ele é incompreensível para os computadores até ser traduzido para linguagem de máquina.

O uso do computador aumentou rapidamente com o advento de linguagens *assembler*, mas a programação nessas linguagens ainda exigia muitas instruções para realizar até mesmo a tarefa mais simples. Para acelerar o processo de programação, foram desenvolvidas *linguagens de alto nível* em que instruções únicas poderiam ser escritas para realizar tarefas substanciais. Os programas-tradutores que convertem programas de linguagem de alto nível em linguagem de máquina são chamados de *compiladores*. As linguagens de alto nível permitem aos programadores escrever instruções que parecem com o inglês cotidiano e contêm notações matemáticas comumente utilizadas. Um programa de folha de pagamento escrito em uma linguagem de alto nível talvez contenha uma instrução como

```
grossPay = basePay + overTimePay
```

Obviamente, as linguagens de alto nível são muito mais desejáveis do ponto de vista do programador que qualquer linguagem de máquina ou linguagem *assembler*. C, C++ e Java estão entre as linguagens de programação de alto nível mais amplamente utilizadas e mais poderosas.

O processo de compilação de um programa de linguagem de alto nível em linguagem de máquina pode tomar uma quantidade considerável de tempo de computador. Foram desenvolvidos programas *interpretadores* para executar programas em linguagem de alto nível diretamente, sem a necessidade de compilar os programas para linguagem de máquina. Embora os programas compilados executem programas muito mais rapidamente que os programas interpretados, os interpretadores são populares em ambientes de desenvolvimento de programas em que os programas são recompilados freqüentemente enquanto novos recursos são adicionados e erros são corrigidos. Uma vez que um programa é desenvolvido, uma versão compilada pode ser produzida para rodar mais eficientemente. À medida que estudamos Java, você verá que os interpretadores têm desempenhado um papel especialmente importante em ajudar Java a alcançar seu objetivo de portabilidade entre uma grande variedade de plataformas.

1.7 A história de C++

C++ desenvolveu-se a partir de C, que se desenvolveu a partir de duas linguagens anteriores, BCPL e B. BCPL foi desenvolvida em 1967 por Martin Richards como uma linguagem para escrever *softwares* de sistemas operacionais

e compiladores. Ken Thompson modelou muitos recursos na sua linguagem B com base nos seus correspondentes na BCPL e utilizou B para criar as primeiras versões do sistema operacional UNIX nos Bell Laboratories em 1970 sobre um computador Digital Equipment Corporation PDP-7. BCPL e B eram linguagens “sem tipos” (“typeless”) – cada item de dados ocupava uma “palavra” na memória. Por exemplo, era responsabilidade do programador tratar um item de dados como um número inteiro ou um número real.

A linguagem C foi desenvolvida a partir da linguagem B por Dennis Ritchie na Bell Laboratories e foi originalmente implementada em um computador DEC PDP-11 em 1972. C utiliza muitos conceitos importantes das linguagens BCPL e B enquanto adiciona tipos de dados e outros recursos. Inicialmente, C tornou-se muito conhecida como a linguagem de desenvolvimento do sistema operacional UNIX. Hoje, praticamente todos os novos sistemas operacionais importantes estão escritos em C e/ou em C++. Nas últimas duas décadas, C tornou-se disponível para a maioria dos computadores. C não depende de *hardware*. Com um projeto cuidadoso, é possível escrever programas em C que sejam *portáveis* para a maioria dos computadores.

No final da década de 1970, C tinha se desenvolvido para o que é agora conhecido como “C tradicional”, ou “C de Kernighan e Ritchie”. A publicação pela Prentice Hall em 1978 do livro de Kernighan e Ritchie, *The C Programming Language*, trouxe ampla atenção à linguagem. Essa publicação tornou-se um dos livros mais bem-sucedidos de ciência da computação jamais visto.

A utilização disseminada de C em vários tipos de computadores (às vezes chamados de *plataformas de hardware*) levou a muitas variações, que eram semelhantes, mas freqüentemente incompatíveis. Isso era um problema sério para os programadores que necessitavam escrever programas portáveis que rodassem em várias plataformas. Tornou-se claro que uma versão-padrão de C era necessária. Em 1983, foi criado o comitê técnico X3J11 sob o American National Standards Committee on Computers and Information Processing (X3) para “fornecer uma definição independente de máquina e não-ambígua da linguagem”. Em 1989, o padrão foi aprovado. O ANSI cooperou com a International Standards Organization (ISO) para padronizar C mundialmente; o documento estabelecido em conjunto foi publicado em 1990 e é conhecido como ANSI/ISO 9899: 1990. A segunda edição de Kernighan e Ritchie¹, publicada em 1988, reflete essa versão chamada ANSI C, uma versão da linguagem agora utilizada mundialmente (Ke88).

C++, uma extensão de C, foi desenvolvida por Bjarne Stroustrup no início da década de 1980 na Bell Laboratories. C++ fornece vários recursos que aprimoram a linguagem C, mas, sobretudo, ela fornece recursos para *programação orientada a objetos*. C++ também foi padronizada pelos comitês ANSI e ISO.

Há uma revolução em andamento na comunidade de *softwares*. Construir *softwares* de maneira rápida, correta e econômica permanece um objetivo difícil de alcançar e isso em uma época em que a demanda por novos e mais poderosos programas de *software* está crescendo de maneira impressionante. Os *objetos* são essencialmente *componentes de software* reutilizáveis que modelam itens do mundo real. Os desenvolvedores de *software* estão descobrindo que utilizar uma abordagem de implementação e de projeto modular orientados a objetos e pode tornar os grupos de desenvolvimento de *software* muito mais produtivos do que até então era possível com as técnicas de programação populares anteriores como a programação estruturada. Programas orientados a objetos são freqüentemente mais fáceis de entender, corrigir e modificar.

Muitas outras linguagens orientadas a objetos foram desenvolvidas, incluindo Smalltalk, desenvolvida no Palo Alto Research Center (PARC) da Xerox. Smalltalk é uma linguagem puramente orientada a objetos – literalmente tudo é um objeto. C++ é uma linguagem híbrida – é possível programar tanto no estilo C como no estilo orientado a objetos ou ambos.

1.8 A história de Java

Talvez a contribuição mais importante da revolução dos microprocessadores até esta data seja o fato de ela ter possibilitado o desenvolvimento de computadores pessoais, que agora chegam a centenas de milhões em todo o mundo. Os computadores pessoais tiveram um profundo impacto sobre as pessoas e a maneira como as organizações conduzem e gerenciam seus negócios.

Muitas pessoas acreditam que a próxima área importante em que os microprocessadores terão um impacto profundo serão os dispositivos eletrônicos inteligentes destinados ao consumidor final. Reconhecendo isso, a Sun Microsystems financiou uma pesquisa corporativa interna com o codinome Green em 1991. O projeto resultou no de-

1. Kernighan, B. W., e D. M. Ritchie, *The C Programming Language* (Second Edition), Englewood Cliffs, NJ: Prentice Hall, 1998.

senvolvimento de uma linguagem baseada em C e C++ que seu criador, James Gosling, chamou de Oak (carvalho) em homenagem a uma árvore que dava para a janela do seu escritório na Sun. Descobriu-se mais tarde que já havia uma linguagem de computador chamada Oak. Quando uma equipe da Sun visitou uma cafeteria local, o nome Java (cidade de origem de um tipo de café importado) foi sugerido e pegou.

Mas o projeto Green atravessava algumas dificuldades. O mercado para dispositivos eletrônicos inteligentes destinados ao consumidor final não estava se desenvolvendo tão rapidamente como a Sun tinha previsto. Pior ainda, um contrato importante pelo qual a Sun competia fora concedido a outra empresa. Então, o projeto estava em risco de cancelamento. Por pura sorte, a World Wide Web explodiu em popularidade em 1993 e as pessoas da Sun viram o imediato potencial de utilizar Java para criar páginas da Web com o chamado *conteúdo dinâmico*. Isso deu nova vida ao projeto.

Em maio de 1995, a Sun anunciou Java formalmente em uma conferência importante. Normalmente, um evento como esse não teria gerado muita atenção. Entretanto, Java gerou interesse imediato na comunidade comercial por causa do fenomenal interesse pela World Wide Web. Java é agora utilizada para criar páginas da Web com conteúdo interativo e dinâmico, para desenvolver aplicativos corporativos de grande porte, para aprimorar a funcionalidade de servidores da World Wide Web (os computadores que fornecem o conteúdo que vemos em nossos navegadores da Web), fornecer aplicativos para dispositivos destinados ao consumidor final (como telefones celulares, págers e assistentes pessoais digitais) e para muitas outras finalidades.

1.9 Bibliotecas de classes Java

Os programas em Java consistem em partes chamadas *classes*. Estas consistem em partes chamadas *métodos* que realizam tarefas e retornam as informações ao completarem suas tarefas. Você pode programar cada pedaço que talvez você precise para formar um programa em Java. Entretanto, a maioria dos programadores em Java tira proveito de ricas coleções de classes existentes em *bibliotecas de classes Java*. As bibliotecas de classes são também conhecidas como *Java APIs (Applications Programming Interfaces* – interfaces de programas aplicativos). Portanto, há realmente duas partes para aprender o “mundo” de Java. A primeira é aprender a própria linguagem Java de modo que você possa programar suas próprias classes e a segunda é aprender como utilizar as classes nas extensas bibliotecas de classes Java. Por todo o livro discutimos muitas classes dessas bibliotecas. As bibliotecas de classes são fornecidas principalmente por fornecedores de compiladores, mas muitas bibliotecas de classes são fornecidas por fornecedores independentes de *software* (ISV – *independent software vendor*). Da mesma forma, muitas bibliotecas de classes estão disponíveis a partir da Internet e da World Wide Web como *freeware* ou *shareware*. Você pode baixar os produtos *freeware* e usá-los de graça – sujeitando-se a quaisquer restrições especificadas pelo proprietário dos direitos de cópia. Você também pode baixar produtos *shareware* de graça, para experimentar o *software*. Os produtos *shareware* freqüentemente são isentos de pagamento para uso pessoal. Entretanto, para os produtos que você usa regularmente ou usa para fins comerciais, espera-se que você pague uma taxa especificada pelo proprietário dos direitos de cópia.

Muitos produtos *freeware* e *shareware* são de *código-fonte aberto*. O código-fonte para os produtos com fonte aberta está disponível gratuitamente na Internet, o que lhe permite aprender estudando o código-fonte, confirmar que o código serve para a finalidade indicada para ele e até mesmo modificar o código. Freqüentemente, produtos de fonte aberta exigem que você publique quaisquer aperfeiçoamentos que você faça, de modo que a comunidade de fonte aberto possa continuar a evoluir aqueles produtos. Um exemplo de um produto de fonte aberto muito popular é o sistema operacional Linux.



Observação de engenharia de software 1.1

Utilize uma abordagem de bloco de construção para criar programas. Evite reinventar a roda. Utilize partes existentes – isso se chama reutilização de software e é um ponto central da programação orientada a objetos.

[Nota: incluiremos muitas dessas Observações de engenharia de *software* por todo o texto para explicar conceitos que afetam e aprimoram a arquitetura total e a qualidade de um sistema de *software* e, particularmente, de grandes sistemas de *software*. Também iremos destacar Boas práticas de programação (práticas que podem ajudar a escrever programas que sejam mais claros, mais compreensíveis, mais sustentáveis e mais fáceis de testar e depurar), Erros comuns de programação (problemas com os quais se deve ter cuidado para não se cometer os mesmos erros em seus programas), Dicas de desempenho (técnicas que o ajudarão a escrever programas que rodem mais rápido e utilizem menos memória), Dicas de portabilidade (técnicas que o ajudarão a escrever programas que possam rodar, com pequena ou nenhuma modificação, em uma variedade de computadores; essas dicas também incluem observa-

ções gerais sobre como Java alcança seu alto grau de portabilidade), Dicas de teste e depuração (técnicas que o ajudarão a remover *bugs* de seus programas e, sobretudo, técnicas que o ajudarão a escrever programas sem *bugs* para começar) e Observações de aparência e comportamento (técnicas que o ajudarão a projetar a “aparência” e o “comportamento” de suas interfaces gráficas com o usuário para melhor aparência e facilidade de uso). Muitas dessas técnicas e práticas são somente diretrizes; você, sem dúvida, desenvolverá seu próprio estilo de programação preferido.]



Observação de engenharia de software 1.2

Ao programar em Java, você geralmente utilizará os seguintes blocos de construção: classes de bibliotecas de classes, classes e métodos que você mesmo cria e classes e métodos que outras pessoas criam e tornam disponíveis.

A vantagem de criar suas próprias classes e métodos é que você sabe exatamente como eles funcionam e você pode examinar o código em Java. A desvantagem é o consumo de tempo e a complexidade do esforço necessário para projetar e desenvolver novas classes e métodos.



Dica de desempenho 1.1

Utilizar classes e métodos da Java API em vez de escrever suas próprias versões pode melhorar o desempenho do programa, uma vez que essas classes e métodos são cuidadosamente escritos para rodar de maneira eficiente. Esta técnica também melhora a velocidade de prototipagem no desenvolvimento de programas (i.e., o tempo necessário para desenvolver um programa novo e fazer funcionar a sua primeira versão).



Dica de portabilidade 1.1

Utilizar classes e métodos da Java API em vez de escrever suas próprias versões melhora a portabilidade do programa, uma vez que essas classes e métodos são incluídos em todas as implementações de Java (supondo que tenham o mesmo número de versão).



Observação de engenharia de software 1.3

Extensas bibliotecas de classes de componentes de software reutilizáveis estão disponíveis na Internet e na Web. Muitas dessas bibliotecas fornecem o código-fonte e estão disponíveis sem nenhum custo.

1.10 Outras linguagens de alto nível

Centenas de linguagens de alto nível foram desenvolvidas, mas somente algumas alcançaram ampla aceitação. *Fortran* (FORmula TRANslator) foi desenvolvida pela IBM Corporation entre 1954 e 1957 para ser utilizada em aplicações científicas de engenharia que requerem cálculos matemáticos complexos. Fortran é ainda amplamente utilizada.

COBOL (COmmon Business Oriented Language) foi desenvolvida em 1959 por um grupo de fabricantes de computadores e usuários de computadores industriais e governamentais. COBOL é utilizada principalmente para aplicativos comerciais que exigem manipulação precisa e eficiente de grandes quantidades de dados. Hoje, aproximadamente metade de todos os *softwares* de negócios ainda é programada em COBOL. Quase um milhão de pessoas estão ativamente escrevendo programas em COBOL.

Pascal foi projetada quase na mesma época que C. Foi criada pelo professor Nicklaus Wirth e foi projetada para utilização acadêmica. Falaremos mais sobre Pascal na próxima seção.

Basic foi desenvolvida em 1965 na Dartmouth University como uma linguagem simples para ajudar as iniciantes a se sentirem à vontade com programação. Bill Gates implementou Basic em vários dos primeiros computadores pessoais. Hoje, a Microsoft – empresa criada por Bill Gates – é a principal organização de desenvolvimento de *software* do mundo.

1.11 Programação estruturada

Durante a década de 1960, muitos grandes esforços no desenvolvimento de *software* encontraram diversas dificuldades. Em geral, os cronogramas de produção de *software* se atrasavam, os custos excediam enormemente os orçamentos.

mentos e os produtos finais não eram confiáveis. As pessoas começaram a perceber que o desenvolvimento de *software* era uma atividade muito mais complexa do que haviam imaginado. Em 1960, a atividade de pesquisa resultou na evolução da *programação estruturada* – uma abordagem disciplinada para escrever programas que são mais claros, mais fáceis de testar e depurar e mais fáceis de modificar do que programas não-estruturados. Os Capítulos 4 e 5 discutem os princípios da programação estruturada.

Um dos resultados mais tangíveis dessa pesquisa foi o desenvolvimento da linguagem de programação Pascal por Nicklaus Wirth em 1971. Pascal, em homenagem a Blaise Pascal, matemático e filósofo do século 17, foi projetada para o ensino de programação estruturada em ambientes acadêmicos e rapidamente tornou-se a linguagem de programação preferida na maioria das universidades. Infelizmente, a linguagem carece de muitos recursos necessários para torná-la útil em aplicações industriais, comerciais e governamentais, o que a impediu de ser amplamente aceita nesses ambientes.

A linguagem de programação Ada foi desenvolvida sob o patrocínio do Departamento de Defesa dos Estados Unidos (DOD) durante a década de 1970 e o início da década de 1980. Centenas de linguagens distintas estavam sendo utilizadas a fim de produzir os volumosos sistemas de *software* para controle e comando do DOD. O DOD queria uma única linguagem que atendesse à maioria de suas necessidades. Pascal foi escolhida como base, mas a linguagem final Ada é bem diferente de Pascal. O nome da linguagem veio de Lady Ada Lovelace, a filha do poeta Lord Byron. Atribuem-se a Lady Lovelace os créditos de ter escrito o primeiro programa de computador do mundo no início de 1800 (o dispositivo mecânico de computação conhecido como Máquina Analítica, projetado por Charles Babbage). Um recurso importante de Ada é a chamada *multitarefa*; isto permite que os programadores especifiquem que muitas atividades ocorrerão paralelamente. Os recursos nativos de outras linguagens de alto nível amplamente utilizadas que discutimos – incluindo C e C++ – geralmente permitem que o programador escreva programas que realizam apenas uma atividade por vez. Java, através de uma técnica que explicaremos mais adiante chamada de *multithreading (multiescalonamento)*, também permite que os programadores escrevam programas com atividades paralelas. [Nota: a maioria dos sistemas operacionais inclui bibliotecas específicas para plataformas individuais (às vezes chamadas de *bibliotecas dependentes de plataforma*) que permitem às linguagens de alto nível como C e C++ especificar que muitas atividades devem ocorrer em paralelo em um programa.]

1.12 A Internet e a World Wide Web

A *Internet* foi desenvolvida há mais de três décadas, financiada pelo Departamento da Defesa dos EUA. Originalmente projetada para conectar os principais sistemas de computadores de cerca de uma dezena de universidades e organizações de pesquisa, a Internet atualmente é acessível a centenas de milhões de computadores no mundo.

Com a introdução da *World Wide Web* – que permite que os usuários de computador localizem e visualizem documentos baseados em multimídia sobre quase todos os assuntos – a Internet explodiu, para se tornar um dos principais mecanismos de comunicação do mundo.

Ao entrarmos no próximo milênio, a Internet e a *World Wide Web* seguramente estarão listadas entre as mais profundas e importantes criações da raça humana. Antigamente, a maioria dos aplicativos de computador rodava em computadores que não se comunicavam entre si. Atualmente, podem-se escrever aplicativos que se comunicam com as centenas de milhões de computadores do mundo. A Internet funde tecnologias de computação e de comunicações. Ela torna nosso trabalho mais fácil. Torna informações acessíveis mundialmente de forma instantânea e conveniente. Possibilita que indivíduos e empresas locais de pequeno porte tenham uma exposição mundial. Ela está alterando a forma como o comércio é feito. As pessoas podem procurar os melhores preços de praticamente todos os produtos ou serviços. Comunidades de interesse especial podem permanecer em contato entre si. Pesquisadores podem se informar instantaneamente sobre os últimos avanços em qualquer parte do globo.

Java Como Programar: Quarta Edição apresenta técnicas de programação que permitem que os aplicativos em Java utilizem a Internet e a *World Wide Web* para interagir com outros aplicativos. Esses recursos e os recursos discutidos em nosso livro associado *Advanced Java 2 Platform How to Program* permitem que os programadores em Java desenvolvam os tipos de aplicativos distribuídos de nível corporativo que são usados atualmente nas empresas. Podem-se escrever aplicativos Java em qualquer plataforma de computador, o que resulta em uma importante economia de tempo e custo de desenvolvimento de sistemas para as corporações. Se você ultimamente tem ouvido falar muito sobre a Internet e a *World Wide Web* e se está interessado em desenvolver aplicativos para rodar na Internet e na Web, aprender Java pode ser a chave de oportunidades desafiadoras e recompensadoras para sua carreira.

1.13 Princípios básicos de um ambiente Java típico

Os sistemas Java geralmente consistem em várias partes: um ambiente, a linguagem, a interface de programas aplicativos (API – *Applications Programming Interface*) Java e várias bibliotecas de classes. A discussão a seguir explica um ambiente de desenvolvimento de programa Java típico, como mostra a Fig. 1.1.

Os programas Java normalmente passam por cinco fases para serem executados (Fig. 1.1). São as seguintes: *edição, compilação, carga, verificação e execução*. Discutimos estes conceitos no contexto do *Java 2 Software Development Kit (J2SDK)* que está incluído no CD que acompanha este livro. *Siga cuidadosamente as instruções de instalação para o J2SDK fornecidas no CD para assegurar que você ajuste seu computador apropriadamente para compilar e executar programas em Java.* [Nota: se você não está utilizando o UNIX/Linux, o Windows 95/98/ME ou o Windows NT/2000, consulte os manuais do ambiente Java do seu sistema, ou pergunte a seu instrutor como realizar essas tarefas em seu ambiente (que provavelmente será semelhante ao ambiente na Fig. 1.1).

A Fase 1 consiste em editar um arquivo. Isso é realizado com um *programa editor* (normalmente conhecido como *editor*). O programador digita um programa em Java utilizando o editor e faz correções se necessário. Quando o programador especifica que o arquivo deve ser salvo, o programa é armazenado em um dispositivo de armazenamento secundário, como um disco. Nomes de arquivos de programa Java terminam com a extensão `.java`. Dois editores amplamente utilizados em sistemas UNIX/Linux são o `vi` e o `emacs`. No Windows 95/98/ME e no Windows NT/2000, programas de edição simples como o comando `Edit` do DOS e o Windows Notepad são suficientes. Ambientes de desenvolvimento integrado (IDEs – *integrated development environments*) Java como Forté for Java Community Edition, NetBeans, o JBuilder da Borland, o Visual Cafe da Symantec e o Visual Age da IBM têm editores embutidos que são transparentemente integrados ao ambiente de programação. Estamos pressupondo que o leitor saiba como editar um arquivo.

[Observe que o Forté for Java Community Edition está escrito em Java e é gratuito para utilização não-comercial. Ele está incluído no CD que acompanha este livro. A Sun atualiza este software aproximadamente duas vezes por ano. Versões mais novas podem ser baixadas do endereço

www.sun.com/forte/ffj

Forté for Java Community Edition pode ser executado na maioria das principais plataformas. Este livro está escrito para qualquer ambiente de desenvolvimento Java 2 genérico. Ele não depende do Forté for Java Community Edition. Nossos programas de exemplo devem operar adequadamente na maioria dos ambientes integrados de desenvolvimento Java.]

Na Fase 2 (discutida novamente nos Capítulos 2 e 3), o programador emite o comando `javac` para *compilar* o programa. O compilador Java traduz o programa Java para *bytecodes* – a linguagem entendida pelo interpretador Java. Para compilar um programa chamado `Welcome.java`, digite

```
javac Welcome.java
```

na janela de comando de seu sistema (i.e., o *prompt* do MS-DOS no Windows, no *Command Prompt* no Windows NT/2000 ou no *prompt* do *shell* no UNIX/Linux). Se o programa compilar corretamente, o compilador produz um arquivo chamado `Welcome.class`. Esse é o arquivo que contém os *bytecodes* que serão interpretados durante a fase de execução.

A Fase 3 é chamada de *carga*. O programa deve ser primeiramente colocado na memória antes de poder ser executado. Isso é feito pelo *carregador de classe*, que pega o arquivo (ou arquivos) `.class` que contém os *bytecodes* e o transfere para a memória. O arquivo `.class` pode ser carregado a partir de um disco em seu sistema ou através de uma rede (como a rede local de sua universidade ou empresa, ou mesmo a Internet). Há dois tipos de programas para os quais o carregador de classe carrega arquivos `.class` – *aplicativos* e *applets*. O aplicativo é um programa (tal como um programa processador de texto, um programa de planilha, um programa de desenho ou um programa de correio eletrônico) que normalmente é armazenado e executado a partir do computador local do usuário. O *applet* é um programa pequeno que normalmente é armazenado em um computador remoto que os usuários conectam através de um navegador da World Wide Web. Os *applets* são carregados no navegador a partir de um computador remoto, são executados no navegador e descartados quando se completa a execução. Para executar um *applet* novamente, o usuário deve direcionar um navegador para a localização apropriada na World Wide Web e recarregar o programa no navegador.

Os aplicativos são carregados na memória e executados com o *interpretador Java* através do comando `java`. Ao executar um aplicativo Java chamado `Welcome`, o comando

```
java Welcome
```

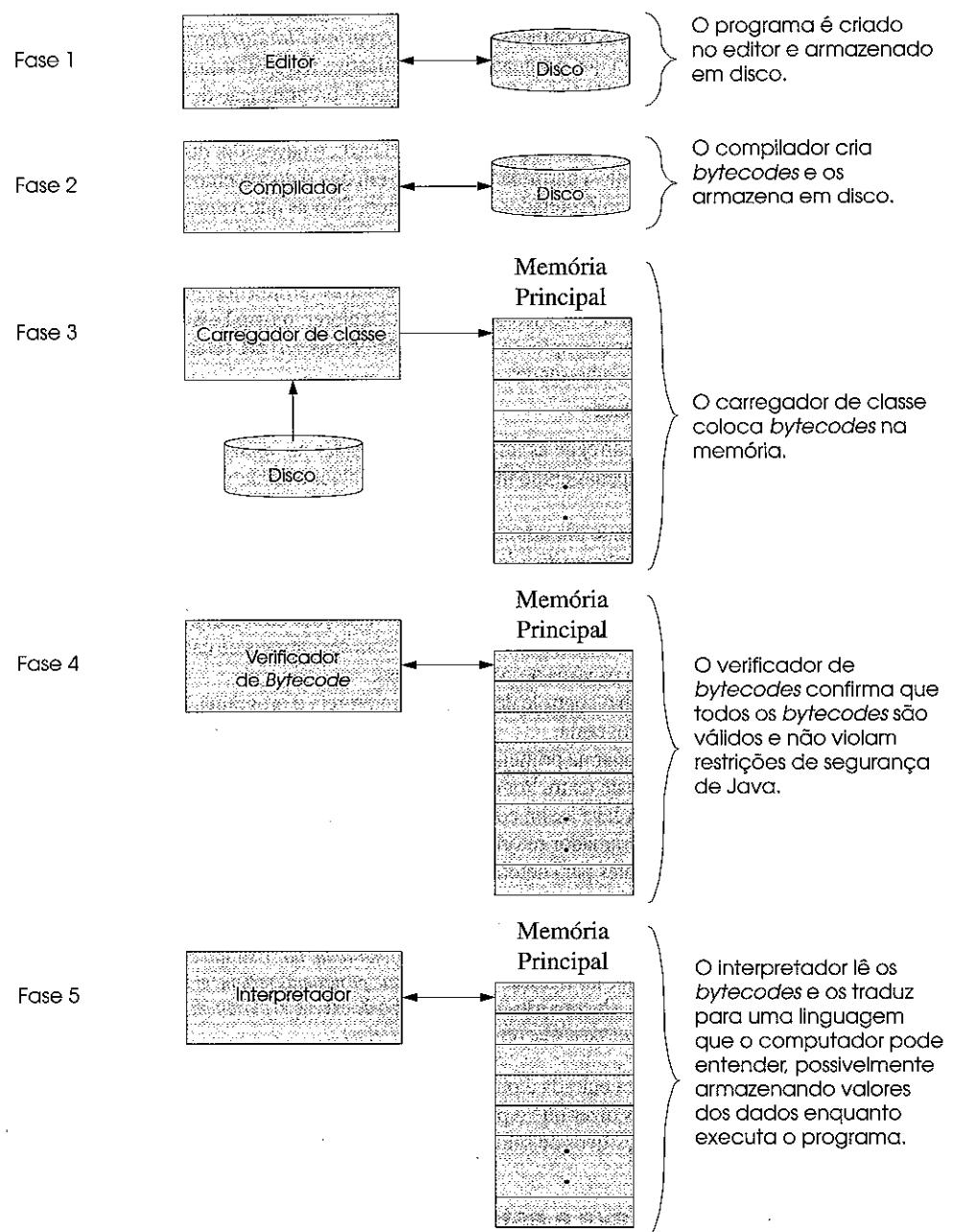


Fig. 1.1 Ambiente Java típico.

invoca o interpretador para o aplicativo **Welcome** e faz com que o carregador de classe carregue as informações utilizadas no programa **Welcome**. [Nota: muitos programadores Java se referem ao interpretador como a *Java Virtual Machine*, ou *JVM* – a máquina virtual Java.]

O carregador de classe também é executado quando um navegador da World Wide Web como o *Netscape Navigator* ou o Microsoft *Internet Explorer* carrega um *applet Java*. Os navegadores são utilizados para visualizar documentos na World Wide Web denominados documentos *HTML (Hypertext Markup Language)*. HTML descreve o formato de um documento de uma maneira que é entendida pelo aplicativo navegador (apresentamos HTML na Seção 3.4; para um tratamento detalhado de HTML e outras tecnologias de programação de Internet, veja nosso *Internet and World Wide Web How to Program, Second Edition*). O documento HTML pode se referir a um *applet Java*. Quando o navegador vê um *applet* mencionado em um documento HTML, o navegador dispara o carregador de classe Java para carregar o *applet* (normalmente a partir da localização em que o documento HTML está armazenado). Cada navegador que suporta Java tem um interpretador Java embutido. Depois que o *applet* é carregado, o interpretador Java do navegador o executa. Os *applets* também podem ser executados a partir da linha de comando utilizando o comando **appletviewer** fornecido com o J2SDK – o conjunto de ferramentas que inclui o compilador (**javac**), o interpretador (**java**), o visualizador (**appletviewer**) e outras ferramentas utilizadas por programadores Java. Assim como o Netscape Navigator e o Microsoft Internet Explorer, o **appletviewer** requer um documento HTML para invocar um *applet*. Por exemplo, se o arquivo **Welcome.html** refere-se ao *applet Welcome*, o comando **appletviewer** é utilizado como segue:

```
appletviewer Welcome.html
```

Isso faz com que o carregador de classe carregue as informações utilizadas no *applet Welcome*. O **appletviewer** é um navegador mínimo – ele sabe apenas como interpretar referências a *applets* e ignora todo o resto do código HTML em um documento.

Antes que o interpretador Java embutido em um navegador ou o **appletviewer** executem os *bytecodes* em um *applet*, os *bytecodes* são verificados pelo *verificador de bytecode* na Fase 4 (isso também acontece em aplicativos que baixam *bytecodes* de uma rede). Isso assegura que os *bytecodes* para as classes que são baixadas da Internet (conhecidas como *classes baixadas*) são válidos e não violam as restrições de segurança de Java. Java impõe intensa segurança porque os programas Java baixados da rede não devem ser capazes de causar danos aos seus arquivos e ao seu sistema (como acontece com os vírus de computadores).

Por fim, na Fase 5, o computador, sob o controle de sua CPU, interpreta o programa, um *bytecode* por vez, realizando assim a ação especificada pelo programa.

Alguns programas podem não funcionar na primeira tentativa. Cada uma das fases precedentes pode falhar por causa de vários erros que discutiremos neste texto. Por exemplo, um programa executável talvez tente realizar uma divisão por zero (uma operação ilegal em Java assim como em aritmética). Isso faria com que o programa Java imprimisse uma mensagem de erro. O programador retornaria para a fase de edição, faria as correções necessárias e prosseguiria novamente pelas fases restantes para determinar se as correções funcionaram adequadamente.



Erro comum de programação 1.1

Erros como o de divisão por zero ocorrem enquanto o programa roda, por isso são chamados de erros em tempo de execução. Erros fatais em tempo de execução fazem com que programas terminem imediatamente sem ter realizado seus trabalhos com sucesso. Os erros não-fatais em tempo de execução permitem que os programas rodem até a sua conclusão, produzindo freqüentemente resultados incorretos.

A maioria dos programas Java realiza entrada e/ou saída de dados. Quando dizemos que um programa imprime um resultado, normalmente queremos dizer que o programa exibe os resultados na tela do computador. Os dados podem ser enviados para outros dispositivos de saída, como discos e impressoras.

1.14 Notas gerais sobre Java e este livro

Java é uma linguagem poderosa. Os programadores experientes às vezes ficam orgulhosos por serem capazes de fazer algum uso exótico, desvirtuado ou distorcido de uma linguagem. Essa é uma prática de programação pobre. Ela torna os programas mais difíceis de ler, mais propensos a se comportarem estranhamente, mais difíceis de se testar e de depurar e mais difíceis de adaptar em caso de alteração de requisitos. Este livro também é projetado para os programadores iniciantes, por isso enfatizamos a *clareza*. Segue nossa primeira “boa prática de programação”.



Boa prática de programação 1.1

Escreva seus programas em Java de uma maneira simples e direta. Isso é às vezes chamado de KIS (“keep it simple”, “mantenha a coisa simples”). Não “estique” a linguagem, tentando usos bizarros.

Você já ouviu dizer que Java é uma linguagem portável e que os programas escritos em Java podem rodar em muitos computadores diferentes. Para a programação em geral, *portabilidade é um objetivo vago*. Por exemplo, o documento-padrão ANSI² de C contém uma longa lista de questões de portabilidade; e foram escritos livros inteiros que discutem portabilidade^{3,4}.



Dica de portabilidade 1.2

Embora seja mais fácil escrever programas portáveis em Java do que em outras linguagens de programação, há diferenças entre compiladores, interpretadores e computadores que podem tornar difícil alcançar a portabilidade. Simplesmente escrever programas em Java não garante a portabilidade. Ocasionalmente, o programador precisará lidar diretamente com variações de compilador e computador.



Dica de teste e depuração 1.1

Sempre teste seus programas Java em todos os sistemas em que você pretende executar esses programas, para assegurar que seus programas Java funcionarão corretamente para o público ao qual são dirigidos.

Fizemos uma cuidadosa revisão da documentação de Java da Sun e a comparamos com nossa apresentação quanto à abrangência e precisão. Mas Java é uma linguagem rica e há algumas sutilezas na linguagem e alguns tópicos que não abrangemos. Se você precisar de detalhes técnicos adicionais sobre Java, sugerimos a leitura da documentação de Java mais atual disponível na Internet em java.sun.com. Nossa livro contém uma extensa bibliografia de livros e artigos sobre linguagem Java em particular e sobre programação orientada a objetos em geral. Uma versão baseada na Web da documentação da Java API pode ser encontrada em java.sun.com/j2se/1.3/docs/api/index.html. Você também pode baixar esta documentação para seu próprio computador a partir de java.sun.com/j2se/1.3/docs.html.



Boa prática de programação 1.2

Leia a documentação da versão de Java que você está utilizando. Consulte essa documentação freqüentemente para se certificar de que você esteja ciente da rica coleção de recursos Java e de estar utilizando esses recursos corretamente.



Boa prática de programação 1.3

Seu computador e compilador são bons professores. Se depois de ler cuidadosamente o manual de documentação de Java você não estiver certo sobre como funciona um recurso de Java, teste-o para ver o que acontece. Estude cada erro ou mensagem de aviso obtida ao compilar seus programas e corrija os programas para eliminar essas mensagens.



Boa prática de programação 1.4

O Java 2 Software Development Kit contém o código-fonte em Java. Muitos programadores leem o código-fonte das classes da Java API para determinar como estas classes funcionam e aprender técnicas de programação adicionais. Se a documentação da Java API não for clara sobre um tópico particular, tente estudar o código-fonte da classe.

Neste livro, explicamos como funciona Java em suas implementações atuais. Talvez o problema mais surpreendente nas primeiras versões de Java é que os programas em Java rodam interpretativamente na máquina do cliente. Os interpretadores rodam lentamente comparados aos códigos de máquina integralmente compilados.



Dica de desempenho 1.2

Os interpretadores têm uma vantagem sobre os compiladores para o mundo de Java, isto é, um programa interpretado pode começar a executar imediatamente logo que baixado na máquina do cliente, ao passo que para um programa-fonte ser compilado ele deve primeiro sofrer uma demora potencialmente longa porque deve ser compilado antes de poder ser executado.

² ANSI, American National Standard for Information Systems – Programming Language C (ANSI Document ANSI/ISO 9899: 1990), New York, NY: American National Standards Institute, 1990.

³ Jaeschke, R., *Portability and the C Language*, Indianapolis, IN: Hayden Books, 1989.

⁴ Rabinowitz, H. e C. Schaap, *Portable C*, Englewood Cliffs, NJ: Prentice Hall, 1990.



Dica de portabilidade 1.3

Embora apenas os interpretadores Java estivessem disponíveis para executar bytecodes no site do cliente nos primeiros sistemas Java, foram escritos compiladores Java que traduzem bytecodes Java (ou, em alguns casos, o código fonte de Java) para o código de máquina nativo da máquina do cliente, para a maioria das plataformas mais populares. Esses programas compilados têm desempenho comparável ao de códigos C ou C++ compilados. Entretanto, não há compiladores de bytecodes para cada plataforma Java, de modo que os programas Java não terão o mesmo nível de desempenho em todas as plataformas.

Os applets apresentam algumas questões mais interessantes. Lembre-se, um applet pode estar vindo de praticamente qualquer servidor da Web no mundo. Assim, ele terá de ser capaz de executar em qualquer plataforma Java possível.



Dica de portabilidade 1.4

Applets Java de rápida e curta execução certamente podem ainda ser interpretados. Mas e quanto aos applets mais substanciais e que utilizam mais o computador? Neste caso, o usuário pode estar disposto a aceitar uma demora de compilação para obter melhor desempenho de execução. Para alguns applets de uso especialmente mais intenso do computador, o usuário pode não ter nenhuma escolha; o código interpretado rodaria muito lentamente para um desempenho aceitável do applet; assim, o applet teria de ser compilado.



Dica de portabilidade 1.5

Um passo intermediário entre interpretadores e compiladores é um compilador just-in-time (JIT) que, enquanto o interpretador é executado, produz um código compilado para os programas e executa os programas em linguagem de máquina em vez de reinterpretá-los. Os compiladores JIT não produzem linguagem de máquina tão eficiente quanto um compilador completo.



Dica de portabilidade 1.6

Para obter as informações mais recentes sobre tradução de programas Java em alta velocidade, você pode querer ler sobre o compilador HotSpot™ da Sun; visite o endereço java.sun.com/products/hotspot. O compilador HotSpot é um componente-padrão do Java 2 Software Development Kit.

O compilador Java, `javac`, não é um compilador tradicional no sentido que ele não converte um programa Java de código-fonte para o código de máquina nativo para uma plataforma de computação particular. Em vez disso, o compilador Java traduz o código-fonte para *bytecodes*. Os *bytecodes* são a linguagem da máquina virtual Java – um programa que simula a operação de um computador e roda sua própria linguagem de máquina (i.e., *bytecodes* Java). A máquina virtual Java é implementada no J2SDK como o interpretador `java`, que traduz os *bytecodes* para linguagem de máquina nativa para a plataforma de computação local.



Observação de engenharia de software 1.4

Para as organizações que queiram desenvolver sistemas de informação para trabalho pesado, estão disponíveis ambientes integrados de desenvolvimento (IDEs – Integrated Development Environments) a partir de muitos fornecedores de software importantes, incluindo a Sun Microsystems. Os IDEs fornecem muitas ferramentas para suportar o processo de desenvolvimento de software, como editores para escrever e editar programas, depuradores para localizar erros de lógica em programas e muitos outros recursos.



Observação de engenharia de software 1.5

O poderoso IDE Java da Sun Microsystems – Forté for Java, Community Edition – está disponível no CD que acompanha este livro e pode ser baixado do endereço www.sun.com/forte/ffj.

1.15 Pensando em objetos: introdução à tecnologia de objetos e à Unified Modeling Language

Agora começamos nossa introdução antecipada à orientação a objetos. Veremos que a orientação a objetos é um modo natural de pensar sobre o mundo e de escrever programas de computador.

No corpo de cada um dos sete primeiros capítulos nos concentramos na metodologia “convencional” da programação estruturada, porque os objetos que vamos construir serão compostos, em parte, por pedaços de programas estruturados. Entretanto, terminamos cada capítulo com uma seção “Pensando em objetos” na qual apresentamos uma introdução cuidadosamente encadeada à orientação a objetos. Nossa objetivo, nestas seções “Pensando em objetos”, é ajudá-lo a desenvolver uma maneira de pensar orientada a objetos, de modo que você possa usar imediatamente as técnicas de programação orientada a objetos que apresentamos a partir do Capítulo 8. As seções “Pensando em objetos” também apresentam a *Unified Modeling Language (UML)*. A UML é uma linguagem gráfica que permite às pessoas que constroem sistemas (p. ex., projetistas de software, engenheiros de sistemas, programadores, etc.) representar seus projetos orientados a objetos usando uma notação comum.

Nesta seção, apresentamos conceitos básicos (i.e., “pensar em objetos”) e terminologia (i.e., “falar em objetos”). Os Capítulos 2 a 13, 15 e 22 e os Apêndices G a I incluem seções “Pensando em objetos” opcionais, que apresentam um estudo de caso substancial que aplica as técnicas de *projeto orientado a objetos (OOD – object oriented design)*. As seções opcionais nos finais dos Capítulos 2 a 7 analisam uma definição típica de problema, que requer a construção de um sistema, determinam os objetos necessários para implementar aquele sistema, determinam os atributos que os objetos terão, determinam os comportamentos que estes objetos irão exibir e especificam como os objetos irão interagir uns com os outros para atender aos requisitos do sistema. Tudo isso acontece *antes* que você aprenda a escrever programas orientados a objetos em Java! As seções opcionais “Pensando em objetos” nos finais dos Capítulos 8 a 13 e 15 modificam e melhoram o projeto apresentado nos Capítulos 2 a 7. O Capítulo 22 apresenta como exibir nosso projeto rico em multimídia na tela. As seções “Pensando em objetos” em cada capítulo aplicam os conceitos discutidos naquele capítulo ao estudo de caso. Nos Apêndices G, H e I, apresentamos uma implementação completa em Java do sistema orientado a objetos que projetamos nos capítulos anteriores.

Este estudo de caso vai ajudar a prepará-lo para os tipos de projetos substanciais encontrados nas empresas. Se você é um estudante e seu professor não planeja incluir este estudo de caso em seu curso, você pode querer cobri-lo por conta própria. Acreditamos que valerá a pena empregar seu tempo para percorrer este projeto grande e desafiador, porque o material apresentado nas seções do estudo de caso reforçam o material coberto nos capítulos correspondentes. Você experimentará uma introdução sólida ao projeto orientado a objetos com a UML. Você também irá aguçar sua habilidade de leitura de código passeando por um programa Java com 3594 linhas de código, escrito cuidadosamente e bem documentado, que resolve completamente o problema apresentado no estudo de caso.

Começamos nossa introdução à orientação a objetos com alguma terminologia fundamental. Para onde quer que você olhe no mundo real, você os vê – *objetos*! Pessoas, animais, plantas, carros, aviões, construções, computadores, etc. Seres humanos pensam em termos de objetos. Temos a maravilhosa habilidade da *abstração*, que nos permite visualizar imagens numa tela, como pessoas, aviões, árvores e montanhas, como objetos, em vez de ver pontos coloridos isolados (chamados *píxeis* – abreviatura de *picture elements*). Podemos, se desejarmos, pensar em termos de praias em vez de grãos de areia, florestas em vez de árvores e casas em vez de tijolos.

Poderíamos estar propensos a dividir os objetos em duas categorias – objetos animados e objetos inanimados. Objetos animados, em certo sentido, são “vivos”; eles se movem ao nosso redor e fazem coisas. Objetos inanimados, por outro lado, parecem não fazer mesmo muita coisa. Eles não se movem por si. Todos estes objetos, porém, têm algumas coisas em comum. Todos eles têm *atributos*, como tamanho, forma, cor e peso, e todos eles exibem *comportamentos* (p. ex.: a bola rola, salta, incha e esvazia; o bebê chora, dorme, engatinha, caminha e pisca; o carro acelera, freia e muda de direção; a toalha absorve água).

Os seres humanos aprendem sobre objetos estudando seus atributos e observando seus comportamentos. Objetos diferentes podem ter atributos semelhantes e podem exibir comportamentos semelhantes. Podem-se fazer comparações, por exemplo, entre bebês e adultos e entre seres humanos e chimpanzés. Carros, caminhões, pequenas caminhonetes vermelhas e patins de rodas têm muito em comum.

O *projeto orientado a objetos (OOD)* modela objetos do mundo real. Ele se aproveita das relações de *classe*, nas quais os objetos de uma certa classe – tal como uma classe de veículos – têm as mesmas características. Ela tira proveito de relações de *herança* e até de relações de *herança múltipla*,⁵ em que as classes de objetos recém-criadas são derivados absorvendo-se as características das classes existentes e adicionando-se características próprias. Um objeto da classe “conversível” certamente tem as características da classe mais genérica “automóvel”, mas, além disso, a capota de um conversível sobe e desce.

⁵ Aprenderemos mais tarde que, embora Java – ao contrário de C++ – não suporte herança múltipla, ela oferece a maioria dos benefícios fundamentais desta tecnologia através do suporte a múltiplas interfaces para uma classe.

O projeto orientado a objetos oferece uma maneira mais natural e intuitiva para visualizar o processo de projeto – a saber, *modelando* objetos do mundo real, seus atributos e seu comportamento. O OOD também modela a comunicação entre objetos. Da mesma maneira que as pessoas enviam *mensagens* umas às outras (por exemplo, o sargento que ordena ao soldado para se manter em posição de sentido), os objetos também se comunicam através de mensagens.

O OOD *encapsula* dados (atributos) e funções (comportamento) em *objetos*; os dados e as funções de um objeto estão intimamente amarrados. Os objetos têm a propriedade de *ocultação de informações*. Isto significa que, embora os objetos possam saber como se comunicar uns com os outros através de *interfaces* bem-definidas, normalmente não é permitido aos objetos saber como outros objetos são implementados – os detalhes de implementação ficam escondidos dentro dos próprios objetos. Certamente é possível dirigir um carro eficazmente sem conhecer os detalhes de como o motor, as transmissões e os sistemas de exaustão trabalham internamente. Veremos por que a ocultação de informações é tão crucial para a boa engenharia de *software*.

Linguagens como Java são *orientadas a objetos* – o ato de programar em uma linguagem como esta chama-se *programação orientada a objetos* (OOP – *object-oriented programming*) e permite aos projetistas implementar o projeto orientado a objetos como um sistema que funcione. Linguagens como C, por outro lado, são *linguagens de programação procedurais*, de modo que a programação tende a ser *orientada a ações*. Em C, a unidade de programação é a *função*. Em Java, a unidade de programação é a *classe*, a partir da qual, em algum momento, os objetos são *instanciados* (um termo elegante para “criados”). As classes de Java contêm *métodos* (que implementam os comportamentos da classe) e *atributos* (que implementam os dados da classe).

Os programadores de C se concentram em escrever funções. Os grupos de ações que executam alguma tarefa comum são reunidos para formar funções e as funções são agrupadas para formar programas. Certamente os dados são importantes em C, mas o ponto de vista é que os dados existem principalmente para suportar as ações que as funções executam. Os *verbos* em uma especificação de sistema ajudam o programador C a determinar o conjunto de funções necessárias para implementar aquele sistema.

Os programadores de Java se concentram em criar seus *próprios tipos definidos pelo usuário*, chamados *classes* e *componentes*. Cada classe contém dados e o conjunto de funções que manipulam aqueles dados. Os componentes de dados de uma classe são conhecidos como *atributos*. Os componentes funções de uma classe são conhecidos como *métodos*. Da mesma maneira que uma instância de um tipo primitivo da linguagem, como `int`, é chamada de *variável*, uma instância de um tipo definido pelo usuário (i.e., uma classe) é chamada de *objeto*. O programador usa tipos primitivos como blocos de construção para construir tipos definidos pelo usuário. O foco de atenção em Java está nas classes (com as quais criamos objetos) e não nas funções. Os *substantivos* em uma especificação de sistema ajudam o programador de Java a determinar o conjunto de classes a partir das quais serão criados objetos que irão trabalhar juntos para implementar o sistema.

As classes estão para os objetos assim como as plantas arquitetônicas estão para as casas. Podemos construir muitas casas a partir de uma planta e também podemos instanciar muitos objetos a partir de uma classe. As classes também podem ter relacionamentos com outras classes. Por exemplo, em um projeto orientado a objetos de um banco, a classe “caixa de banco” precisa se relacionar com a classe “cliente”. Estes relacionamentos são chamados de *associações*.

Veremos que, quando o *software* é empacotado como classes, estas classes podem ser usadas novamente (reusadas) em sistemas de *software* futuros. Grupos de classes relacionadas entre si são freqüentemente empacotadas como *componentes* reutilizáveis. Da mesma maneira que corretores de imóveis dizem a seu clientes que os três fatores mais importantes que afetam o preço dos imóveis são “localização, localização e localização”, muitas pessoas na comunidade de *software* acreditam que os três fatores mais importantes que afetam o futuro do desenvolvimento de *software* são “reutilização, reutilização e reutilização.”

Realmente, com a tecnologia de objetos, podemos construir a maior parte do *software* que vamos precisar combinando “peças padronizadas e intercambiáveis” chamadas classes. Este livro ensina como “elaborar classes valiosas” para reutilizar. Cada nova classe que você criar terá o potencial para se tornar um valioso patrimônio de *software* que você e outros programadores podem usar para acelerar e aumentar a qualidade de futuros trabalhos de desenvolvimento de *software* – uma possibilidade fascinante.

Introdução à análise e projeto orientados a objetos (OOAD)

Em breve você estará escrevendo programas em Java. Como você irá criar o código para seus programas? Se for como muitos programadores principiantes, você irá simplesmente ligar o computador e começar a digitar. Esta abordagem pode funcionar para projetos pequenos, mas o que você faria se fosse contratado para criar um sistema de

software para controlar as máquinas de caixa automáticas de um banco importante? Um projeto como este é muito grande e complexo para que você possa simplesmente sentar e sair digitando.

Para criar as melhores soluções, você deve seguir um processo detalhado para obter uma *análise dos requisitos* de seu projeto e desenvolver um *projeto* para satisfazer tais requisitos. Numa situação ideal, você passaria por este processo e teria seus resultados revisados e aprovados por seus superiores antes de escrever qualquer código para seu projeto. Se este processo envolve analisar e projetar seu sistema de um ponto de vista orientado a objetos, denomina-lo *processo de análise e projeto orientados a objetos (OOAD – object-oriented analysis and design)*. Programadores experientes sabem que, não importa quão simples um problema pareça ser, o tempo gasto em análise e projeto pode poupar incontáveis horas que poderiam ser perdidas ao se abandonar uma abordagem de desenvolvimento de sistema mal planejada, a meio caminho de sua implementação.

OOAD é o termo genérico para as idéias por trás do processo que empregamos para analisar um problema e desenvolver uma abordagem para resolvê-lo. Problemas pequenos como os discutidos nestes primeiros poucos capítulos não requerem um processo exaustivo. Pode ser suficiente escrever *pseudocódigo* antes de *começarmos* a escrever código. (*Pseudocódigo* é um meio informal de representar o código de um programa. Não é uma linguagem de programação de verdade, mas podemos usá-lo como uma espécie de “esboço” para nos guiar à medida que escrevemos o código. Apresentamos o pseudocódigo no Capítulo 4).

Pseudocódigo pode ser suficiente para problemas pequenos, mas, à medida que os problemas e os grupos de pessoas que resolvem estes problemas aumentam em tamanho, os métodos da OOAD são mais usados. Idealmente, um grupo deveria concordar quanto a um processo estritamente definido para resolver o problema e quanto a uma maneira uniforme de comunicar os resultados deste processo uns para os outros. Embora existam muitos processos diferentes de OOAD, uma linguagem gráfica única para informar os resultados de qualquer processo de OOAD se tornou largamente usada. Esta linguagem é conhecida como a *Unified Modeling Language (UML)*. A UML foi desenvolvida em meados da década de 90, sob a direção inicial de um trio de metodologistas de *software*: Grady Booch, James Rumbaugh e Ivar Jacobson.

História da UML

Na década de 80, um número crescente de organizações começou a usar a OOP para programar suas aplicações e surgiu a necessidade de um processo adequado para tratar da OOAD. Muitos metodologistas – incluindo Booch, Rumbaugh e Jacobson – produziram e promoveram individualmente processos separados para satisfazer esta necessidade. Cada um destes processos tinha sua própria notação, ou “linguagem” (sob a forma de diagramas gráficos), para comunicar os resultados da análise e projeto.

No início da década de 90, empresas diferentes, e até mesmo divisões diferentes de uma mesma empresa, usavam processos e notações distintos. Além disso, estas empresas queriam usar ferramentas de *software* que suportassem seus processos particulares. Com tantos processos, os vendedores de *software* achavam difícil fornecer tais ferramentas. Ficou claro que eram necessários processos e notação padronizados.

Em 1994, James Rumbaugh juntou-se a Grady Booch na Rational Software Corporation e os dois começaram a trabalhar para unificar seus populares processos. Em seguida, juntou-se a eles Ivar Jacobson. Em 1996, o grupo liberou versões preliminares da UML para a comunidade de engenharia de *software* e pediu realimentação. Mais ou menos na mesma época, uma organização conhecida como *Object Management Group™ (OMG™)* solicitou propostas para uma linguagem comum de modelagem. O OMG é uma organização sem fins lucrativos que promove o uso da tecnologia de orientação a objetos publicando diretrizes e especificações para tecnologias orientadas a objetos. Diversas corporações – entre elas HP, IBM, Microsoft, Oracle e Rational Software – já haviam reconhecido a necessidade de uma linguagem de modelagem comum. Estas empresas constituíram a *UML Partners* em resposta à solicitação de propostas do OMG. Este consórcio desenvolveu e enviou a versão 1.1 da UML para o OMG. O OMG aceitou a proposta e, em 1997, assumiu a responsabilidade pela manutenção e revisão continuadas da UML. Em 2001, o OMG liberou a versão 1.4 da UML (a versão atual por ocasião da publicação deste livro nos EUA) e está trabalhando na versão 2.0 (com liberação prevista para 2002).

O que é a UML?

A Unified Modeling Language é agora o esquema de representação gráfica mais amplamente utilizado para modelagem de sistemas orientados a objetos. Ela certamente unificou os diversos esquemas de notação mais populares. Aquelas que projetam sistemas usam a linguagem (sob a forma de diagramas gráficos) para modelar seus sistemas.

Uma característica atraente da UML é sua flexibilidade. A UML é extensível e independente dos muitos processos da OOAD. Os modeladores UML ficam livres para desenvolver sistemas com diversos processos, mas todos os desenvolvedores podem agora expressar tais sistemas com um conjunto-padrão de notações.

A UML é uma linguagem gráfica complexa e repleta de recursos. Em nossas seções “Pensando em objetos” apresentamos um subconjunto conciso e simplificado destes recursos. Usamos então este subconjunto para guiar o leitor através de uma primeira experiência de projeto com a UML, voltada para o programador/projetista principiante em orientação a objetos. Para uma discussão mais completa da UML, consulte o *site* do OMG na Web (www.omg.org) e o documento com as especificações oficiais da UML 1.4 (www.omg.org/uml). Além disso, foram publicados muitos livros sobre UML: *UML Distilled: Second Edition*, por Martin Fowler (com Kendall Scott) (ISBN #020165783X) oferece uma introdução detalhada à UML, com muitos exemplos. *The Unified Modeling Language User Guide* (ISBN #0201571684), escrito por Booch, Rumbaugh e Jacobson, é o tutorial definitivo para a UML. O leitor que procura um produto de aprendizado interativo pode pensar no *The Complete UML Training Course*, de Grady Brooch (ISBN #0130870145).

A tecnologia de orientação a objetos está em toda a parte no setor de *softwares* e a UML está rapidamente ficando assim. Nossa objetivo, nestas seções “Pensando em objetos”, é incentivá-lo a pensar de uma maneira orientada a objetos tão cedo e tão seguido quanto possível. Na seção “Pensando em objetos” no fim do Capítulo 2, você começará a aplicar a tecnologia de objetos para implementar a solução de um problema substancial. Esperamos que você ache este projeto opcional uma introdução agradável e desafiadora ao projeto orientado a objetos com a UML e à programação orientada a objetos.

1.16 Descobrindo padrões de projeto: introdução

Esta seção inicia o nosso tratamento de padrões de projeto, intitulado “Descobrindo padrões de projetos”. A maior parte dos exemplos fornecidos neste livro contêm menos de 150 linhas de código. Estes exemplos não requerem um processo de projeto extenso, porque só usam algumas poucas classes e ilustram conceitos de introdução à programação. Entretanto, alguns programas, como nosso estudo de caso opcional de simulação de um elevador, são mais complexos – eles podem exigir milhares de linhas de código ou até mais, conter muitas interações entre objetos e envolver muitas interações com o usuário. Sistemas maiores, como máquinas de caixa automáticas ou sistemas de controle de tráfego aéreo, poderiam conter milhões de linhas de código. Um projeto eficaz é crucial para a construção apropriada de tais sistemas complexos.

Ao longo da última década, o setor de engenharia de *software* fez progressos significativos no campo de *padrões de projeto* – arquiteturas comprovadas para construir *software* orientado a objetos flexível e fácil de manter⁶. O uso de padrões de projeto pode reduzir substancialmente a complexidade do processo de projetar. Projetar um sistema de ATMs é uma tarefa significativamente menos formidável se os desenvolvedores usam padrões de projeto. Além disso, o *software* orientado a objetos bem-projetado permite aos projetistas reutilizar e integrar componentes preexistentes em sistemas futuros. Os padrões de projeto beneficiam os desenvolvedores de sistemas:

- ajudando a construir *software* confiável com arquiteturas comprovadas e a experiência acumulada das empresas;
- promovendo a reutilização de projeto em sistemas futuros;
- ajudando a identificar erros e armadilhas comuns que ocorrem quando se constroem sistemas;
- ajudando a projetar sistemas de forma independente da linguagem na qual eles serão finalmente implementados;
- estabelecendo um vocabulário de projeto comum entre desenvolvedores;
- encurtando a fase de projeto em um processo de desenvolvimento de *software*.

O conceito de usar padrões de projeto para construir sistemas de *software* originou-se na área de arquitetura. Os arquitetos usam um conjunto de elementos arquiteturais estabelecidos, como arcos e colunas, quando projetam edi-

⁶ Gamma, Erich, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software*. (Massachussets: Addison-Wesley, 1995).

ficações. Projetar com arcos e colunas é uma estratégia comprovada para construir edificações seguras – estes elementos podem ser vistos como padrões para projetos arquitetônicos.

Em *software*, os padrões de projeto não são classes nem objetos. Em vez disso, os projetistas usam padrões de projeto para construir conjuntos de classes e objetos. Para usar padrões de projetos de maneira eficaz, os projetistas precisam se familiarizar com os padrões mais populares e eficazes usados no setor de engenharia de *software*. Neste capítulo, discutimos padrões de projeto e arquiteturas orientadas a objetos fundamentais, bem como sua importância na construção de *software* bem projetado.

Apresentamos diversos padrões de projeto em Java, mas estes padrões de projeto podem ser implementados em qualquer linguagem de programação orientada a objetos, como C++ ou Visual Basic. Descrevemos diversos padrões de projeto usados pela Sun Microsystems na Java API. Usamos padrões de projeto em muitos programas neste livro, os quais identificaremos ao longo de nossa discussão. Estes programas oferecem exemplos do uso de padrões de projeto para construir *softwares* orientados a objetos confiáveis e robustos.

História dos padrões de projeto orientados a objetos

No período de 1991 a 1994, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides – conhecidos como “a turma dos quatro” – usou sua experiência conjunta para escrever o livro *Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley: 1995). Este livro descreveu 23 padrões de projeto, cada um fornecendo a solução para um problema projeto de *software* comum na indústria. O livro agrupa padrões de projeto em três categorias – *padrões de criação de projeto*, *padrões estruturais de projeto* e *padrões comportamentais de projeto*. Os padrões de criação de projeto descrevem técnicas para instanciar objetos (ou grupos de objetos). Os padrões de projeto permitem organizar classes e objetos em estruturas maiores. Os padrões comportamentais estruturais de projeto atribuem responsabilidades a objetos.

A turma dos quatro mostrou que os padrões de projeto evoluíram naturalmente ao longo de anos de experiência da indústria. Em seu artigo *Seven Habits of Successful Pattern Writers*⁷, John Vlissides afirma que “a atividade isolada mais importante na escrita de padrões é a reflexão”. Esta afirmação implica que, para criar padrões, os desenvolvedores precisam refletir sobre seus sucessos (e erros) e documentá-los. Os desenvolvedores usam padrões de projeto para capturar e empregar esta experiência coletiva da indústria, o que, no final, os ajuda a não cometer o mesmo erro duas vezes.

Novos padrões de projeto estão sendo criados todo o tempo e sendo apresentados rapidamente para desenvolvedores no mundo inteiro através da Internet. O tópico de padrões de projeto é considerado, em geral, avançado, mas autores como nós estão incluindo este material em livros-textos introdutórios e de nível intermediário para tornar este importante conhecimento disponível para um público bem mais amplo.

Nossa abordagem de padrões de projeto começa com esta seção obrigatória no Capítulo 1 e continua com cinco seções opcionais “Descobrindo padrões de projeto” no final dos Capítulos 9, 13, 15, 17 e 21. Cada uma destas seções é colocada no final do capítulo que apresenta as tecnologias Java necessárias. Se você é um aluno e seu instrutor não planeja incluir este material em seu curso, incentivamo-lo a ler este material por sua própria conta.

1.17 Um passeio pelo livro

Você está para estudar uma das linguagens atuais de programação de computador mais animadoras e em rápida evolução. Dominar Java ajudará a desenvolver poderosos *softwares* aplicativos de computador pessoal e de comércio. Nesta seção, fazemos um passeio pelos muitos recursos de Java que você estudará em *Java Como Programar: Quarta Edição*.

Capítulo 1: Introdução aos computadores, à Internet e à Web – discute o que são computadores, como eles funcionam e como são programados. O capítulo dá um breve histórico do desenvolvimento de linguagens de programação, de linguagens de máquina, passando por linguagens simbólicas, até linguagens de alto nível. Discute-se a origem da linguagem de programação Java. O capítulo inclui uma introdução a um ambiente típico de programação Java. O capítulo também faz uma introdução à tecnologia de objetos, à Unified Modeling Language e a padrões de projeto.

Capítulo 2: Introdução a aplicativos Java – oferece uma introdução suave à programação de *aplicativos* na linguagem de programação Java. O capítulo apresenta a quem não é programador os conceitos e as construções básicas

⁷ Vlissides, John. *Pattern Hatching; Design Patterns Applied*. (Massachusetts: Addison-Wesley, 1998).

cos de programação. Os programas neste capítulo ilustram como exibir (também chamado de *enviar para a saída*) dados na tela para o usuário e como obter (também chamado de *dar entrada*) dados do usuário através do teclado. Algumas operações de entrada e saída são executadas com um componente de interface gráfica com o usuário (GUI – *graphical user interface*) chamado *JOptionPane*, que fornece janelas predefinidas (chamadas de caixas de diálogo) para entrada e saída. Isto permite que alguém sem experiência em programação se concentre em conceitos e construções de programação fundamentais em vez de no tratamento de eventos da GUI, que é mais complexo. Usá-lo aqui nos permite retardar nossa introdução ao tratamento de eventos da GUI para o Capítulo 6, “Métodos”. O Capítulo 2 também fornece abordagens detalhadas de operações aritméticas e de tomada de decisão. Após estudar este capítulo, o aluno irá entender como escrever aplicativos Java simples, mas completos.

Capítulo 3: Introdução a applets Java – apresenta um outro tipo de programa Java, chamado de *applet*. *Applets* são programas Java projetados para serem transportados pela Internet e executados em navegadores da World Wide Web (como o Netscape Navigator e o Microsoft Internet Explorer). O capítulo apresenta *applets* com vários dos *applets* de demonstração fornecidos com o Java 2 Development Kit (J2SDK). Usamos o *appletviewer* (um utilitário fornecido com o J2SDK) ou um navegador para executar diversos *applets* de exemplo. A seguir escrevemos *applets* Java que executam tarefas semelhantes às dos programas do Capítulo 2 e explicamos as semelhanças e diferenças entre *applets* e aplicativos. Após estudar este capítulo, o aluno irá entender como escrever *applets* Java simples, mas completos. Os capítulos seguintes usam tanto *applets* quanto aplicativos para demonstrar conceitos fundamentais de programação adicionais.

Capítulo 4: Estruturas de controle: parte 1 – focaliza o processo de desenvolvimento de programas. O capítulo discute como tomar uma *definição de problema* (isto é, um *documento de requisitos*) e, a partir dela, desenvolver um programa Java, incluindo executar etapas intermediárias em pseudocódigo. O capítulo apresenta alguns tipos de dados fundamentais e estruturas de controle simples utilizadas para tomada de decisão (*if* e *if/else*) e repetição (*while*). Examinamos a repetição controlada por contador e a repetição controlada por sentinelas e apresentamos os operadores Java de incremento, decremento e atribuição. O capítulo usa fluxogramas simples para mostrar o fluxo de controle através de cada uma das estruturas de controle. As técnicas discutidas nos Capítulos 2 a 7 constituem uma grande parte do que tem sido ensinado tradicionalmente nas universidades sob o tópico programação estruturada. Com Java, fazemos programação orientada a objetos. Fazendo isto, descobrimos que o interior dos objetos que construímos faz uso abundante de estruturas de controle. Tivemos uma experiência particularmente positiva aplicando os problemas 4.11 a 4.14 em nossos cursos de introdução. Como estes quatro problemas têm estrutura semelhante, fazer todos os quatro é uma maneira interessante do aluno “pegar o jeito” do processo de desenvolvimento de programas. Este capítulo ajuda o estudante a desenvolver bons hábitos de programação, como preparação para lidar com as tarefas de programação mais substanciais no restante do texto.

Capítulo 5: Estruturas de controle: parte 2 – continua as discussões das estruturas de controle de Java (*for*, a estrutura de seleção *switch* e a estrutura de repetição *do/while*). O capítulo explica os comandos *break* e *continue* com rótulo, usando exemplos de código ativo. O capítulo também contém uma discussão de operadores lógicos – **&&** (E lógico), **&** (E lógico booleano), **||** (OU lógico), **|** (OU lógico booleano inclusivo), **^** (OU lógico booleano exclusivo) e **!** (NÃO). Existe um conjunto substancial de exercícios, incluindo aplicações matemáticas e comerciais. Os estudantes vão gostar do exercício 5.25, que lhes pede para escrever um programa com estruturas de repetição e decisão que imprime a canção iterativa “*The Twelve Days of Christmas*”. Os alunos mais inclinados para a matemática gostarão dos problemas sobre os sistemas de numeração binário, octal, decimal e hexadecimal, cálculo da constante matemática π com uma série finita, triplas de Pitágoras e leis de De Morgan. Nossos alunos gostam particularmente dos desafios de imprimir triângulos e losangos nos exercícios 5.10, 5.18 e 5.20; estes problemas ajudam os alunos a aprender a lidar com estruturas de repetição aninhadas – um tópico complexo de ser dominado em cursos introdutórios.

Capítulo 6: Métodos – traz uma abordagem mais profunda de objetos. Os objetos contêm dados chamados de *variáveis de instância* e unidades executáveis chamadas *métodos* (estes são freqüentemente chamados de *funções* nas linguagens de programação procedurais, não orientadas a objetos, como C e *funções-membro* em C++). Exploramos os métodos em profundidade e incluímos uma discussão de métodos que “chamam a si mesmos”, chamados de métodos recursivos. Discutimos métodos de bibliotecas de classes, métodos definidos pelo programador e recursividade. As técnicas apresentadas no Capítulo 6 são essenciais para a produção de programas adequadamente estruturados, especialmente os tipos de programas e *software* maiores que os programadores de sistemas e de aplicações provavelmente irão desenvolver nas aplicações do mundo real. A estratégia de “dividir para conquistar” é apresentada como um meio eficaz de resolver problemas complexos dividindo-os em componentes mais simples que interagem. Os alunos gostam da abordagem de números randômicos e simulação e gostam da discussão do jogo de da-

dos *craps*, que faz um uso elegante de estruturas de controle (esta é uma de nossas aulas de mais sucesso em nossos cursos de introdução). O capítulo oferece uma introdução sólida à recursividade e inclui uma tabela que resume as dezenas de exemplos e exercícios sobre recursividade distribuídos ao longo do restante do livro. Alguns textos deixam a recursividade para um capítulo mais adiante no livro; achamos que este tópico é abordado melhor de forma gradual ao longo do texto. O tópico de sobrecarga de métodos (i.e., permitir que diversos métodos tenham o mesmo nome, desde que tenham “assinaturas” diferentes) é motivado e explicado claramente. Apresentamos *eventos* e *tratamento de eventos* – elementos necessários para programar interfaces gráficas com o usuário. Os eventos são avisos sobre mudanças de estado, como o clique de um botão, cliques do mouse e pressionamento de uma tecla. Java permite que os programadores especifiquem as respostas a eventos codificando métodos chamados de tratadores de eventos. A extensa coleção de exercícios no fim do capítulo inclui diversos problemas clássicos de recursividade como as Torres de Hanói; voltamos a este problema mais tarde no texto, quando usamos gráficos, animação e som para fazer o problema “criar vida”. Existem muitos exemplos matemáticos e gráficos. Nossos alunos gostam em particular do desenvolvimento de um sistema de “instrução assistida por computador” nos exercícios 6.31 a 6.33; pedimos aos alunos para desenvolver uma versão em multimídia deste sistema mais tarde no livro. Os alunos irão gostar do desafio dos “programas misteriosos”. Os alunos mais inclinados para matemática vão gostar dos problemas sobre números perfeitos, máximo divisor comum, números primos e fatoriais.

Capítulo 7: Arrays – explora o processamento de dados em listas e tabelas de valores. Os *arrays* em Java são processados como objetos, mais evidência do compromisso de Java com quase 100% de orientação a objetos. Discutimos a estruturação de dados em *arrays*, ou grupos, de itens de dados relacionados do mesmo tipo. O capítulo apresenta inúmeros exemplos tanto de *arrays* de um subscrito quanto *arrays* de dois subscritos. É amplamente reconhecido que estruturar os dados apropriadamente é tão importante quanto usar estruturas de controle de forma eficaz no desenvolvimento de programas estruturados adequadamente. Os exemplos no capítulo investigam várias manipulações comuns de *arrays*, impressão de histogramas, ordenação de dados, passagem de *arrays* para métodos e uma introdução à área de análise de dados de pesquisas (com estatísticas simples). Um recurso deste capítulo é a discussão de técnicas elementares de ordenação e pesquisa e a apresentação da pesquisa binária como uma melhoria significativa em relação à pesquisa linear. Os exercícios do fim do capítulo incluem uma variedade de problemas interessantes e desafiadores, como técnicas de ordenação melhoradas, o projeto de um sistema de reserva de companhia aérea e uma introdução ao conceito de gráficos de tartaruga (tornados famosos na linguagem de programação LOGO) e os problemas do Passeio do Cavalo e das Oito Rainhas, que apresentam as noções de programação heurística tão largamente usadas na área de inteligência artificial. Os exercícios terminam com uma série de problemas de recursividade, que incluem a ordenação por seleção, palíndromos, pesquisa linear, pesquisa binária, as oito rainhas, imprimir um *array*, imprimir um *string* de trás para diante e encontrar o valor mínimo em um *array*. Os exercícios do capítulo incluem uma deliciosa simulação da clássica corrida entre a tartaruga e a lebre, algoritmos para embaralhar e distribuir cartas, ordenação recursiva com o método *quicksort* e percorrer um labirinto de forma recursiva. Uma seção especial, denominada “Construindo seu próprio computador”, explica a programação em linguagem de máquina e continua com o projeto e a implementação de um simulador de computador que permite ao leitor escrever e executar programas em linguagem de máquina. Este recurso incomparável do texto será especialmente útil para o leitor que queira entender como os computadores realmente funcionam. Nossos alunos apreciam este projeto e freqüentemente implementam melhorias substanciais; muitas melhorias são sugeridas nos exercícios. No Capítulo 19, uma outra seção especial orienta o aluno na construção de um compilador; a linguagem de máquina produzida pelo compilador é então executada no simulador de linguagem de máquina produzido no Capítulo 7. As informações são passadas do compilador para o simulador em arquivos seqüenciais (apresentados no Capítulo 16).

Capítulo 8: Programação baseada em objetos – inicia nossa discussão mais profunda de classes. O capítulo representa uma maravilhosa oportunidade para ensinar a abstração de dados da “maneira correta” – através de uma linguagem (Java) expressamente dedicada a implementar tipos de dados abstratos (ADTs). O capítulo focaliza a essência e a terminologia de classes e objetos. O que é um objeto? O que é uma classe de objetos? Qual é a aparência interna de um objeto? Como os objetos são criados? Como são destruídos? Como os objetos se comunicam entre si? Por que as classes são um mecanismo tão natural para empacotar *software* como componentes reutilizáveis? O capítulo discute a implementação de ADTs como classes no estilo de Java, acesso a membros de classes, garantia de ocultação de informações com variáveis de instância **private**, separação de interface da implementação, uso de métodos de acesso e métodos utilitários e inicialização de objetos com construtores (e o uso de construtores sobre-carregados). O capítulo discute a declaração e o uso de referências, *composição* – o processo de construir classes que têm como membros referências para objetos, a referência **this**, que permite a um objeto “conhecer a si mesmo”, alocação dinâmica de memória, membros de classe **static** para armazenar e manipular dados comuns a todos.

da a classe e exemplos de tipos de dados abstratos populares, como pilhas e filas. O capítulo apresenta a declaração **package** e discute como criar pacotes reutilizáveis. O capítulo também apresenta a criação de arquivos JAR (*Java archive*) e demonstra como utilizar arquivos JAR para compor *applets* que consistem de diversas classes. Os exercícios do capítulo desafiam o aluno a desenvolver classes para números complexos, números racionais, horas, datas, retângulos, inteiros enormes, uma classe para jogar o jogo da velha, uma classe para conta de poupança e uma classe para armazenar conjuntos de inteiros.

Capítulo 9: Programação orientada a objetos – discute os relacionamentos entre classes de objetos e programação com classes relacionadas. Como podemos explorar o que há de comum entre classes de objetos para minimizar a quantidade de trabalho necessário para construir grandes sistemas de *software*? O que é polimorfismo? O que significa “programação genérica” em oposição a “programação específica”? Como a programação genérica torna fácil modificar sistemas e adicionar novos recursos com esforço mínimo? Como podemos programar para uma categoria inteira de objetos em vez de programar individualmente para cada tipo de objeto? O capítulo lida com um dos recursos mais fundamentais das linguagens de programação orientadas a objetos, herança, que é uma forma de reutilização de *software* na qual novas classes são desenvolvidas rápida e facilmente, absorvendo os recursos das classes existentes e adicionando novos recursos apropriados. O capítulo discute os conceitos de superclasses e subclasses, membros **protected**, superclasses diretas, superclasses indiretas, uso de construtores em superclasses e subclasses e engenharia de *software* com herança. Este capítulo apresenta classes internas, que ajudam a ocultar detalhes de implementação. Classes internas são usadas mais frequentemente para criar tratadores de eventos de GUI. Classes internas com nomes podem ser declaradas dentro de outras classes e são úteis na definição de tratadores de eventos comuns para diversos componentes de GUI. Classes internas anônimas são declaradas dentro de métodos e são usadas para criar um objeto – normalmente um tratador de eventos para um componente GUI específico. O capítulo compara herança (relacionamentos “é um”) com composição (relacionamentos “tem um”). Diversos estudos de casos substanciais são usados como recurso no capítulo. Em particular, um longo estudo de caso implementa uma hierarquia de classes ponto, círculo e cilindro. Os exercícios pedem ao aluno para comparar a criação de novas classes por herança *versus* por composição; para estender as hierarquias de herança discutidas no capítulo; para escrever uma hierarquia de herança para quadriláteros, trapezóides, paralelogramos, retângulos e quadrados e para criar uma hierarquia de formas mais genérica com formas bidimensionais e tridimensionais. O capítulo explica o comportamento polimórfico. Quando muitas classes estão relacionadas através de herança a uma superclasse comum, cada objeto de subclasse pode ser tratado como um objeto da superclasse. Isto permite que os programas sejam escritos de uma maneira genérica independente dos tipos específicos dos objetos da subclasse. Novos tipos de objetos podem ser manipulados pelo mesmo programa, desta forma tornando os sistemas mais fáceis de estender. O polimorfismo permite eliminar lógica complexa do tipo **switch**, em favor de lógica “linear” mais simples. Um gerenciador de tela de vídeo game, por exemplo, pode enviar uma mensagem “desenhar” para todos os objetos em uma lista encadeada de objetos a serem desenhados. Cada objeto sabe como desenhar a si mesmo. Um novo tipo de objeto pode ser adicionado ao programa sem modificar aquele programa, desde que o novo objeto também saiba como desenhar a si mesmo. Este estilo de programação é usado geralmente para implementar as interfaces gráficas com o usuário populares hoje em dia. O capítulo diferencia classes **abstract** (das quais os objetos *não podem* ser instanciados) de classes concretas (das quais os objetos *podem* ser instanciados). O capítulo também apresenta interfaces – conjuntos de métodos que devem ser definidos por qualquer classe que implemente a interface. As interfaces são o substituto de Java para o perigoso (embora poderoso) recurso de C++ chamado de herança múltipla.

As classes abstratas são úteis para fornecer um conjunto básico de métodos e implementação *default* para classes ao longo de toda a hierarquia. Interfaces são úteis em muitas situações semelhantes às classes abstratas; entretanto, interfaces não incluem nenhuma implementação – interfaces não têm nenhum corpo de método nem variáveis de instância. Três estudos de caso importantes com polimorfismo são usados como recurso no capítulo – um sistema de folha de pagamento, uma hierarquia de formas derivada de uma classe **abstract** e uma hierarquia de formas que implementa uma interface. Os exercícios do capítulo pedem ao aluno para discutir diversos aspectos e abordagens conceituais, trabalhar com classes **abstract**, desenvolver um pacote básico de gráficos, modificar a classe **Employee** do capítulo e procurar resolver todos estes projetos com programação polimórfica.

Capítulo 10: Strings e caracteres – trata do processamento de palavras, frases, caracteres e grupos de caracteres. A diferença-chave entre Java e C aqui é que em Java *strings* são objetos, tornando, assim, a manipulação de *strings* mais conveniente e muito mais segura do que em C, em que a manipulação de *strings* e *arrays* é baseada nos perigosos *ponteiros*. Apresentamos as classes **String**, **StringBuffer**, **Character** e **StringTokenizer**. Para cada uma, fornecemos longos exemplos com código ativo, demonstrando a maior parte de seus métodos “em ação”. Em todos os casos, mostramos janelas de saída de modo que o leitor possa ver os efeitos exatos de cada uma

das manipulações de *strings* e caracteres. Os alunos gostarão do exemplo de embaralhar e distribuir cartas (que eles vão melhorar nos exercícios dos capítulos posteriores sobre gráficos e multimídia). Uma característica importante do capítulo é uma extensa coleção de exercícios de manipulação de *strings* relacionados a *limericks*, *pig Latin*, análise de texto, processamento de texto, impressão de datas em vários formatos, proteção de cheques, preenchimento por extenso do valor de um cheque, código Morse e conversão do sistema métrico para o sistema inglês. Os alunos vão gostar dos desafios de desenvolver seu próprio verificador ortográfico e gerador de palavras cruzadas.

Tópicos avançados

Os Capítulos 11, 12 e 13 foram escritos em co-autoria com o Sr. Tem Nieto da Deitel & Associates, Inc. A infinita paciência, atenção aos detalhes, habilidades de ilustração e criatividade de Tem são visíveis ao longo destes capítulos. [Dê uma rápida olhada na Fig. 12.19 para ver o que acontece quando deixamos Tem solto!]

Capítulo 11: Imagens gráficas e Java2D – é o primeiro de diversos capítulos que apresentamos o “lado” multimídia de Java. Consideramos os Capítulos 11 a 22 o material avançado do livro. É “pura diversão”. A programação tradicional em C e C++ é muito limitada à entrada/saída no modo de caractere. Algumas versões de C++ são suportadas por bibliotecas de classes dependentes de plataforma que podem trabalhar graficamente, mas utilizar essas bibliotecas deixa seus aplicativos não portáveis. Os recursos gráficos de Java são independentes de plataforma e, consequentemente, portáveis – e queremos dizer portáveis no sentido mundial. Você pode desenvolver *applets* Java que usam intensamente gráficos e distribuí-los através da World Wide Web para colegas em todos os lugares e eles irão funcionar perfeitamente nas plataformas Java locais. Discutimos contextos gráficos e objetos gráficos; desenhar *strings*, caracteres e *bytes*; controle de cores e fontes de caracteres; manipulação da tela e modos de pintura e desenhar linhas, retângulos, retângulos com cantos arredondados, retângulos tridimensionais, elipses, arcos e polígonos. Apresentamos a Java2D API, que oferece ferramentas de manipulação de gráficos poderosas. A Figura 11.22 é um exemplo da facilidade de se usar a Java2D API para criar efeitos gráficos complexos como texturas e gradientes. O capítulo tem 23 figuras que ilustram exaustivamente cada um destes recursos gráficos com exemplos em código ativo, saídas na tela convincentes, tabelas de características detalhadas e esboços gráficos. Alguns dos 27 exercícios desafiam os estudantes a desenvolver versões gráficas de suas soluções para exercícios anteriores sobre gráficos de tartaruga, passeio do cavalo, simulação da tartaruga e a lebre, percorrer um labirinto e a ordenação pelo método *Bucket Sort*. Nossa livro complementar, *Advanced Java 2 Platform How to Program*, apresenta a Java 3D API.

Capítulo 12: Componentes da interface gráfica com o usuário: parte 1 – introduz a criação de *applets* e aplicativos com interfaces gráficas com o usuário (GUIs) amigáveis. Este capítulo se concentra nos *componentes da GUI Swing* de Java. Estes componentes, *independentes de plataforma*, são inteiramente escritos em Java. Isso confere aos componentes da GUI Swing uma enorme flexibilidade – os componentes GUI podem ser personalizados para terem a aparência de componentes de interface com o usuário da plataforma de computador no qual o programa é executado ou eles podem utilizar a aparência e o comportamento padrão de Java que fornece uma interface de usuário idêntica em todas as plataformas de computador. O desenvolvimento de GUI é um tópico muito longo, de modo que o dividimos em dois capítulos. Estes capítulos cobrem o material em profundidade suficiente para permitir que você construa interfaces GUI com “nível profissional”. Discutimos o pacote `javax.swing`, que oferece componentes GUI muito mais poderosos do que os componentes `java.awt` que tiveram origem em Java 1.0. Através de seus 16 programas e muitas tabelas e desenhos, o capítulo ilustra os princípios de GUI, a hierarquia `javax.swing`, rótulos, botões de acionamento, listas, campos de texto, caixas combinadas, caixas de marcação, botões de rádio, painéis, tratamento de eventos do mouse, tratamento de eventos do teclado e o uso de três dos gerenciadores de layout mais simples de Java, a saber `FlowLayout`, `BorderLayout` e `GridLayout`. O capítulo se concentra no modelo de delegação de eventos para o processamento de GUI. Os 33 exercícios desafiam o aluno a criar GUIs específicas, exercitam várias características de GUIs, desenvolvem programas para desenhar que deixam o usuário desenhar usando o mouse e controlar fontes de caracteres.

Capítulo 13: Componentes da interface gráfica com o usuário: parte 2 – continua a discussão detalhada de Swing iniciada no Capítulo 12. Através de seus 13 programas, bem como tabelas e desenhos, o capítulo ilustra os princípios de projeto de GUIs, a hierarquia `javax.swing`, áreas de texto, criação de subclasses de componentes Swing, controles deslizantes, janelas, menus, menus escamoteáveis, alteração da aparência e do comportamento e o uso de três gerenciadores de layout de GUI avançados, a saber: `BoxLayout`, `CardLayout` e `GridBagLayout`. Dois dos exemplos mais importantes apresentados neste capítulo são um programa que pode ser executado tanto como um *applet* quanto como um aplicativo e um programa que demonstra como criar uma interface gráfica com o usuário do tipo *multiple document interface (MDI)*. A MDI é uma interface gráfica com o usuário complexa, na qual uma janela – chamada *mãe* – atua como a janela de controle para a aplicação. Esta janela-mãe contém uma ou mais

janelas-filhas – que sempre são exibidas graficamente dentro da janela-mãe. A maioria dos processadores de texto usam interfaces gráficas com o usuário MDI. O capítulo conclui com uma série de exercícios que incentivam o leitor a desenvolver GUIs substanciais com as técnicas e os componentes apresentados no capítulo. Um dos exercícios mais desafiantes neste capítulo é uma aplicação de desenho completa que pede ao leitor para criar um programa orientado a objetos que mantém controle das formas que o usuário desenhou. Outros exercícios usam herança para criar subclasses de componentes Swing e reforçam os conceitos de gerenciamento de layout. Os seis primeiros capítulos de nosso livro complementar, *Advanced Java 2 Platform How to Program*, são projetados para cursos sobre programação avançada de GUIs.

Capítulo 14: Tratamento de exceções – é um dos capítulos mais importantes do livro do ponto de vista da construção dos chamados aplicativos de “missão crítica” ou aplicativos para “negócios críticos” que exigem alto grau de robustez e tolerância a falhas. As coisas dão errado e, nas velocidades dos computadores atuais – comumente centenas de milhões de operações por segundo (com os computadores pessoais mais recentes executando um bilhão ou mais de instruções por segundo) – se elas podem dar errado, darão e muito rapidamente. Frequentemente, os programadores são um pouco ingênuos quanto à utilização de componentes. Eles perguntam: “Como eu peço que um componente faça algo para mim?”. Eles também perguntam: “Que valor(es) esse componente me retorna para indicar que realizou o trabalho que pedi para ele fazer?”. Mas os programadores também precisam se preocupar com: “O que acontece quando o componente que chamo para fazer um trabalho encontra dificuldade? Como esse componente sinalizará que teve um problema?”. Em Java, quando um componente (p. ex., um objeto de classe) encontra dificuldades, ele pode “disparar uma exceção”. O ambiente desse componente é programado para “capturar” essa exceção e tratá-la. Os recursos de tratamento de exceções de Java são projetados para um mundo orientado a objetos em que os programadores constroem sistemas em grande parte a partir de componentes reutilizáveis pré-fabricados, construídos por outros programadores. Para utilizar um componente Java, você precisa saber não apenas como esse componente se comporta quando as “coisas vão bem”, mas também que exceções esse componente dispara quando as “coisas dão errado”. O capítulo faz a distinção entre erros de sistema bastante sérios (normalmente fora do controle da maioria dos programas) e exceções (com as quais os programas geralmente podem lidar para assegurar operação robusta). O capítulo discute o vocabulário do tratamento de exceções. O bloco `try` executa código do programa que ou é executado adequadamente ou dispara uma exceção se algo der errado. Associados a cada bloco `try` existem um ou mais blocos `catch` que tratam as exceções disparadas, numa tentativa de restaurar a ordem e manter os sistemas “em execução” em vez de deixar que eles entrem em “colapso”. Mesmo se a ordem não pode ser totalmente restabelecida, os blocos `catch` podem executar operações que permitem que um sistema continue sendo executado, embora em nível de desempenho reduzido – tal atividade é frequentemente chamada de “degradação elegante”. Quer tenham sido disparadas exceções ou não, o bloco `finally` que acompanha o bloco `try` vai ser sempre executado; o bloco `finally` normalmente executa operações de limpeza como fechar arquivos e liberar recursos adquiridos no bloco `try`. O material neste capítulo é crucial para muitos dos exemplos de código ativo no restante do livro. O capítulo enumera muitos dos erros e exceções dos pacotes de Java. O capítulo tem algumas das citações mais apropriadas do livro, graças à pesquisa incansável de Barbara Deitel. A grande maioria das Dicas de teste e depuração do livro derivaram naturalmente do material no Capítulo 14.

Capítulo 15: Multithreading – trata da programação de *applets* e aplicativos que podem realizar múltiplas atividades em paralelo. Embora o nosso organismo seja muito bom nisso (respiração, alimentação, circulação de sangue, visão, audição, etc., todos podem ocorrer em paralelo), nossa mente tem problemas com isso. Os computadores costumavam ser construídos com um único processador bastante caro. Hoje os processadores estão tornando-se tão baratos que é possível construir computadores com muitos processadores que trabalham em paralelo – esses computadores são chamados de *multiprocessadores*. Há uma clara tendência em direção a computadores que podem realizar muitas tarefas em paralelo. A maioria das linguagens de programação atuais, incluindo C e C++, não inclui recursos embutidos para expressar operações paralelas. Essas linguagens são frequentemente conhecidas como linguagens de programação “sequenciais” ou como linguagens de “uma única *thread* de controle”. Java inclui recursos que permitem aplicativos *multithreaded*, ou multiescalonados, isto é, aplicativos que podem especificar que múltiplas atividades deverão ocorrer em paralelo. Isso deixa Java mais bem preparada para lidar com aplicativos multimídia mais sofisticados, aplicativos baseados em multiprocessador e baseados em rede que os programadores desenvolverão. Como veremos, o *multithreading* é eficaz até mesmo em sistemas de um único processador. Por muitos anos, o “velho” ministrou cursos sobre sistemas operacionais e escreveu livros sobre sistemas operacionais, mas ele nunca teve uma linguagem que suportasse *multithreading*, como Java, disponível para demonstrar os conceitos. Neste capítulo, ficamos satisfeitos em apresentar programas com múltiplas *threads*, que demonstram claramente os tipos de problemas que podem ocorrer em programação paralela. Existem todos os tipos de sutilezas que surgem em

programas paralelos sobre as quais você simplesmente nunca pensa quando escreve programas seqüenciais. Um dos recursos usado no capítulo é o extenso conjunto de exemplos que mostram estes problemas e como resolvê-los. Um outro recurso é a implementação do “*buffer circular*”, um meio popular de coordenar o controle entre processos assíncronos concorrentes do tipo “produtor” e “consumidor”, que, se fosse deixado em execução sem sincronização, provocaria a perda ou duplicação incorreta de dados, freqüentemente com resultados catastróficos. Discutimos a construção de monitor desenvolvida por C. A. R. Hoare e implementada em Java; este é um tópico-padrão nos cursos sobre sistemas operacionais. O capítulo discute *threads* e métodos de *threads*. Ele percorre os vários estados e transições de estado de *threads* com um desenho detalhado que mostra o ciclo de vida de uma *thread*. Discutimos as prioridades das *threads* e o escalonamento de *threads* e usamos um desenho para mostrar o mecanismo de escalonamento com prioridade fixa de Java. Examinamos um relacionamento produtor/consumidor sem sincronização, observamos os problemas que ocorrem e resolvemos os problemas com a sincronização de *threads*. Implementamos um relacionamento produtor/consumidor com um *buffer circular* e sincronização adequada com um monitor. Discutimos *threads* do tipo *daemon*, que “ficam por perto” e executam tarefas (p. ex., “coleta de lixo”) quando o tempo do processador está disponível. Discutimos a interface `Runnable`, que permite aos objetos serem executados como *threads* sem ser derivados como subclasses de `Thread`. Terminamos com uma discussão de grupos de *threads* que, por exemplo, permitem que seja garantida a separação entre *threads* como o coletor de lixo e *threads* do usuário. O capítulo tem um bom complemento de exercícios. O exercício em destaque é o clássicos problema dos leitores e escritores, um dos favoritos em cursos de mais alto nível sobre sistemas operacionais; nos exercícios aparecem citações para os alunos que desejarem pesquisar sobre este assunto. Trata-se de um problema importante em sistemas de processamento de transações orientados para bancos de dados. Ele levanta aspectos sutis da solução de problemas de controle de concorrência ao mesmo tempo em que se assegura que cada atividade separada que precisa ser atendida o é, sem a possibilidade de “postergação indefinida”, que poderia fazer com que algumas atividades nunca fossem atendidas – uma condição também chamada de “inanição”. Os professores de sistemas operacionais irão gostar dos projetos implementados por estudantes proficientes em Java. Podemos esperar progresso substancial no campo da programação paralela à medida que Java permite que um número maior de alunos de computação implemente projetos de aula com programação paralela. À medida que estes alunos ingressarem no mercado de trabalho, ao longo dos próximos anos, esperamos um aumento significativo na programação de sistemas paralelos e programação de aplicações paralelas. Temos previsto isto há muitas décadas – Java está tornando isto uma realidade.

Se este é seu primeiro livro sobre Java e você é um profissional de computação com experiência, você pode muito bem estar pensando: “Puxa, isto está ficando cada vez melhor. Não posso esperar para começar a programar nesta linguagem. Ele vai me permitir fazer todos os tipos de coisas que eu sempre quis fazer, mas que nunca foi fácil para mim fazer com as outras linguagens que usei”. Você tem razão. Java é uma facilitadora. Portanto, se você gostou da discussão sobre *multithreading*, se prepare, pois Java vai permitir que você programe aplicações com multimídia e as torne disponíveis instantaneamente através da World Wide Web.

Capítulo 16: Arquivos e fluxos – trata da entrada/saída que é realizada por fluxos de dados dirigidos de/para arquivos. Esse é um dos capítulos mais importantes para os programadores que desenvolverão aplicativos comerciais. Os negócios modernos são centrados em torno de dados. Neste capítulo, traduzimos dados (objetos) para um formato persistente que pode ser usado por outras aplicações. Ter a capacidade de armazenar dados em arquivos ou movê-los através de redes (Capítulo 17) torna possível para os programas salvar dados e se comunicar uns com os outros. Este é o verdadeiro ponto forte do *software* hoje em dia. O capítulo começa com uma introdução à hierarquia de dados, de *bits* a *bytes*, campos, registros, até arquivos. A seguir, é apresentada a visão simples de Java sobre arquivos e fluxos. Apresentamos então um passeio pelas dezenas de classes da extensa hierarquia de classes de arquivos e fluxos de entrada/saída de Java. Colocamos em ação muitas destas classes em exemplos de código ativo neste capítulo e no Capítulo 17. Mostramos como os programas passam dados para dispositivos de armazenamento secundário, como discos, e como os programas recuperam dados já armazenados naqueles dispositivos. São discutidos os arquivos de acesso seqüencial com uma série de três programas que mostram como abrir e fechar arquivos, como armazenar dados em um arquivo seqüencialmente e como ler dados seqüencialmente de um arquivo. São discutidos os arquivos de acesso aleatório com uma série de quatro programas que mostram como criar seqüencialmente um arquivo para acesso aleatório, como ler e escrever dados em um arquivo com acesso aleatório e como ler dados seqüencialmente de um arquivo acessado aleatoriamente. O quarto programa de acesso aleatório combina muitas das técnicas de acesso tanto a arquivos seqüenciais quanto aleatórios em um programa completo de processamento de transações. Discutimos a utilização de *buffers* e como ela ajuda os programas que fazem quantidades significativas de entrada/saída a ter melhor desempenho. Discutimos a classe `File`, que é usada pelos programas para obter diversas informações sobre arquivos e diretórios. explicamos como objetos podem ser escritos e lidos de dis-

positivos de armazenamento secundário. Os estudantes em nossos seminários para empresas nos disseram que, depois de estudar o material sobre processamento de arquivos, eles foram capazes de produzir programas de processamento de arquivos substanciais que tiveram utilidade imediata para suas organizações. Os exercícios pedem ao aluno para implementar diversos programas que constroem e processam arquivos de acesso seqüencial e arquivos de acesso aleatório.

Capítulo 17: Redes – lida com *applets* e aplicativos que podem se comunicar através de redes. Este capítulo apresenta os recursos de Java para redes de mais baixo nível. Escrevemos programas que “andam pela Web”. Os exemplos do capítulo ilustram um *applet* interagindo com o navegador no qual está sendo executado, a criação de um mininavegador da Web, a comunicação entre dois programas Java com soquetes baseados em fluxos e a comunicação entre dois programas Java usando pacotes de dados. Um destaque do capítulo é a implementação em código ativo de um jogo da velha cooperativo cliente/servidor no qual dois clientes jogam um com o outro arbitrados por um servidor *multithreaded* – coisa muito boa! A arquitetura do servidor *multithreaded* é exatamente o que é usado atualmente em servidores de rede UNIX e Windows NT populares. O exemplo principal no capítulo é o estudo de caso do Deitel Messenger, que simula muitas das aplicações de mensagens instantâneas de hoje em dia que permitem aos usuários de computadores se comunicar com amigos, parentes e colegas de trabalho através da Internet. Este estudo de caso cliente/servidor *multithreaded*, com 1130 linhas, usa a maior parte das técnicas de programação apresentadas no livro até este ponto. A aplicação Messenger também apresenta a transmissão múltipla, que permite a um programa enviar pacotes de dados para grupos de clientes. O capítulo tem uma bela coleção de exercícios, incluindo diversas sugestões de modificações para o exemplo de servidor *multithreaded*. Nossa livro complementar, *Advanced Java 2 Platform How to Program*, oferece um tratamento bem mais aprofundado de computação através de redes e distribuída, com tópicos que incluem a invocação remota de métodos (RMI), *servlets*, JavaServer Pages (JSP), computação, Java sem-fio (e a Java 2 Micro Edition) e a Common Object Request Broker Architecture (CORBA).

Capítulo 18: Multimídia: imagens, animação, áudio e vídeo – é o primeiro de dois capítulos que apresentam os recursos de Java para fazer os aplicativos de computadores “ganhar vida” (o Capítulo 22 oferece uma extensa abordagem do Java Media Framework). É incrível que os alunos em cursos de programação básicos estejam escrevendo aplicativos com todos esses recursos. As possibilidades são fascinantes. Imagine ter acesso (através da Internet e tecnologia de CD-ROM) a vastas bibliotecas de imagens gráficas, áudio e vídeos e ser capaz de juntar os seus próprios com aqueles das bibliotecas para formar aplicativos criativos. A maioria dos computadores novos vendidos já vêm “equipados com multimídia”. Os alunos podem criar trabalhos de conclusão de semestre e apresentações para fazer em aula com componentes retirados de enormes bibliotecas de imagens, desenhos, vozes, fotografias, vídeos, animações e coisas parecidas, de domínio público. Quando muitos de nós estávamos nos primeiros anos de faculdade, um “artigo” era uma coleção de caracteres, às vezes escritos à mão, às vezes datilografados. Hoje em dia, um “artigo” pode ser uma “extravagância” em multimídia que faz o assunto abordado criar vida. Ele pode prender sua atenção, despertar sua curiosidade e fazer você sentir o que os personagens do artigo sentiram quando estavam fazendo história. A multimídia está tornando os laboratórios de ciências muito mais excitantes. Os livros estão adquirindo vida. Em vez de olhar para uma imagem estática de algum fenômeno, você pode observar aquele fenômeno acontecer em uma apresentação colorida e animada, com sons, vídeos e vários outros efeitos, alavancando o processo de aprendizado. As pessoas são capazes de aprender mais, aprender em mais profundidade e experimentar mais pontos de vista.

O capítulo discute imagens e manipulação de imagens, áudio e animação. Um destaque do capítulo são os mapas em imagens, que permitem a um programador detectar a presença do mouse sobre uma região de uma imagem, sem que o mouse seja clicado. Apresentamos uma aplicação de mapa em imagem com código ativo com os ícones das dicas de programação que você viu neste capítulo e vai ver ao longo de todo o livro. À medida que o usuário move o ponteiro do mouse sobre as sete imagens dos ícones, o tipo de dica é exibido, seja “Boas práticas de programação” para o ícone do polegar para cima, “Dica de portabilidade” para o ícone do inseto com a mala, e assim por diante. Depois de ter lido o capítulo, você estará ansioso para experimentar todas estas técnicas, de modo que incluímos 35 problemas para desafiá-lo e diverti-lo (outros são fornecidos no Capítulo 22). Aqui estão alguns dos exercícios que você pode querer transformar em projetos de conclusão de semestre:

<i>Ampliador de imagem</i>	<i>Caça-níqueis</i>	<i>Designer de fogos de artifício</i>
<i>Animação</i>	<i>Caleidoscópio</i>	<i>Flasher de imagem</i>
<i>Apagar aleatoriamente uma imagem</i>	<i>Chamando a atenção para imagem</i>	<i>Flasher de texto</i>
<i>Arquivo de calendário/anotações</i>	<i>Colorindo imagens P&B</i>	<i>Gerador de quebra-cabeças</i>
<i>Artista</i>	<i>Corrida de cavalos</i>	<i>Gerando e percorrendo labirinto</i>

<i>Jogo de bilhar</i>	<i>Marquee de imagem que rola</i>	<i>Rotação de imagens</i>
<i>Jogo de malha</i>	<i>Painel de marquee que gira</i>	<i>Simulador Simpletron multimídia</i>
<i>Jogo do 15</i>	<i>Palavras cruzadas</i>	<i>Testador de tempo de reação</i>
<i>Limericks</i>	<i>Relógio analógico</i>	<i>Transição aleatória entre imagens</i>

Você vai se divertir muito atacando estes problemas! Alguns vão tomar algumas horas e alguns são grandes projetos de conclusão de semestre. Vislumbramos todos os tipos de oportunidades para as disciplinas eletivas sobre multimídia começarem a aparecer nos currículos de computação nas universidades. Esperamos que você tenha competições com seus colegas de aula para desenvolver as melhores soluções para diversos destes problemas.

Capítulo 19: Estruturas de dados – é particularmente valioso em cursos universitários de segundo e terceito semestre. O capítulo discute as técnicas usadas para criar e manipular estruturas de dados dinâmicas, como listas encadeadas, pilhas, filas (i.e., filas de espera) e árvores. O capítulo inicia com discussões sobre classes auto-referenciais e alocação dinâmica de memória. Prosseguimos com uma discussão sobre como criar e manter diversas estruturas de dados dinâmicas. Para cada tipo de estrutura de dados, apresentamos programas com código ativo e mostramos exemplos de saída. Embora valha a pena saber como estas classes são implementadas, os programadores Java vão descobrir rapidamente que muitas das estruturas de dados de que eles precisam já estão disponíveis em bibliotecas de classes, como a própria `java.util` de Java, que discutimos no Capítulo 20, e nas `Collections` de Java, que discutimos no Capítulo 21. O capítulo ajuda o aluno a dominar as referências no estilo de Java (i.e., os substitutos de Java para os mais perigosos ponteiros de C e C++). Um problema quando se trabalha com referências é que os alunos podem ter dificuldade em visualizar as estruturas de dados e como seus nodos são ligados entre si. Portanto, apresentamos ilustrações que mostram as ligações e a seqüência na qual são criados. O exemplo de árvore binária é um belo coroamento para o estudo de referências e estruturas de dados dinâmicas. Este exemplo cria uma árvore binária; garante a eliminação de duplicatas e apresenta as travessias da árvore em pré-ordem, na ordem e em pós-ordem. Os alunos têm uma genuína sensação de missão cumprida quando eles estudam e implementam este exemplo. Eles gostam em especial de ver que a travessia na ordem imprime os valores dos nodos em ordem crescente. O capítulo inclui uma coleção substancial de exercícios. Um destaque dos exercícios é a seção especial “Construindo seu próprio compilador”. Este exercício é baseado em exercícios anteriores que conduzem os estudantes através do desenvolvimento de um programa de conversão de notação infixa para pós-fixa e um programa de avaliação de expressões pós-fixas. A seguir modificamos o algoritmo de avaliação pós-fixa para gerar código em linguagem de máquina. O compilador coloca este código em um arquivo (usando as técnicas que o aluno dominou no Capítulo 16). Os alunos então executam a linguagem de máquina produzida por seus compiladores nos simuladores em *software* que eles construíram nos exercícios do Capítulo 7! Os vários exercícios incluem uma simulação de supermercado usando enfileiramento, pesquisa recursiva em uma lista, impressão recursiva de uma lista de trás para diante, remoção de nodos de uma árvore, travessia de uma árvore binária em ordem de nível, impressão de árvores, escritura de um pedaço de um compilador otimizador, análise do desempenho de pesquisa e ordenação em árvores binárias e implementação de uma classe lista indexada.

Capítulo 20: Pacote de utilitários Java e manipulação de bits – percorre as classes do pacote `java.util` e discute cada um dos operadores sobre *bits* de Java. Este é um bom capítulo para reforçar a noção de reutilização. Quando já existem classes, é muito mais rápido desenvolver *software* simplesmente reutilizando estas classes do que “reinventando a roda”. As classes são incluídas em bibliotecas de classes porque as classes geralmente são úteis, corretas, estão ajustadas para o melhor desempenho, são certificadas quanto à portabilidade e/ou por diversas outras razões. Alguém investiu considerável trabalho na preparação dessas classes, então por que você deveria escrever sua própria? As bibliotecas de classe no mundo estão crescendo a uma taxa fenomenal. Sendo assim, sua habilidade e valor enquanto programador dependerá da sua familiaridade com as classes existentes e da maneira você pode reutilizá-las de maneira inteligente para desenvolver rapidamente *softwares* de alta qualidade. Os cursos universitários sobre estruturas de dados irão mudar drasticamente ao longo dos próximos anos porque as estruturas de dados mais importantes já estão implementadas em bibliotecas de classes amplamente disponíveis. Este capítulo discute muitas classes. Duas das mais úteis são `Vector` (um *array* dinâmico que pode crescer e encolher conforme necessário) e `Stack` (uma estrutura de dados dinâmica que permite inserções e retiradas a partir de apenas uma extremidade – chamada de *topo* – assegurando assim um comportamento “último a entrar primeiro a sair”). A beleza de estudar essas duas classes é que elas são relacionadas por herança, como discutido no Capítulo 9, de modo que o próprio pacote `java.util` implementa algumas classes em termos de outras, e isso o impede de reinventar a roda e lhe permite tirar o máximo proveito do recurso. Também discutimos as classes `Dictionary`, `Hashtable`, `Properties` (para criar e manipular `Hashtables` persistentes), `Random` e `BitSet`. A discussão sobre `BitSet` inclui código ativo para uma das aplicações clássicas de `BitSets`, que é o *Crivo de Eratostenes*, usado para determinar

se um número é primo. O capítulo discute em detalhes os poderosos recursos de Java para manipulação de *bits*, que permitem aos programadores exercitar recursos de *hardware* de mais baixo nível. Isto ajuda os programas a processar cadeias de *bits*, ligar e desligar *bit* individual e armazenar informações de uma maneira mais compacta. Tais recursos – herdados de C – são característicos de linguagens simbólicas de baixo nível e são valorizados por programadores que escrevem *software* de sistemas, como sistemas operacionais e *software* para redes.

Capítulo 21: Coleções – discute muitas das classes de Java 2 (do pacote `java.util`) que fornecem implementações predefinidas de muitas das estruturas de dados discutidas no Capítulo 19. Este capítulo também reforça a noção de reutilização. Essas classes são modeladas segundo uma biblioteca de classes semelhante em C++ – biblioteca-padrão de gabaritos. As coleções fornecem aos programadores Java um conjunto-padrão de estruturas de dados para armazenar e recuperar dados e um conjunto-padrão de algoritmos (isto é, procedimentos) que permitem aos programadores manipular dados (como procurar itens de dados particulares e armazenar dados em ordem crescente ou decrescente). Os exemplos do capítulo demonstram coleções, como listas encadeadas, árvores, mapeamentos e conjuntos, e algoritmos para pesquisar, ordenar, encontrar o valor máximo, encontrar o valor mínimo e assim por diante. Cada exemplo mostra claramente o poder e a facilidade de se usar as coleções. Os exercícios sugerem modificações para os exemplos do capítulo e pedem ao leitor para implementar novamente, usando coleções, estruturas de dados apresentadas no Capítulo 19.

Capítulo 22: Java Media Framework e Java Sound (no CD) – é o segundo de nossos dois capítulos dedicados aos tremendos recursos de Java para multimídia. Este capítulo se concentra no Java Media Framework (JMF) e na Java Sound API. O Java Media Framework oferece recursos tanto para áudio quanto para vídeo. Com o JMF, um programa Java pode reproduzir mídia de áudio e vídeo e capturar mídia de áudio e vídeo a partir de dispositivos como microfones e câmeras de vídeo. Muitas das aplicações com multimídia de hoje em dia envolvem o envio de trechos de áudio e vídeo através da Internet. Por exemplo, você pode visitar o site cnn.com para ver ou escutar conferências de imprensa ao vivo e muitas pessoas escutam estações de rádio baseadas na Internet através de seus navegadores da Web. O JMF permite criar as chamadas aplicações de *streaming media*, nas quais um programa Java envia trechos de áudio ou vídeo, gravados ou ao vivo, através da Internet para outros computadores e então os aplicativos nestes outros computadores reproduzem a mídia à medida que ela chega através da rede. As Java Sound APIs permitem manipular sons e mídia gravados do tipo Musical Instrument Digital Interface (MIDI), isto é, mídia oriunda de um dispositivo como um microfone. Este capítulo termina com uma aplicação substancial de processamento MIDI que permite selecionar arquivos MIDI para reproduzir e gravar novos arquivos MIDI. Os usuários podem criar sua própria música MIDI interagindo com o teclado de sintetizador simulado da aplicação. Além disso, a aplicação pode sincronizar a reprodução das notas em um arquivo MIDI com o pressionamento das teclas de sintetizador simulado – de forma semelhante a uma pianola! Assim como no Capítulo 18, depois de ler este capítulo você estará ansioso para experimentar todas estas técnicas, de modo que incluímos 44 exercícios adicionais sobre multimídia para desafiá-lo e diverti-lo. Eis alguns projetos interessantes:

A tartaruga e a lebre

Código Morse

Contador de histórias

Craps

Demonstração de física: bola que salta

Demonstração de física: cinética

Jogo da velha

Karaokê

Máquina de fliperama

Percorrendo o passeio do cavalo

Relógio digital

Reprodutor de MP3

Roleta

Simulador de vôo

Sistema de autoria multimídia

Torres de Hanói

Vídeo games

Videoconferência

Apêndice A: Demos de Java – apresenta uma enorme coleção de algumas das melhores demonstrações de Java disponíveis na Web. Muitos destes sites tornam seu código-fonte disponível para você, de modo que você pode baixar o código e adicionar suas próprias características – uma maneira verdadeiramente fantástica de aprender Java! Incentivamos nossos alunos a fazer isto, e estamos impressionados com os resultados! Você deve começar sua pesquisa dando uma olhada na página de *applets* da Sun Microsystems, java.sun.com/applets. Você pode economizar tempo para encontrar as melhores demonstrações dando uma olhada no JARS (o Java Applet Rating Service – serviço de avaliação de *applets* Java) em www.jars.com. Aqui está uma lista de algumas das demonstrações mencionadas no Apêndice A (os URLs e as descrições de cada uma estão no Apêndice A):

Animated SDSU Logo

Bumpy Lens 3D

Centípedo

Crazy Counter

Famous Curves Applet Index

Goldmine

Iceblox Game

Java Game Park

Java4fun games

Missile Comando

PhotoAlbum II

Play a Piano

*Sab's Game Arcade
SabBowl bowling game
Sevilla RDM 168*

*Stereoscopic 3D Hypercube
Teamball
Tube*

*Urbanoids
Warp 1.5*

Apêndice B: Recursos para Java – apresenta alguns dos melhores recursos para Java disponíveis na Web. Esta é uma maneira fantástica para você entrar no “mundo de Java”. O apêndice lista vários recursos para Java, como consórcios, jornais e empresas que fazem vários produtos fundamentais relacionados com Java. Aqui estão alguns dos recursos mencionados no Apêndice B:

<i>ambiente de desenvolvimento integrado JBuilder da Borland</i>	<i>grupos de discussão</i>	<i>Object Management Group</i>
<i>ambiente de desenvolvimento integrado NetBeans</i>	<i>grupos de notícias</i>	<i>produtos</i>
<i>applets animados</i>	<i>grupos de usuários Java (JUGs)</i>	<i>projetos</i>
<i>applets para baixar</i>	<i>homepage de CORBA</i>	<i>publicações</i>
<i>applets/aplicativos</i>	<i>IBM Developers Java Zone</i>	<i>quebra-cabeças</i>
<i>aprendendo Java</i>	<i>informações atuais</i>	<i>recursos</i>
<i>artes e entretenimento</i>	<i>Intelligence.com</i>	<i>revista on-line Java World</i>
<i>bancos de dados</i>	<i>Java Applet Rating Service</i>	<i>revista on-line Sun World</i>
<i>boletins de notícias</i>	<i>Java Developer Connection</i>	<i>seminários</i>
<i>coleções de multimídia</i>	<i>Java Developer's Journal</i>	<i>sessões ao vivo de bate-papo sobre Java</i>
<i>concursos</i>	<i>Java Media Framework</i>	<i>sites</i>
<i>conferências</i>	<i>Java Report</i>	<i>sites de áudio</i>
<i>consultores</i>	<i>Java Toys</i>	<i>softwares</i>
<i>demonstrações (muitas com código-fonte)</i>	<i>Java Woman</i>	<i>Sun Microsystems</i>
<i>documentação</i>	<i>java.sun.com</i>	<i>Team Java</i>
<i>FAQs (perguntas freqüentes)</i>	<i>jogos</i>	<i>The Java Tutorial</i>
<i>feiras e exposições</i>	<i>kits para desenvolvedores</i>	<i>treinamento (ligue para nós !)</i>
<i>ferramentas de desenvolvimento</i>	<i>links para sites sobre Java</i>	<i>tutoriais para aprender Java</i>
<i>ferramentas para Java</i>	<i>lista de recursos</i>	<i>URLs para applets Java</i>
<i>galeria de multimídia da NASA</i>	<i>lista do que há de novo e interessante</i>	<i>www.javaworld.com</i>
<i>gráficos</i>	<i>livros</i>	<i>Yahoo (mecanismo de busca na Web)</i>
	<i>materiais de referência</i>	
	<i>news:comp.lang.java</i>	
	<i>notícias</i>	

Apêndice C: Tabela de precedência de operadores – lista cada um dos operadores de Java e indica sua precedência relativa e associatividade. Listamos cada operador em uma linha separada e incluímos o nome completo do operador.

Apêndice D: Conjunto de caracteres ASCII – lista os caracteres do conjunto de caracteres ASCII (American Standard Code for Information Interchange) e indica o valor do código do caractere para cada um. Java usa o conjunto de caracteres Unicode, com caracteres de 16 bits para representar todos os caracteres das linguagens “significativas comercialmente” do mundo. O Unicode inclui ASCII como subconjunto. Atualmente, a maior parte dos países de língua inglesa estão usando ASCII e apenas começando a experimentar o Unicode.

Apêndice E: Sistemas de numeração (no CD) – discute os sistemas de numeração binário (base 2), decimal (base 10), octal (base 8) e hexadecimal (base 16). Este material é valioso para os cursos de introdução em ciência da computação e engenharia de computação. O apêndice é apresentado com os mesmos recursos pedagógicos de aprendizado que os capítulos do livro. Um recurso interessante do apêndice são seus 31 exercícios, 19 dos quais são exercícios de auto-revisão, com respostas.

Apêndice F: Criando documentação em HTML com javadoc (no CD) – apresenta a ferramenta de geração de documentação **javadoc**. A Sun Microsystems utiliza **javadoc** para documentar as APIs de Java. O exemplo neste apêndice conduz o leitor através do processo de documentação de **javadoc**. Primeiro, apresentamos o estilo de comentários e marcas que **javadoc** reconhece e usa para criar documentação. A seguir, discutimos os comandos e opções usados para executar o utilitário. Finalmente, examinamos os arquivos-fonte que **javadoc** utiliza e os arquivos HTML que **javadoc** cria.

1.18 (Opcional) Um passeio pelo estudo de caso sobre projeto orientado a objetos com a UML

Nesta seção e na seguinte, passeamos pelos dois principais destaques opcionais do livro – o estudo de caso opcional sobre projeto orientado a objetos com a UML e nossa introdução a padrões de projeto. O estudo de caso envolvente do projeto orientado a objetos com a UML é um acréscimo importante a *Java Como Programar, Quarta edição*. Este passeio antecipa o conteúdo das seções “Pensando em objetos” e discute como elas se relacionam com o estudo de caso. Após completar este estudo de caso, você terá completado o projeto e a implementação orientados a objetos de uma aplicação Java significativa.

Seção 1.15: Pensando em objetos: introdução à tecnologia de objetos e à Unified Modeling Language

Esta seção apresenta o estudo de caso de projeto orientado a objetos com a UML. Fornecemos um embasamento geral sobre o que são objetos e como eles interagem com outros objetos. Também discutimos brevemente o estado atual do setor de engenharia de *software* e como a UML influenciou os processos de análise e projeto orientados a objetos.

Seção 2.9: (Estudo de caso opcional) Pensando em objetos: examinando a definição do problema

Nosso estudo de caso começa com uma *definição de problema* que especifica os requisitos para um sistema que iremos criar. Neste estudo de caso, projetamos e implementamos uma simulação de um sistema de elevador em um prédio de dois andares. O usuário da aplicação pode “criar” uma pessoa em cada andar. Esta pessoa caminha pelo andar até o elevador, aperta o botão, espera que o elevador chegue e anda nele até o outro andar. Fornecemos o projeto de nosso sistema de elevador após investigar a estrutura e o comportamento de sistemas orientados a objetos em geral. Discutimos como a UML facilitará o processo de projeto em seções “Pensando sobre objetos” subsequentes, nos fornecendo diversos tipos de diagramas para modelar nosso sistema. Finalmente, fornecemos uma lista de referências para URLs e livros sobre projeto orientado a objetos com a UML. Você poderá achar estas referências úteis à medida que prosseguir através de nossa apresentação do estudo de caso.

Seção 3.8: (Estudo de caso opcional) Pensando em objetos: identificando as classes em uma definição de problema

Nesta seção, projetamos o modelo de simulação do elevador, que representa as operações do sistema do elevador. Identificamos as classes, ou “blocos de construção”, de nosso modelo extraiendo os substantivos e as frases com substantivos da definição do problema. Organizamos estas classes em um diagrama de classes UML que descreve a estrutura de classes de nosso modelo. O diagrama de classes também descreve relacionamentos, conhecidos como *associações*, entre classes (por exemplo, uma pessoa tem uma associação com o elevador, porque a pessoa anda no elevador). Finalmente, extraímos do diagrama de classes um outro tipo de diagrama da UML – o diagrama de objetos. O diagrama de objetos modela os objetos (instâncias das classes) em um momento específico de nossa simulação.

Seção 4.14: (Estudo de caso opcional) Pensando em objetos: identificando os atributos das classes

Uma classe contém *atributos* (dados) e *operações* (comportamentos). Esta seção focaliza os atributos das classes discutidas na Seção 3.7. Como vemos em seções posteriores, mudanças nos atributos de um objeto freqüentemente afetam o comportamento daquele objeto. Para determinar os atributos para as classes em nosso estudo de caso, extraímos os adjetivos que descrevem os substantivos e as frases com substantivos (os quais definiram nossas classes) da definição do problema e, então, colocamos os atributos no diagrama de classes que criamos na Seção 3.7.

Seção 5.11: (Estudo de caso opcional) Pensando em objetos: identificando estados e atividades dos objetos

Um objeto, a qualquer momento, se encontra em uma condição específica chamada de *estado*. Ocorre uma *transição de estado* quando aquele objeto recebe uma mensagem para mudar de estado. A UML fornece o *diagrama de estados*, que identifica o conjunto de estados possíveis em que um objeto pode se encontrar e modela as transições de estado daquele objeto. O objeto também tem uma *atividade* – o trabalho executado por um objeto ao longo de sua existência. A UML fornece o diagrama de atividade – um fluxograma que modela a atividade de um objeto. Nesta seção, usamos os dois tipos de diagramas para começar a modelar aspectos de comportamento específicos de nossa simulação de elevador, tais como: como uma pessoa anda de elevador e como o elevador responde quando o botão é pressionado em um determinado andar.

Seção 6.16: (Estudo de caso opcional) Pensando em objetos: identificando operações de classes

Nesta seção, identificamos as operações, ou serviços, de nossas classes. Extraímos da definição do problema os verbos e as frases com verbos que especificam as operações para cada classe. A seguir modificamos o diagrama de classes da Fig. 3.16 para incluir cada operação com sua classe associada. Neste ponto do estudo de caso, teremos reunido todas as informações possíveis a partir da definição do problema. Entretanto, à medida que os capítulos futuros introduzirem tópicos como herança, tratamento de eventos e *multithreading*, modificaremos nossas classes e diagramas.

Seção 7.10: (Estudo de caso opcional) Pensando em objetos: colaboração entre objetos

Neste ponto, criamos um “esboço grosso” do modelo para nosso sistema de elevador. Nesta seção, vemos como ele funciona. Investigamos o comportamento do modelo discutindo *colaborações* – mensagens que os objetos enviam uns para os outros para se comunicar. As operações de classes que descobrimos na Seção 6.16 se tornam as colaborações entre objetos em nosso sistema. Determinamos as colaborações em nosso sistema e então as reunimos em um *diagrama de colaborações* – o diagrama da UML para modelar colaborações. Este diagrama revela quais objetos colaboram e quando. Apresentamos um diagrama de colaborações das pessoas que entram e saem do elevador.

Seção 8.17: (Estudo de caso opcional) Pensando em objetos: começando a programar as classes para a simulação do elevador

Nesta seção, fazemos uma pausa no projeto do comportamento de nosso sistema. Começamos o processo de implementação para enfatizar o material discutido no Capítulo 8. Usando o diagrama de classes da UML da Seção 3.7 e os atributos e as operações discutidos nas Seções 4.14 e 6.16, mostramos como implementar uma classe em Java a partir de um projeto. Não implementamos todas as classes – porque ainda não completamos nosso processo de projeto. Trabalhando a partir de nossos diagramas de UML, criamos o código para a classe **Elevator**.

Seção 9.23: (Estudo de caso opcional) Pensando em objetos: incorporando herança à simulação do elevador

O Capítulo 9 inicia nossa discussão de programação orientada a objetos. Analisamos herança – as classes que compartilham características semelhantes podem herdar atributos e operações de uma classe “base”. Nesta seção, investigamos como nossa simulação de elevador pode se beneficiar do uso de herança. Documentamos nossas descobertas em um diagrama de classes que modela relacionamentos de herança – a UML se refere a estes relacionamentos como *generalizações*. Modificamos o diagrama de classes da Seção 3.7 usando herança para agrupar classes com características semelhantes. Continuamos a implementar a classe **Elevator** da Seção 8.17 com o recurso herança.

Seção 10.22: (Estudo de caso opcional) Pensando em objetos: tratamento de eventos

Nesta seção, incluímos as interfaces necessárias para que os objetos em nossa simulação de elevador enviem mensagens para outros objetos. Em Java, freqüentemente os objetos se comunicam enviando um *evento* – um aviso de que alguma ação ocorreu. O objeto que recebe o evento então executa uma ação em resposta ao tipo de evento recebido – isto é conhecido como *tratamento de eventos*. Na Seção 7.10, delineamos a passagem de mensagens, ou as colaborações, em nosso modelo, usando um diagrama de colaborações. Agora modificamos este diagrama para incluir o tratamento de eventos e, como exemplo, explicamos em detalhes como as portas em nossa simulação abrem quando o elevador chega.

Seção 11.10: (Estudo de caso opcional) Pensando em objetos: projetando interfaces com a UML

Nesta seção, projetamos um diagrama de classes que modela os relacionamentos entre classes e interfaces em nossa simulação – a UML se refere a estes relacionamentos como *realizações*. Além disso, listamos todas as operações que cada interface fornece para as classes. Finalmente, mostramos como criar as classes Java que implementam estas interfaces. Como na Seção 8.17 e na Seção 9.23, usamos a classe **Elevator** para demonstrar a implementação.

Seção 12.16: (Estudo de caso opcional) Pensando em objetos: casos de uso

O Capítulo 12 discute interfaces com o usuário que permitem a um usuário interagir com um programa. Nesta seção, discutimos a interação entre nossa simulação de elevador e seu usuário. Especificamente, investigamos os cenários que podem ocorrer entre o usuário da aplicação e a própria simulação – este conjunto de cenários é chamado de *caso de uso*. Modelamos estas interações usando *diagramas de casos de uso* da UML. Discutimos então a interface gráfica com o usuário para nossa simulação, usando nossos diagramas de casos de uso.

Seção 13.17: (Estudo de caso opcional) Pensando em objetos: Model – View – Controller

Projetamos nosso sistema para consistir em três componentes, cada um tendo uma responsabilidade distinta. A esta altura do estudo de caso, quase completamos o primeiro componente, chamado de *modelo*, que contém os dados que representam a simulação. Projetamos a *visão* – o segundo componente, que trata de como o modelo é exibido – na Seção 22.8. Projetamos o *controlador* – o componente que permite ao usuário controlar o modelo – na Seção 12.16. De um sistema como o nosso, que usa os componentes modelo, visão e controlador, diz-se que adere à arquitetura *modelo-visão-controlador* (MVC – *model, view, controller*). Nesta seção, explicamos as vantagens de usar esta arquitetura para projetar softwares. Usamos o *diagrama de componentes* da UML para modelar os três componentes e depois implementamos este diagrama como código em Java.

Seção 15.12: (Estudo de caso opcional) Pensando em objetos: multithreading

No mundo real, objetos operam e interagem concorrentemente. Java é uma linguagem *multithreaded*, o que permite aos objetos em nossa simulação agir de forma aparentemente independente uns dos outros. Nesta seção, declaramos certos objetos como “*threads*” para permitir que estes objetos operem concorrentemente. Modificamos o diagrama de colaborações apresentado originalmente na Seção 7.10 (e modificado na Seção 10.22) para incorporar *multithreading*. Apresentamos o diagrama de seqüência de UML para modelar interações em um sistema. Este diagrama enfatiza o ordenamento cronológico de mensagens. Usamos um diagrama de seqüência para modelar como uma pessoa dentro da simulação interage com o elevador. Esta seção conclui o projeto da parte modelo de nossa simulação. Projetamos como este modelo é exibido na Seção 22.9 e implementamos este modelo como código Java no Apêndice H.

Seção 22.9: (Estudo de caso opcional) Pensando em objetos: animação e som na visão

Esta seção projeta a visão, que especifica como a parte modelo da simulação é exibida. O Capítulo 18 apresenta diversas técnicas para integrar animação em programas e o Capítulo 22 apresenta técnicas para integrar som. A Seção 22.9 usa algumas destas técnicas para incorporar som e animação à nossa simulação de elevador. Especificamente, esta seção trata da animação dos movimentos de pessoas e de nosso elevador, gerando efeitos de som e tocando “música de elevador” quando uma pessoa anda no elevador. Esta seção conclui o projeto de nossa simulação de elevador. Os Apêndices G, H e I implementam este projeto como um programa Java de 3.594 linhas, completamente operacional.

Apêndice G: Eventos e interfaces listener do elevador (no CD)

[Nota: este apêndice encontra-se no CD que acompanha este livro.] Como discutimos na Seção 10.22, diversos objetos em nossa simulação interagem uns com os outros enviando mensagens, denominadas eventos, para outros objetos que desejam receber estes eventos. Os objetos que recebem os eventos são chamados de *objetos ouvintes* – estes precisam implementar as *interfaces ouvintes (listener)*. Nesta seção, implementamos todas as classes de evento e interfaces ouvintes usadas pelos objetos em nossa simulação.

Apêndice H: Modelo do elevador (no CD)

[Nota: este apêndice encontra-se no CD que acompanha este livro.] A maior parte do estudo de caso envolveu projetar o modelo (i.e., os dados e a lógica) da simulação do elevador. Nesta seção, implementamos aquele modelo em Java. Usando todos os diagramas de UML que criamos, apresentamos as classes Java necessárias para implementar o modelo. Aplicamos os conceitos de projeto orientado a objetos com a UML e a programação orientada a objetos e Java que você aprendeu nos capítulos.

Apêndice I: Visão do elevador (no CD)

[Nota: este apêndice encontra-se no CD que acompanha este livro.] A seção final implementa como exibimos o modelo do Apêndice H. Para implementar a visão, usamos a mesma abordagem que usamos para implementar o modelo – criamos todas as classes necessárias para executar a visão, usando os diagramas UML e os conceitos fundamentais discutidos nos capítulos. Ao chegar ao fim desta seção, você terá completado o projeto e a implementação “profissionais” de um sistema de grande porte. Você deve estar confiante para atacar sistemas maiores, como o estudo de caso de 8.000 linhas em Enterprise Java que apresentamos em nosso livro complementar *Advanced Java 2 Platform How to Program* e os tipos de aplicações que os engenheiros de software profissionais constroem. Esperamos que você prossiga com estudos ainda mais aprofundados de projeto orientado a objetos com a UML.

1.19 (Opcional) Um passeio pelas seções “Descobrindo padrões de projeto”

Nossa abordagem de padrões de projeto está espalhada ao longo de cinco seções opcionais do livro. Damos uma visão geral destas seções aqui.

Seção 9.24 – (Opcional) Descobrindo padrões de projeto: apresentando os padrões de criação, estruturais e comportamentais de projeto

Esta seção fornece tabelas que listam as seções nas quais discutimos os diversos padrões de projeto. Dividimos a discussão de cada seção em padrões de criação, estruturais e comportamentais de projetos. Os padrões de criação oferecem maneiras de instanciar objetos, os padrões estruturais dizem respeito à organização de objetos e os padrões comportamentais tratam das interações entre objetos. O restante da seção apresenta alguns destes padrões de projeto, como os padrões de projeto Singleton, Proxy, Memento e State. Finalmente, fornecemos diversos URLs para estudo adicional sobre padrões de projeto.

Seção 13.18 – (Opcional) Descobrindo padrões de projeto: padrões de projeto usados nos pacotes `java.awt` e `javax.swing`

Esta seção contém a maior parte de nossa discussão sobre padrões de projeto. Usando o material sobre componentes GUI de Java Swing nos Capítulos 12 e 13, investigamos alguns exemplos de uso de padrões nos pacotes `java.awt` e `javax.swing`. Discutimos como estas classes usam os padrões de projeto Factory Method, Adapter, Bridge, Composite, Chain-of-Responsibility, Command, Observer, Strategy e Template Method. Motivamos cada padrão e apresentamos exemplos de como aplicá-los.

Seção 15.13 – (Opcional) Descobrindo padrões de projeto: padrões de projeto simultâneos

Os desenvolvedores introduziram diversos padrões de projeto desde aqueles descritos pela “turma dos quatro”. Nesta seção, discutimos padrões de projeto simultâneos, incluindo Single-Threaded Execution, Guarded Suspension, Balking, Read/Write Lock e Two-Phase Termination – os quais resolvem diversos problemas de projeto em sistemas *multithreaded*. Investigamos como a classe `java.lang.Thread` utiliza padrões de simultaneidade.

Seção 17.11 – (Opcional) Descobrindo padrões de projeto: padrões de projeto usados nos pacotes `java.io` e `java.net`

Usando o material sobre arquivos, fluxos e redes nos Capítulos 16 e 17, investigamos alguns exemplos de uso de padrões nos pacotes `java.io` e `java.net`. Discutimos como estas classes usam os padrões de projeto Abstract Factory, Decorator e Facade. Também pensamos em padrões arquiteturais, os quais especificam um conjunto de subsistemas – agregados de objetos que coletivamente compreendem uma responsabilidade importante do sistema – e como estes subsistemas interagem uns com os outros. Discutimos os populares padrões arquiteturais Model-View-Controller e Layers.

Seção 21.12 – (Opcional) Descobrindo padrões de projeto: padrões de projeto usados no pacote `java.util`

Usando o material sobre estruturas de dados e coleções nos Capítulos 19, 20 e 21, investigamos o uso de padrões no pacote `java.util`. Discutimos como estas classes usam os padrões de projeto Prototype e Integrator. Esta seção conclui a discussão de padrões de projeto. Após terminar o material *Descobrindo padrões de projeto*, você deve ser capaz de reconhecer e usar padrões fundamentais e ter uma melhor compreensão do funcionamento da Java API. Depois de completar este material, recomendamos que você passe para o livro da “turma dos quatro”.

Bem, aí está! Trabalhamos muito para criar este livro e sua versão opcional *Cyber Classroom*. O livro está repleto de exemplos de código ativo, dicas de programação, exercícios de auto-revisão e respostas, exercícios e projetos desafiadores e inúmeros auxílios didáticos para ajudar a dominar o material. Java é uma linguagem de programação poderosa que ajuda a escrever programas de maneira rápida e eficiente. E Java é uma linguagem que pode ser escalonada elegantemente no domínio de desenvolvimento de sistemas corporativos para ajudar as organizações a construírem seus principais sistemas de informações. Enquanto você lê o livro, se algo não estiver claro, ou se você encontrar algum erro, escreva-nos para o endereço de correio eletrônico deitel@deitel.com. Responderemos prontamente e publicaremos as correções e esclarecimentos em nosso site da Web:

www.deitel.com

Esperamos que você goste de aprender com *Java Como Programar: Quarta Edição* tanto quanto nós gostamos escrevê-lo!

Resumo

- O *software* controla os computadores (freqüentemente conhecidos como *hardware*).
- Java é uma das linguagens de desenvolvimento de *softwares* mais populares atualmente.
- Java foi desenvolvida pela Sun Microsystems. A Sun fornece uma implementação do Java 2 Platform, Standard Edition, chamada Java 2 Software Development Kit (J2SDK), versão 1.3.1, que inclui o conjunto mínimo de ferramentas necessárias para escrever *softwares* em Java.
- Java é uma linguagem completamente orientada a objetos com forte suporte para técnicas adequadas de engenharia de *software*.
- O computador é um dispositivo capaz de realizar computações e tomar decisões lógicas a velocidades milhões e até bilhões de vezes mais rápidas que os seres humanos.
- Os computadores processam dados sob o controle de conjuntos de instruções chamados programas de computador. Esses programas de computador orientam o computador ao longo de conjuntos ordenados de ações especificadas por pessoas chamadas programadores de computador.
- Os vários dispositivos (como teclado, tela, discos, memória e unidades de processamento) que abrangem um sistema de computador são conhecidos como *hardware*.
- Os programas de computador que executam em um computador são conhecidos como *software*.
- A unidade de entrada é a seção “receptora” do computador. Ela recebe informações (dados e programas de computador) de vários dispositivos de entrada e coloca essas informações à disposição de outras unidades, de modo que as informações possam ser processadas.
- A unidade de saída é a seção de “despacho” do computador. Ela recebe as informações processadas pelo computador e as coloca em dispositivos de saída para torná-las disponíveis para utilização fora do computador.
- A unidade de memória é a seção de armazenamento de relativamente baixa capacidade e acesso rápido do computador. Ela retém informações que foram inseridas pela unidade de entrada, de modo que as informações podem tornar-se imediatamente disponíveis ao processamento quando for necessário, e retém informações que já foram processadas até que possam ser colocadas em dispositivos de saída pela unidade de saída.
- A unidade de aritmética e de lógica (ALU) é a seção de “fabricação” do computador. Ela é responsável por realizar cálculos como adição, subtração, multiplicação e divisão e por tomar decisões.
- A unidade central de processamento (CPU) é a seção “administrativa” do computador. Ela coordena o computador e é responsável por supervisionar a operação de outras seções.
- A unidade de armazenamento secundário é a seção de armazenamento de alta capacidade e de longo prazo do computador. Programas ou dados que não estão sendo utilizados pelas outras unidades normalmente são colocados em dispositivos de armazenamento secundários (como discos) até que sejam necessários, possivelmente horas, dias, meses ou mesmos anos mais tarde.
- Os primeiros computadores eram capazes de realizar apenas um trabalho ou tarefa por vez. Essa forma de operação de computador é freqüentemente chamada de processamento em lotes de um único usuário.
- Os sistemas de *software* chamados sistemas operacionais foram desenvolvidos para ajudar a tornar a utilização dos computadores mais conveniente. Os primeiros sistemas operacionais gerenciavam a transição suave entre os trabalhos e minimizavam o tempo que os operadores de computador levavam para alternar entre trabalhos.
- A multiprogramação envolve a operação “simultânea” de muitos trabalhos no computador – o computador compartilha seus recursos entre os trabalhos que disputam sua atenção.
- O compartilhamento de tempo é um caso especial da multiprogramação em que dezenas ou mesmo centenas de usuários compartilham o computador por terminais. O computador executa uma pequena parte de um trabalho de um usuário e então vai adiante para atender o próximo usuário. O computador faz isso tão rapidamente que pode fornecer serviço para cada usuário várias vezes por segundo, de tal modo que os programas dos usuários parecem rodar simultaneamente.
- Uma vantagem do compartilhamento de tempo é que o usuário recebe respostas quase imediatas às suas solicitações em vez de precisar esperar um longo tempo por resultados como ocorria com os primeiros modos de computação.
- Em 1977, a Apple Computer popularizou o fenômeno da computação pessoal.
- Em 1981, a IBM lançou o IBM Personal Computer. Quase da noite para o dia, a computação pessoal tornou-se uma realidade no comércio, na indústria e na organizações governamentais.

- Embora os primeiros computadores pessoais não fossem suficientemente poderosos para oferecer compartilhamento de tempo para vários usuários, essas máquinas podiam ser interconectadas em redes de computadores, às vezes através de linhas telefônicas e às vezes em redes locais (LANs) dentro de uma organização. Isso levou ao fenômeno do processamento distribuído em que a computação de uma organização é distribuída pelas redes entre os locais em que o trabalho real da organização é realizado.
- Atualmente, as informações são facilmente compartilhadas pelas redes de computadores em que alguns computadores, chamados servidores de arquivos, oferecem um armazenamento comum de programas e dados que podem ser utilizados por computadores clientes distribuídos por toda a rede – daí o termo computação cliente/servidor.
- Java se tornou a linguagem a linguagem preferida para desenvolver aplicativos baseados na Internet (e para muitos outros propósitos).
- As linguagens de computador podem ser divididas em três tipos gerais: linguagens de máquina, linguagens *assembler* e linguagens de alto nível.
- Qualquer computador pode entender diretamente apenas sua própria linguagem de máquina. As linguagens de máquina geralmente consistem em *strings* de números (em última instância reduzidas a 1s e 0s) que instrui os computadores a realizar suas operações mais elementares uma por vez. As linguagens de máquina são dependentes de máquina.
- Abreviações semelhantes ao inglês formaram a base das linguagens *assembler*. Os programas tradutores chamados *montadores* convertem programas de linguagem *assembler* em linguagem de máquina a velocidades de computador.
- Compiladores traduzem programas de linguagem de alto nível em programas de linguagem de máquina. As linguagens de alto nível (como Java) contêm palavras em inglês e notações matemáticas convencionais.
- Os programas interpretadores executam diretamente programas em linguagem de alto nível sem a necessidade de compilar esses programas para linguagem de máquina.
- Embora os programas compilados executem muito mais rapidamente que os programas interpretados, os interpretadores são populares em ambientes de desenvolvimento de programas em que os programas são freqüentemente recompilados à medida que novos recursos são adicionados e os erros são corrigidos.
- Os objetos são componentes de *software* essencialmente reutilizáveis que modelam itens do mundo real. As abordagens de projeto e implementação modulares e orientadas a objetos tornam os grupos de desenvolvimento de *software* mais produtivos do que até então era possível com as primeiras técnicas conhecidas de programação como a programação estruturada. Os programas orientados a objetos são freqüentemente mais fáceis de entender, corrigir e modificar.
- Java originou-se na Sun Microsystems como um projeto para dispositivos eletrônicos inteligentes para o consumidor.
- Quando a World Wide Web explodiu em popularidade em 1993, as pessoas da Sun viram imediatamente o potencial de utilizar Java para criar páginas da Web com o chamado conteúdo dinâmico.
- Java agora é utilizada para criar páginas da Web com conteúdo dinâmico e interativo, desenvolver aplicativos corporativos de larga escala, aprimorar a funcionalidade de servidores da Web, fornecer aplicativos para dispositivos destinados ao consumidor final e assim por diante.
- Os programas Java consistem em partes chamadas classes. As classes consistem em partes chamadas métodos que realizam tarefas e retornam informações quando elas completam suas tarefas.
- A maioria dos programadores Java utiliza ricas coleções de classes existentes nas bibliotecas de classes Java.
- FORTRAN (FORMula TRANslator) foi desenvolvida pela IBM Corporation entre 1954 e 1957 para aplicativos científicos de engenharia que exigem computações matemáticas complexas.
- COBOL (COmmon Business Oriented Language) foi desenvolvida em 1959 por um grupo de fabricantes de computador, pelo governo e pelos usuários de computadores industriais. COBOL é utilizada principalmente para aplicações comerciais que exigem manipulação precisa e eficiente de grandes quantidades de dados.
- Pascal foi projetada quase na mesma época que C. Foi criada pelo professor Nicklaus Wirth e planejada para utilização acadêmica.
- Basic foi desenvolvida em 1965 no Dartmouth College como uma linguagem simples para ajudar os iniciantes a se sentir à vontade com programação.
- A programação estruturada é uma abordagem disciplinada para escrever programas que são mais claros e mais fáceis de testar, depurar e modificar que os programas não-estruturados.
- A linguagem Ada foi desenvolvida sob o patrocínio do Departamento da Defesa dos EUA durante a década de 1970 e início da de 1980. Um importante recurso de Ada é a chamada multitarefa; permite que os programadores especifiquem quantas atividades devem ocorrer em paralelo.
- A maioria das linguagens de alto nível – incluindo C e C++ – geralmente permitem que o programador escreva programas que realizam apenas uma atividade por vez. Java, por meio de uma técnica chamada *multithreading*, permite que os programadores escrevam programas com atividades paralelas.

- A Internet foi desenvolvida há mais de três décadas com financiamento fornecido pelo Departamento da Defesa dos EUA. Originalmente projetada para conectar os principais sistemas de computador de aproximadamente uma dezena de universidades e organizações de pesquisa, hoje a Internet é acessível a centenas de milhões de computadores no mundo.
- A Web permite que os usuários de computador localizem e visualizem documentos que usam intensivamente multimídia pela Internet.
- Os sistemas Java geralmente consistem em várias partes: o ambiente, a linguagem, a Java Applications Programming Interface (API) e várias bibliotecas de classes.
- Os programas Java normalmente passam por cinco fases para serem executados – edição, compilação, carga, verificação e execução.
- Os nomes de arquivos de programas Java terminam com a extensão **.java**.
- O compilador Java (**javac**) traduz um programa Java em *bytecodes* – a linguagem entendida pelo interpretador Java. Se um programa compilar corretamente, o compilador produz um arquivo com a extensão **.class**. Esse é o arquivo que contém os *bytecodes* que são interpretados durante a fase de execução.
- Um programa Java deve primeiro ser colocado na memória antes de poder ser executado. Isso é feito pelo carregador de classes, que obtém o arquivo **.class** (ou arquivos) que contém os *bytecodes* e o transfere para memória. O arquivo **.class** pode ser carregado a partir de um disco em seu sistema ou através de uma rede.
- O aplicativo é um programa que normalmente é armazenado e executado no computador local do usuário.
- O *applet* é um pequeno programa que normalmente é armazenado em um computador remoto a que os usuários se conectam através de um navegador de Web. Os *applets* são carregados a partir de um computador remoto no navegador, executados no navegador e descartados ao completar a execução.
- Os aplicativos são carregados na memória e executados com o interpretador **java**.
- Navegadores são utilizados para visualizar documentos HTML (*HyperText Markup Language*) na World Wide Web.
- Quando o navegador vê um *applet* em um documento de HTML, o navegador carrega o carregador de classes Java para carregar o *applet*. Todos os navegadores que suportam Java têm um interpretador Java embutido. Uma vez que o *applet* é carregado, o interpretador Java no navegador começa a sua execução.
- Os *applets* também podem ser executados a partir da linha de comando com o comando **appletviewer** fornecido junto com Java 2 Software Development Kit (J2SDK). O **appletviewer** é comumente citado como navegador mínimo – ele só sabe como interpretar *applets*.
- Antes de os *bytecodes* de um *applet* serem executados pelo interpretador Java embutido em um navegador ou **appletviewer**, eles são verificados pelo verificador de *bytecode* para assegurar que os *bytecodes* das classes descarregadas são válidos e que não violam restrições de segurança de Java.
- Um passo intermediário entre interpretadores e compiladores é um compilador *just-in-time* (JIT) que, enquanto executa o interpretador, produz código compilado para os programas e executa os programas em linguagem de máquina em vez de reinterpretá-los. Os compiladores JIT não produzem linguagem de máquina que seja tão eficiente quanto um compilador completo.
- Para as organizações que desejam fazer desenvolvimento de sistemas de informações duráveis, ambientes integrados de desenvolvimento (IDEs) são disponibilizados por importantes fornecedores de *software*. Os IDEs fornecem muitas ferramentas para suportar o processo de desenvolvimento de *software*.
- A orientação a objetos é uma maneira natural de pensar no mundo e de escrever programas para computadores.
- A Unified Modeling Language (UML) é uma linguagem gráfica que permite às pessoas que constroem sistemas representar seus projetos orientados a objetos em uma notação comum.
- Os seres humanos pensam em termos de objetos. Possuímos a maravilhosa habilidade da abstração, que nos permite ver imagens em uma tela como pessoas, aviões, árvores e montanhas em vez de como pontos coloridos individuais (chamados píxeis – abreviatura de “*picture elements*”).
- Os seres humanos aprendem sobre objetos estudando seus atributos ou observando seus comportamentos. Objetos diferentes podem ter atributos semelhantes e podem exibir comportamentos semelhantes.
- O projeto orientado a objetos (OOD) modela objetos do mundo real. Ele tira proveito de relacionamentos entre classes, nos quais objetos de uma certa classe – como uma classe de veículos – têm as mesmas características. Ele se aproveita de relacionamentos de herança, e até de relacionamentos de herança múltipla, dos quais as classes de objetos recém-criadas se derivam absorvendo as características das classes existentes e adicionando características únicas destas próprias classes.
- O OOD encapsula dados (atributos) e funções (comportamento) em objetos; os dados e as funções de um objeto estão intimamente ligados entre si.
- Os objetos têm a propriedade de ocultação de informações. Significa que, embora os objetos possam saber como se comunicar uns com os outros através de interfaces bem-definidas, os objetos normalmente não estão autorizados a conhecer como outros objetos são implementados.

- Linguagens como Java são orientadas a objetos – o ato de programar numa linguagem assim chama-se programação orientada a objetos (OOP) e permite aos projetistas implementar o projeto orientado a objetos como um sistema que funciona.
- Em Java, a unidade de programação é a classe a partir da qual os objetos em algum momento são instanciados (um termo elegante para “criados”). As classes de Java contêm métodos (que implementam os comportamentos da classe) e atributos (que implementam os dados da classe).
- Os programadores Java se concentram em criar seus próprios tipos definidos pelo usuário, denominados classes. Cada classe contém dados e o conjunto de funções que manipulam aqueles dados. Os componentes de dados de uma classe de Java são chamados de atributos. Os componentes funções de uma classe Java são chamados de métodos.
- Uma instância de um tipo definido pelo usuário (i.e., uma classe) é chamada de objeto.
- As classes também podem ter relacionamento com outras classes. Estes relacionamentos são chamados de associações.
- Com a tecnologia de objetos, podemos construir a maior parte do *software* de que precisaremos combinando “peças padronizadas e intercambiáveis” denominados classes.
- O processo de analisar e projetar um sistema de um ponto de vista orientado a objetos é chamado de análise e projeto orientados a objetos (OOAD).
- A Unified Modeling Language (UML) é agora o esquema de representação gráfica mais amplamente utilizado para modelar sistemas orientados a objetos. Aqueles que projetam sistemas usam a linguagem (sob a forma de diagramas gráficos) para modelar seus sistemas.
- Ao longo da última década, o setor de engenharia de *software* fez progressos significativos no campo de padrões de projetos – arquiteturas testadas e aprovadas para a construção de softwares orientados a objetos flexível e fácil de manter. O uso de padrões de projeto pode reduzir substancialmente a complexidade do processo de projeto.
- Os padrões de projeto beneficiam os desenvolvedores de sistemas ajudando a construir softwares confiáveis com arquiteturas testadas e aprovadas e a experiência acumulada pelas empresas, promovendo a reutilização em sistemas futuros, identificando erros e armadilhas comuns que ocorrem quando se constrói um sistema, ajudando a projetar sistemas independentemente da linguagem na qual serão implementados, estabelecendo um vocabulário de projeto comum entre desenvolvedores e encurtando a fase de projeto em um processo de desenvolvimento de *software*.
- Os projetistas usam padrões de projeto para construir conjuntos de classes e objetos.
- Os padrões de criação de projeto descrevem técnicas para instanciar objetos (ou grupos de objetos).
- Os padrões estruturais de projeto permitem aos projetistas organizar classes e objetos em estruturas maiores.
- Os padrões comportamentais de projeto atribuem responsabilidades a objetos.

Terminologia

<i>abordagem de código ativo (live code™)</i>	<i>compartilhamento de tempo</i>
<i>abstração</i>	<i>compilador</i>
<i>Ada</i>	<i>compilador javac</i>
<i>ALU (arithmetic and logic unit)</i>	<i>compilador JIT (just-in-time)</i>
<i>análise e projeto orientados a objetos (OOAD)</i>	<i>componentes GUI Swing</i>
<i>aplicativo</i>	<i>componentes reutilizáveis</i>
<i>applet</i>	<i>comportamento</i>
<i>arquivo .class</i>	<i>computação cliente/servidor</i>
<i>array</i>	<i>computação pessoal</i>
<i>atributo</i>	<i>computador</i>
<i>Basic</i>	<i>condição</i>
<i>biblioteca-padrão de C</i>	<i>conteúdo dinâmico</i>
<i>bibliotecas de classes</i>	<i>CPU</i>
<i>bytecodes</i>	<i>definição do problema</i>
<i>C</i>	<i>dependente de máquina</i>
<i>CANSI</i>	<i>disco</i>
<i>C++</i>	<i>disparar uma exceção</i>
<i>carregador de classes</i>	<i>dispositivo de entrada</i>
<i>classe</i>	<i>dispositivo de saída</i>
<i>cliente</i>	<i>documento de requisitos</i>
<i>COBOL</i>	<i>Editor</i>
<i>código-fonte aberto</i>	<i>encapsulamento</i>
<i>coleções</i>	<i>entrada/saída (E/S)</i>
<i>comando appletviewer</i>	<i>erro de lógica</i>
	<i>erro de sintaxe</i>

<i>erro em tempo de compilação</i>	<i>navegador da Web Microsoft Internet Explorer</i>
<i>erro em tempo de execução</i>	<i>objeto</i>
<i>erro fatal em tempo de execução</i>	<i>ocultação de informações</i>
<i>erro não-fatal em tempo de execução</i>	<i>padrão comportamental de projeto</i>
<i>extensão .java</i>	<i>padrão de criação de projeto</i>
<i>extensão Java</i>	<i>padrão de projeto</i>
<i>fase de carga</i>	<i>padrão estrutural de projeto</i>
<i>fase de compilação</i>	<i>Pascal</i>
<i>fase de edição</i>	<i>plataformas</i>
<i>fase de execução</i>	<i>portabilidade</i>
<i>fase de verificação</i>	<i>processamento distribuído</i>
<i>Fortran</i>	<i>programa de computador</i>
<i>freeware</i>	<i>programação baseada em eventos</i>
<i>hardware</i>	<i>programação estruturada</i>
<i>herança</i>	<i>programação orientada a objetos (OOP)</i>
<i>HotSpot, compilador</i>	<i>programação procedural</i>
<i>HTML (Hypertext Markup Language)</i>	<i>programador de computador</i>
<i>IDE (Integrated Development Environment)</i>	<i>programas tradutores</i>
<i>independente de máquina</i>	<i>projeto orientado a objetos (OOD)</i>
<i>Internet</i>	<i>referência</i>
<i>interpretador</i>	<i>reutilização de software</i>
<i>interpretador java</i>	<i>servidor de arquivos</i>
<i>Java</i>	<i>shareware</i>
<i>Java 2 Software Development Kit (J2SDK)</i>	<i>sistemas legados</i>
<i>Java Virtual Machine</i>	<i>software</i>
<i>JDBC (Java Database Connectivity)</i>	<i>Sun Microsystems</i>
<i>linguagem assembler</i>	<i>unidade de entrada</i>
<i>linguagem de alto nível</i>	<i>unidade de memória</i>
<i>linguagem de máquina</i>	<i>unidade de saída</i>
<i>linguagem de programação</i>	<i>unidade lógica e aritmética</i>
<i>memória principal</i>	<i>unidade secundária de armazenamento</i>
<i>método</i>	<i>Unified Modeling Language (UML)</i>
<i>Microsoft</i>	<i>variável de instância</i>
<i>modelagem</i>	<i>verificador de bytecode</i>
<i>multiprocessador</i>	<i>vídeo</i>
<i>multitarefa</i>	<i>World Wide Web</i>
<i>multithreading</i>	
<i>navegador da Web Netscape Navigator</i>	

Exercícios de auto-revisão

- 1.1** Preencha as lacunas em cada uma das frases seguintes:
- A empresa que popularizou a computação pessoal foi a_____.
 - O computador que tornou a computação pessoal uma realidade no comércio e na indústria foi o_____.
 - Os computadores processam dados sob o controle de conjuntos de instruções chamados de_____.
 - As seis unidades lógicas-chave do computador são _____, _____, _____, _____, _____, e _____.
 - As três classes de linguagens discutidas no capítulo são _____, _____ e _____.
 - Os programas que traduzem programas de linguagem de alto nível em linguagem de máquina são chamados de_____.
- 1.2** Preencha as lacunas em cada uma das seguintes frases sobre o ambiente Java.
- O comando _____ do Java 2 Software Development Kit executa um *applet* Java.
 - O comando _____ do Java 2 Software Development Kit executa um aplicativo Java.
 - O comando _____ do Java 2 Software Development Kit 2 Kit compila um programa Java.
 - O arquivo _____ é necessário para invocar um *applet* Java.
 - O arquivo _____ de programa Java deve terminar com a extensão de arquivo_____.

- f) Quando um programa Java é compilado, o arquivo produzido pelo compilador termina com a extensão de arquivo _____.
- g) O arquivo produzido pelo compilador Java contém _____ que são interpretados para executar um *applet* ou aplicativo Java.
- 1.3** Preencha as lacunas em cada uma das seguintes frases:
- O _____ permite localizar e visualizar documentos baseados em multimídia sobre aproximadamente qualquer assunto pela Internet.
 - Os _____ Java em geral são armazenados em seu computador e são projetados para executar independente de um navegador da World Wide Web.
 - Listas e tabelas de valores são chamadas de _____.
 - Os componentes GUI _____ estão escritos completamente em Java.
 - O _____ permite a um *applet* ou aplicativo realizar múltiplas atividades em paralelo.
 - As _____ fornecem aos programadores de Java um conjunto-padrão de estruturas de dados para armazenar e recuperar dados e um conjunto-padrão de algoritmos que permite aos programadores manipular os dados.
- 1.4** Preencha as lacunas em cada uma das seguintes declarações (com base nas Seções 1.15 e 1.16):
- Ao longo da última década, o setor de engenharia de *software* fez progressos significativos no campo de _____ – arquiteturas testadas e aprovadas para construir *software* orientado a objetos flexível e fácil de manter.
 - Os objetos possuem a propriedade de _____.
 - Os programadores Java se concentram em criar seus próprios tipos definidos pelo usuário, chamados de _____.
 - As classes também têm relacionamentos com outras classes. Estes relacionamentos são chamados de _____.
 - O processo de analisar e projetar um sistema de um ponto de vista orientado a objetos é chamado de _____.

Respostas aos exercícios de auto-revisão

- 1.1** a) Apple. b) IBM Personal Computer. c) programas. d) unidade de entrada, unidade de saída, unidade de memória, unidade de aritmética e de lógica, unidade central de processamento, unidade secundária de armazenamento. e) linguagens de máquina, linguagens *assembler*, linguagens de alto nível. f) compiladores.
- 1.2** a) *appletviewer*. b) *java*. c) *javac*. d) HTML. e) *.java*. f) *.class*. g) *bytecodes*.
- 1.3** a) World Wide Web. b) Aplicativos. c) *arrays*. b) *Swing*. e) *multithreading*. f) coleções.
- 1.4** a) padrões de projetos. b) ocultação de informações. c) classes. d) associações. e) análise e projeto orientados a objetos (OOAD).

Exercícios

- 1.5** Classifique cada um dos itens seguintes como *hardware* ou *software*:
- CPU
 - compilador Java
 - ALU
 - interpretador Java
 - unidade de entrada
 - programa editor
- 1.6** Por que você poderia querer escrever um programa em uma linguagem independente de máquina em vez de uma linguagem dependente de máquina? Por que uma linguagem dependente de máquina talvez fosse mais apropriada para escrever certos tipos de programas?
- 1.7** Preencha as lacunas em cada uma das frases seguintes:
- Qual é a unidade lógica do computador que recebe informações de fora do computador para utilização pelo computador? _____.

- b) O processo de instrução do computador para resolver problemas específicos é chamado de _____.
- c) Que tipo de linguagem de computador utiliza abreviações semelhantes ao inglês para instruções de linguagem de máquina? _____.
- d) Qual é a unidade lógica do computador que envia informações que já foram processadas pelo computador para vários dispositivos, de modo que as informações possam ser utilizadas fora do computador? _____.
- e) Qual é a unidade lógica do computador que retém informações? _____.
- f) Qual é a unidade lógica do computador que realiza cálculos? _____.
- g) Qual é a unidade lógica do computador que toma decisões lógicas? .
- h) O nível de linguagem de computador mais conveniente para que o programador escreva programas rápida e facilmente é _____.
- i) A única linguagem que um computador pode entender diretamente é chamada de _____ do computador.
- j) Qual é a unidade lógica do computador que coordena as atividades de todas as outras unidades lógicas? _____.

1.8 Explique a diferença entre os termos *erro fatal* e *erro não-fatal*. Por que você preferiria ter um erro fatal em vez de um erro não-fatal?

1.9 Preencha as lacunas em cada uma das frases seguintes:

- a) Java _____ são projetados para serem transportados pela Internet e executados em navegadores da World Wide Web.
- b) A programação _____ faz com que um programa realize uma tarefa em resposta a interações de usuário com componentes de interface gráfica com o usuário (GUI).
- c) As capacidades gráficas de Java são _____ e, portanto, são portáveis.
- d) O padrão _____ pode ser utilizado para fornecer interfaces com usuário idênticas entre todas as plataformas de computador.
- e) As linguagens que não podem realizar múltiplas atividades em paralelo são chamadas de linguagens _____ ou linguagens _____.
- f) As agregações de dados como listas encadeadas, pilhas, filas e árvores são chamadas de _____.

1.10 Preencha as lacunas em cada uma das frases seguintes (baseadas nas Seções 1.15 e 1.16):

- a) Os padrões _____ de projeto descrevem as técnicas para instanciar objetos (ou grupos de objetos).
- b) A _____ é agora o esquema de representação gráfica mais amplamente utilizado para a modelagem de sistemas orientados a objetos..
- c) As classes de Java contêm _____ (que implementam os comportamentos das classes) e _____ (que implementam os dados das classes).
- d) Os padrões _____ de projeto permitem organizar classes e objetos em estruturas maiores.
- e) Os padrões _____ de projeto atribuem responsabilidades aos objetos.
- f) Em Java, a unidade de programação é a _____, da qual _____ são instanciados em algum momento.

2

Introdução a aplicativos Java

Objetivos

- Ser capaz de escrever aplicativos Java simples.
- Ser capaz de utilizar instruções de entrada e saída.
- Familiarizar-se com tipos de dados primitivos.
- Entender conceitos básicos de memória.
- Ser capaz de utilizar operadores aritméticos.
- Entender a ordem de precedência dos operadores aritméticos.
- Ser capaz de escrever instruções de tomada de decisão.
- Ser capaz de utilizar operadores relacionais de igualdade.

A opinião é livre, mas os fatos são sagrados.

C. P. Scott

O credor tem melhor memória que o devedor.

James Howell

Quando preciso tomar uma decisão, sempre pergunto: "O que seria mais divertido?"

Peggy Walker

Ele deixou seu corpo para a ciência – e a ciência está contestando sua vontade.

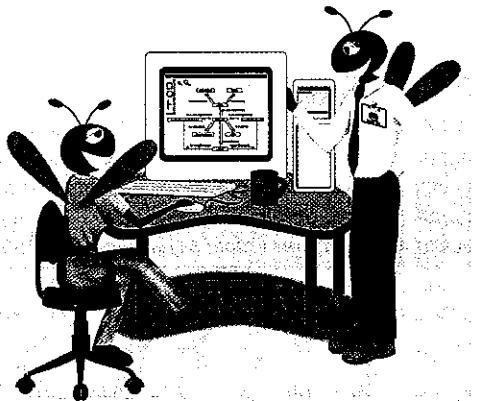
David Frost

As classes lutam, algumas classes triunfam, outras são eliminadas.

Mao Tse Tung

A igualdade, em um sentido social, pode ser dividida naquela de fato e naquela de direito.

James Fenimore Cooper



Sumário do capítulo

- 2.1 Introdução**
- 2.2 Um primeiro programa em Java: imprimindo uma linha de texto**
 - 2.2.1 Compilando e executando seu primeiro aplicativo Java
- 2.3 Modificando nosso primeiro programa em Java**
 - 2.3.1 Exibindo uma única linha de texto com múltiplas instruções
 - 2.3.2 Exibindo múltiplas linhas de texto com uma única instrução
- 2.4 Exibindo texto em uma caixa de diálogo**
- 2.5 Outro aplicativo Java: adicionando inteiros**
- 2.6 Conceitos de memória**
- 2.7 Aritmética**
- 2.8 Tomada de decisão: operadores de igualdade e operadores relacionais**
- 2.9 (Estudo de caso opcional) Pensando em objetos: examinando a definição do problema**

Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão • Exercícios

2.1 Introdução

A linguagem Java facilita uma abordagem disciplinada para o projeto de programas de computador. Apresentamos agora a programação em Java e apresentamos exemplos que ilustram diversos recursos importantes de Java. Cada exemplo é analisado tomando-se uma linha por vez. Neste capítulo e no Capítulo 3, apresentamos dois tipos de programa em Java – *aplicativos* e *applets*. No Capítulo 4 e no Capítulo 5, apresentamos um tratamento detalhado do *desenvolvimento de programa* e *controle de programa* em Java.

2.2 Um primeiro programa em Java: imprimindo uma linha de texto

Java utiliza notações que podem parecer estranhas para os não-programadores. Iniciamos considerando um simples *aplicativo* que exibe uma linha de texto. O aplicativo é um programa que é executado com o interpretador **java** (discutido mais tarde nesta seção). O programa e a sua saída são mostrados na Fig. 2.1.

Esse programa ilustra vários recursos importantes da linguagem Java. Consideremos cada linha do programa em detalhe. Cada programa que apresentamos neste livro tem números nas linhas para a conveniência do leitor; esses números não fazem parte dos programas em si. A linha 9 faz o “trabalho real” do programa, a saber, exibir a frase **Welcome to Java Programming!** na tela. Mas vamos considerar cada linha na ordem. A linha 1

```
// Fig. 2.1: Welcome1.java
```

inicia com **//**, indicando que o restante da linha é um *comentário*. Os programadores inserem comentários para *documentar* os programas e melhorar a legibilidade do programa. Os comentários também ajudam outras pessoas a ler e entender o programa. Os comentários não fazem com que o computador realize uma ação quando o programa é executado. O compilador Java ignora os comentários. Iniciamos cada programa com um comentário indicando o número da figura e o nome do arquivo.



Boa prática de programação 2.1

Use comentários para esclarecer conceitos difíceis usados em um programa.

O comentário que inicia com **//** é chamado de *comentário de uma única linha* porque o comentário termina no fim da linha atual. Observe que o comentário **//** pode iniciar no meio de uma linha e continuar até o final dessa linha.

```

1 // Fig. 2.1: Welcomel.java
2 // Um primeiro programa em Java
3
4 public class Welcomel {
5
6     // o método main inicia a execução do aplicativo Java
7     public static void main( String args[] )
8     {
9         System.out.println( "Welcome to Java Programming!" );
10    } // fim do método main
11
12 } // fim da classe Welcomel

```

Welcome to Java Programming!

Fig. 2.1 Um primeiro programa em Java.

Os comentários de múltiplas linhas podem ser escritos de duas outras maneiras. Por exemplo,

```
/* Esse é um comentário de
múltiplas linhas. Ele pode ser
dividido em várias linhas */
```

é um comentário que pode se estender por várias linhas. Este tipo de comentário inicia com o delimitador `/*` e termina com o delimitador `*/`; este tipo de comentário pode ser chamado de *comentário de múltiplas linhas*. Todo o texto entre os delimitadores do comentário é ignorado pelo compilador. Uma forma semelhante de comentário, chamada de *comentário de documentação*, é delimitada por `/**` e `*/`.



Erro comum de programação 2.1

Esquecer um dos delimitadores de um comentário de múltiplas linhas causa um erro de sintaxe.

Java absorveu os comentários delimitados com `/*` e `*/` da linguagem de programação C, e os comentários de uma única linha delimitados com `//` da linguagem de programação C++. Os programadores de Java geralmente preferem comentários de uma única linha no estilo de C++ a comentários no estilo de C. Por todo este livro, utilizamos comentários de uma única linha no estilo de C++. A sintaxe de comentário de documentação (`/**` e `*/`) é especial para Java. Ela permite embutir a documentação dos programas diretamente nos programas. O programa utilitário `javadoc` (fornecido pela Sun Microsystems com o Java 2 Software Development Kit) lê aqueles comentários do programa e usa os mesmos para preparar a documentação do seu programa. Há questões sutis quanto ao uso correto de comentários no estilo de `javadoc` em um programa. Não utilizamos comentários no estilo de `javadoc` neste livro. Entretanto, os comentários no estilo de `javadoc` serão explicados a fundo no Apêndice F.

A linha 2

```
// Um primeiro programa em Java
```

é um comentário de uma única linha que descreve o propósito do programa.



Boa prática de programação 2.2

Todo programa deve iniciar com um comentário que descreve o propósito do programa.

A linha 3 é simplesmente uma linha em branco. Os programadores usam linhas em branco e caracteres de espaçoamento para tornar os programas mais fáceis de ler. Juntos, as linhas em branco, os caracteres de espaço e os caracteres de tabulação são conhecidos como *espaço em branco* (os caracteres de espaço e as tabulações são conhecidos especificamente como *caracteres de espaço em branco*). Esses caracteres são ignorados pelo compilador. Discutimos convenções para utilizar caracteres de espaço em branco neste capítulo e nos próximos capítulos, pois essas convenções são necessárias em muitos programas em Java.



Boa prática de programação 2.3

Utilize linhas em branco, caracteres de espaço e caracteres de tabulação para melhorar a legibilidade do programa.

A linha 4

```
public class Welcome1 {
```

inicia uma *definição de classe* para a classe **Welcome1**. Cada programa Java consiste em pelo menos uma definição de classe que é definida por você – o programador. Essas classes são conhecidas como *classes definidas pelo programador ou classes definidas pelo usuário*. A palavra-chave **class** introduz uma definição de classe em Java e é imediatamente seguida pelo *nome de classe* (**Welcome1** nesse programa). As palavras-chave (ou *palavras reservadas*) são reservadas para uso por Java (discutiremos as várias palavras-chave em todo o texto) e sempre são escritas com todas as letras minúsculas. A lista completa das palavras-chave de Java é mostrada na Fig. 4.2.

Por convenção, todos os nomes de classe em Java iniciam com uma letra maiúscula e têm uma letra maiúscula para cada palavra no nome de classe (por exemplo, **ExemploDeNomeDeClasse**). O nome da classe é chamado de *identificador*. O identificador é uma série de caracteres que consistem em letras, dígitos, sublinhados (_) e sinais de cifrão (\$) que não iniciem com um dígito e não contenham nenhum espaço. Alguns identificadores válidos são **Welcome1**, **\$value**, **_value**, **m_inputField1** e **button7**. O nome **7button** não é um identificador válido porque inicia com um dígito e o nome **input field** não é um identificador válido porque contém um espaço. Java faz *distinção entre letras maiúsculas e minúsculas* – letras minúsculas e maiúsculas são diferentes, assim **a1** e **A1** são identificadores diferentes.



Erro comum de programação 2.2

Java diferencia letras maiúsculas de minúsculas. O uso incorreto de letras minúsculas e maiúsculas para um identificador é normalmente um erro de sintaxe.



Boa prática de programação 2.4

Por convenção, você deve sempre iniciar um nome de classe com uma letra maiúscula.



Boa prática de programação 2.5

Ao ler um programa em Java, procure identificadores que iniciem com as primeiras letras em maiúsculas. Estes identificadores normalmente representam classes Java.



Observação de engenharia de software 2.1

Evite utilizar identificadores que contêm sinais de cifrão (\$), porque estes são freqüentemente utilizados pelo compilador para criar nomes de identificadores.

Do Capítulo 2 ao Capítulo 7, cada classe que definimos inicia com a palavra-chave **public**. Por enquanto, só exigiremos essa palavra-chave. A palavra-chave **public** é discutida em detalhes no Capítulo 8. Também naquele capítulo, discutimos classes que não iniciam com a palavra-chave **public**. [Nota: várias vezes, no início deste texto, pedimos para você simular certos recursos em Java que introduzimos à medida que você escreve seus próprios programas em Java. Fazemos isso especificamente quando ainda não é importante conhecer todos os detalhes de um recurso para que você possa utilizar esse recurso em Java. Todos os programadores inicialmente aprendem a programar simulando o que outros programadores fizeram antes deles. Para cada detalhe que pedimos que você simule, indicamos onde será apresentada a discussão completa sobre o assunto.]

Quando você salva sua definição de classe **public** em um arquivo, o nome do arquivo deve ser o nome da classe seguido pela extensão de nome de arquivo “**.java**”. Para nosso aplicativo, o nome de arquivo é **Welcome1.java**. Todas as definições de classe Java são armazenadas em arquivos que terminam com a extensão de nome de arquivo “**.java**”.



Erro comum de programação 2.3

*Para uma classe **public**, é um erro se o nome de arquivo não for idêntico ao nome de classe (mais a extensão **.java**), tanto em termos de ortografia quanto do uso de letras maiúsculas e minúsculas. Portanto, é também um erro que um arquivo contenha duas ou mais classes **public**.*

Erro comum de programação 2.4

É um erro não terminar o nome de arquivo com a extensão `.java` para um arquivo que contém uma definição de classe do aplicativo. Se não houver extensão, o compilador Java não será capaz de compilar a definição de classe.

A chave esquerda (no final da linha 4), `{`, inicia o corpo de cada definição de classe. A chave direita correspondente (na linha 13 nesse programa), `}`, deve terminar cada definição de classe. Observe que as linhas 6 a 11 estão recuadas. Esse recuo é uma das convenções de espaçamento mencionadas anteriormente. Definimos cada convenção de espaçamento como uma *Boa prática de programação*.

Boa prática de programação 2.6

Sempre que você digita uma chave esquerda de abertura, `{`, em seu programa, imediatamente digite a chave direita de fechamento, `}`, e depois reposicione o cursor entre as chaves para começar a digitação do corpo. Isso ajuda a evitar erros causados pela falta de uma chave.

Boa prática de programação 2.7

Recue o corpo inteiro de cada definição de classe um “nível” de recuo entre a chave esquerda, `{`, e a chave direita, `}`, que definem o corpo da classe. Este formato enfatiza a estrutura da definição de classe e ajuda a tornar a definição de classe mais fácil de ler.

Boa prática de programação 2.8

Configure uma convenção para o tamanho de recuo que você prefere e depois aplique uniformemente essa convenção. A tecla Tab pode ser utilizada para criar recuos, mas as paradas de tabulação podem variar entre os editores. Recomendamos utilizar três espaços para formar um nível de recuo.

Erro comum de programação 2.5

Se as chaves não ocorrem em pares correspondentes, o compilador indica um erro.

A linha 5 é uma linha em branco, inserida para melhorar a legibilidade do programa. A linha 6,

```
// o método main inicia a execução do aplicativo Java
```

é um comentário de uma única linha que indica a finalidade das linhas 6 a 11 do programa.

A linha 7,

```
public static void main( String args[] )
```

faz parte de todo aplicativo Java. Os aplicativos Java começam a execução por `main`. Os parênteses depois de `main` indicam que `main` é um bloco de construção de programa denominado *método*. As definições de classe de Java normalmente contêm um ou mais métodos. Para uma classe de aplicativo Java, exatamente um desses métodos deve ser chamado de `main` e deve ser definido como mostrado na linha 7; caso contrário, o interpretador `java` não executará o aplicativo. Os métodos podem realizar tarefas e retornar informações quando completam suas tarefas. A palavra-chave `void` indica que esse método realizará uma tarefa (exibindo uma linha de texto nesse programa), mas não retornará nenhuma informação quando completar sua tarefa. Mais tarde, veremos que muitos métodos retornam informações quando completam sua tarefa. Os métodos são explicados em detalhes no Capítulo 6. Por enquanto, simplesmente copie a primeira linha do `main` em cada um de seus aplicativos Java.

A chave esquerda, `{`, na linha 8, inicia o *corpo da definição de método*. A chave direita correspondente, `}`, deve terminar o corpo da definição do método (a linha 11 do programa). Observe que a linha no corpo do método (linha 7) está recuada entre essas chaves.

Boa prática de programação 2.9

Recue o corpo inteiro de cada definição de método um “nível” de recuo entre a chave esquerda, `{`, e a chave direita, `}`, que definem o corpo do método. Este formato faz a estrutura do método se destacar e ajuda a tornar a definição do método mais fácil de ler.

A linha 9

```
System.out.println( "Welcome to Java Programming!" );
```

instrui o computador a realizar uma *ação*, a saber, imprimir o *string* de caracteres contido entre as aspas duplas. O *string* é chamado, às vezes, de *string de caractere*, de *mensagem* ou de *string literal*. Nós nos referimos a caracteres entre aspas duplas genericamente como *strings*. Os caracteres de espaço em branco em *strings* não são ignorados pelo compilador.

System.out é conhecido como *objeto de saída padrão*. **System.out** permite exibir *strings* e outros tipos de informações na *janela de comando* a partir da qual o aplicativo Java é executado. No Microsoft Windows 95/98/ME, a janela de comando é o *prompt do MS-DOS*. No Microsoft Windows NT/2000, a janela de comando é o *prompt de comando (cmd.exe)*. Em UNIX, a janela de comando normalmente é chamada de *janela de comando, ferramenta de comando, ferramenta de shell ou shell*. Em computadores que executam um sistema operacional que não tem uma janela de comando (como um Macintosh), o interpretador **java** normalmente exibe uma janela que contém as informações exibidas pelo programa.

O método **System.out.println** exibe (ou imprime) uma linha de texto na janela de comando. Quando **System.out.println** completa sua tarefa, automaticamente posiciona o *cursor de saída* (a localização na qual o próximo caractere será exibido) no início da próxima linha na janela de comando. (Este movimento do cursor é semelhante ao pressionamento da tecla *Enter* quando se digita em um editor de texto – o cursor aparece no início da próxima linha em seu arquivo).

A linha inteira, incluindo **System.out.println**, seu argumento entre parênteses (o *string*) e o ponto-e-vírgula (;), é uma *instrução*. Cada instrução deve terminar com um ponto-e-vírgula (também conhecido como *terminador de instrução*). Quando a instrução na linha 9 de nosso programa é executada, ela exibe a mensagem **Welcome to Java Programming!** na janela de comando.



Erro comum de programação 2.6

Omitir o ponto-e-vírgula no final de uma instrução é um erro de sintaxe. Ocorre um erro de sintaxe quando o compilador não consegue reconhecer uma instrução. O compilador normalmente emite uma mensagem de erro para ajudar o programador a identificar e corrigir a instrução incorreta. Os erros de sintaxe são violações das regras de linguagem. Eles são também chamados de erros de compilação ou erros durante a compilação, porque o compilador os detecta durante a fase de compilação. Você será incapaz de executar seu programa enquanto não corrigir todos os erros de sintaxe nele existentes.



Dica de teste e depuração 2.1

Quando o compilador acusa um erro de sintaxe, o erro pode não estar na linha cujo número é indicado pela mensagem de erro. Primeiro, verifique a linha para a qual o erro foi informado. Se essa linha não contiver erros de sintaxe, verifique as várias linhas precedentes no programa.

Alguns programadores acham difícil, ao ler e/ou escrever um programa, associar as chaves esquerdas e direitas ({ e }) que delimitam o corpo de uma definição de classe ou de uma definição de método. Por essa razão, alguns programadores preferem incluir um comentário de uma única linha depois de uma chave de fechamento (}) que termina uma definição de método e depois de uma chave direita de fechamento que finaliza uma definição de classe. Por exemplo, a linha 11,

```
} // fim do método main()
```

especifica a chave direita de fechamento (}) do método **main** e a linha 13,

```
} // fim da classe Welcome1
```

especifica a chave direita de fechamento (}) da classe **Welcome1**. Cada comentário indica o método ou a classe que a chave direita termina. Usamos comentários como estes até o Capítulo 6, para ajudar os programadores principiantes a determinar onde termina cada componente do programa. Após o Capítulo 6, usamos comentários como estes quando os pares de chaves contêm muitos comandos, o que torna difícil identificar as chaves de fechamento.



Boa prática de programação 2.10

Alguns programadores preferem seguir a chave direita de fechamento (}) de um corpo de método ou definição de classe com um comentário de uma única linha que indica o método ou a definição de classe à qual a chave pertence. Este comentário melhora a legibilidade do programa.

2.2.1 Compilando e executando seu primeiro aplicativo Java

Agora estamos prontos para compilar e executar nosso programa. Para compilar o programa, abrimos uma janela de comando, mudamos para o diretório onde o programa é armazenado e digitamos

```
javac Welcome1.java
```

Se o programa não contiver erros de sintaxe, o comando precedente criará um novo arquivo chamado **Welcome1.class** contendo os *bytecodes* de Java que representam nosso aplicativo. Esses *bytecodes* serão interpretados pelo interpretador **java** quando lhe dissermos para executar o programa, como mostrado no **Command Prompt** do Microsoft Windows 2000 da Fig. 2.2.

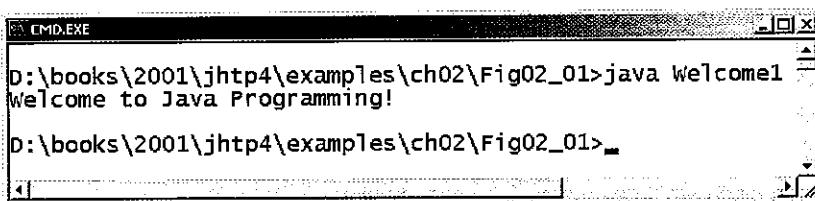


Fig. 2.2 Executando **Welcome1** em um **Command Prompt** do Microsoft Windows 2000.

No *prompt* de comando da Fig. 2.2, digitamos

```
java Welcome1
```

para disparar a execução do interpretador Java e indicar que ele deve carregar o arquivo “**.class**” para a classe **Welcome1**. Observe que a extensão do nome de arquivo “**.class**” é omitida no comando precedente; caso contrário o interpretador não executaria o programa. O interpretador chama automaticamente o método **main**. Em seguida, a instrução na linha 9 de **main** exibe “**Welcome to Java Programming!**”.



Dica de teste e depuração 2.2

O compilador Java gera mensagens de erro de sintaxe quando a sintaxe de um programa está incorreta. Quando você está aprendendo a programar, às vezes é útil “estragar” um programa que está funcionando, de modo que você possa ver as mensagens de erro exibidas pelo compilador. Então, quando você encontrar aquela mensagem de erro novamente, você terá uma ideia da causa do erro. Tente remover um ponto-e-vírgula ou uma chave do programa da Fig. 2.1, depois recompile o programa para ver as mensagens de erro geradas pela omissão.

2.3 Modificando nosso primeiro programa em Java

Esta seção continua nossa introdução à programação em Java com dois exemplos que modificam o exemplo da Fig. 2.1 para imprimir o texto em uma linha usando múltiplas instruções e para imprimir o texto em múltiplas linhas usando uma única instrução.

2.3.1 Exibindo uma única linha de texto com múltiplas instruções

Welcome to Java Programming! pode ser exibido com diversos métodos. A classe **Welcome2**, mostrada na Fig. 2.3, utiliza duas instruções para produzir a mesma saída que aquela mostrada na Fig. 2.1.

A maior parte do programa é idêntica àquele da Fig. 2.1, de modo que aqui discutiremos somente as mudanças. A linha 2,

```
// Imprimindo uma linha com múltiplas instruções
```

é um comentário de uma única linha descrevendo a finalidade deste programa. A linha 4 inicia a definição da classe **Welcome2**.

As linhas 9 e 10 do método **main**

```
System.out.print( "Welcome to " );
System.out.println( "Java Programming!" );
```

```

1 // Fig. 2.3: Welcome2.java
2 // Imprimindo uma linha com múltiplas instruções
3
4 public class Welcome2 {
5
6     // o método main inicia a execução do aplicativo Java
7     public static void main( String args[] ) {
8         {
9             System.out.print( "Welcome to " );
10            System.out.println( "Java Programming!" );
11        } // fim do método main
12    } // fim da classe Welcome2

```

Welcome to Java Programming!

Fig. 2.3 Imprimindo em uma mesma linha com múltiplas instruções.

exibem uma linha na janela de comando. A primeira instrução utiliza o método `print` de `System.out` para exibir um *string*. A diferença entre `println` e `print` é que, depois de exibir seu argumento, `print` não posiciona o cursor de saída no início da próxima linha na janela de comando; o próximo caractere que o programa exibe na janela de comando aparecerá imediatamente após o último caractere que `print` exibe. Portanto, a linha 10 posiciona o primeiro caractere em seu argumento, “J”, imediatamente depois do último caractere que a linha 9 exibe (o caractere de espaço no fim do *string* na linha 9). Cada instrução `print` ou `println` retoma a exibição dos caracteres a partir de onde o último `print` ou `println` parou de exibir os caracteres.

2.3.2 Exibindo múltiplas linhas de texto com uma única instrução

Uma única instrução pode exibir múltiplas linhas utilizando *caracteres de nova linha*. Os caracteres de nova linha são “caracteres especiais” que indicam para os métodos `print` e `println` de `System.out` quando eles devem posicionar o cursor de saída no início da próxima linha na janela de comando. A Figura 2.4 envia quatro linhas de texto para a saída, usando caracteres de nova linha para se determinar quando começar cada nova linha.

A maior parte do programa é idêntica àquele das Figs. 2.1 e 2.3, de modo que discutimos aqui somente as alterações. A linha 2,

```
// Imprimindo múltiplas linhas com uma única instrução
```

é um comando de uma única linha descrevendo a finalidade deste programa. A linha 4 começa a definição da classe `Welcome3`.

A linha 9,

```
System.out.println( "Welcome\n to \nJava\n Programming!" );
```

exibe quatro linhas de texto separadas na janela de comando. Normalmente, os caracteres em um *string* são exibidos exatamente como aparecem entre as aspas duplas. Observe, entretanto, que os dois caracteres “\” e “\n” não estão impressos na tela. A *barra invertida* (\) é chamada *caractere de escape*. Ela indica que um caractere “especial” deve ser enviado para a saída. Quando uma barra invertida aparece em um *string* de caracteres, Java combina o próximo caractere com a barra invertida para formar uma *seqüência de escape*. A seqüência de escape \n é o *caractere de nova linha*. Quando um caractere nova linha aparece no *string* que está sendo enviado para a saída com `System.out`, o caractere nova linha faz com que o cursor de saída na tela se move para o começo da próxima linha na janela de comando. Algumas outras seqüências de escape comuns são listadas na Fig. 2.5.

2.4 Exibindo texto em uma caixa de diálogo

Embora os primeiros programas deste livro exibam a saída na janela de comando, a maioria dos aplicativos em Java que exibem a saída utilizam janelas ou *caixas de diálogo* (também chamadas de *diálogos*) para exibir a saída. Por

```

1 // Fig. 2.4: Welcome3.java
2 // Imprimindo múltiplas linhas de texto com uma única instrução
3
4 public class Welcome3 {
5
6     // o método main inicia a execução do aplicativo Java
7     public static void main( String args[] )
8     {
9         System.out.println( "Welcome\n to\n Java\n Programming!" );
10    } // fim do método main
11
12 } // fim da classe Welcome3

```

```

Welcome
to
Java
Programming!

```

Fig. 2.4 Imprimindo em múltiplas linhas de texto com uma única instrução.

Seqüência de escape Descrição

\n	Nova linha. Posiciona o cursor de tela no início da próxima linha.
\t	Tabulação horizontal. Move o cursor de tela para a próxima parada de tabulação.
\r	Retorno de carro. Posiciona o cursor de tela no início da linha atual; não avança para a próxima linha. Qualquer saída de caracteres depois do retorno de carro sobrescreve a saída anterior de caracteres na linha atual.
\\\	Barra invertida. Utilizada para imprimir um caractere barra invertida.
\"	Aspas duplas. Utilizada para imprimir um caractere aspas duplas. Por exemplo,
	<code>System.out.println("\"entre aspas\"");</code>
	exibe
	"entre aspas"

Fig. 2.5 Algumas seqüências comuns de escape.

exemplo, os navegadores da World Wide Web como o Netscape Communicator ou o Microsoft Internet Explorer exibem páginas da Web em suas próprias janelas. Os programas de correio eletrônico em geral permitem que você digite e leia mensagens em uma janela. Em geral, as caixas de diálogo são janelas nas quais os programas exibem mensagens importantes para o usuário do programa. A classe de Java *JOptionPane* oferece caixas de diálogo predefinidas que permitem aos programas exibir mensagens para os usuários. A Fig. 2.6 exibe o mesmo *string* que a Fig. 2.4 em uma caixa de diálogo predefinida e conhecida como *diálogo de mensagem*.

Um dos pontos fortes de Java é seu rico conjunto de classes predefinidas, as quais os programadores podem reutilizar em vez de “reinventar a roda”. Utilizamos muitas dessas classes ao longo do livro. As inúmeras classes predefinidas de Java são agrupadas em categorias de classes relacionadas chamadas *pacotes*. Os pacotes são conhecidos coletivamente como *biblioteca de classes Java* ou *interface de programação de aplicativos Java (Java applications programming interface – Java API)*. Os pacotes da Java API estão divididos em pacotes do núcleo e pacotes de extensões. Os nomes dos pacotes começam ou com “java” (pacotes do núcleo) ou “javax” (pacotes de extensão). Muitos dos pacotes do núcleo e dos pacotes de extensão estão incluídos como parte do Java 2 Development

Kit. Damos uma visão geral destes pacotes incluídos no Capítulo 6. À medida que Java continua a evoluir, novos pacotes são desenvolvidos como pacotes de extensão. Estas extensões freqüentemente podem ser baixadas da página java.sun.com e usadas para melhorar os recursos de Java. Neste exemplo, usamos a classe `JOptionPane`, que Java define no pacote `javax.swing`.

A linha 4,

```
// Pacotes de extensão de Java
```

é um comentário de uma só linha que indica a seção do programa na qual especificamos as instruções `import` para as classes em pacotes de extensão de Java. Em todo o programa que especifica as instruções `import`, separamos as instruções de importação nos seguintes grupos: pacotes do núcleo de Java (para os nomes de pacote que começam com `java`), pacotes de extensão de Java (para os nomes de pacotes que começam com `javax`) e pacotes Deitel (para nossos próprios pacotes, definidos mais adiante no livro).

A linha 5,

```
import javax.swing.JOptionPane; // importa a classe JOptionPane
```

é uma instrução `import`. O compilador utiliza instruções `import` para identificar e carregar classes usadas em um programa Java. Quando você utiliza classes da Java API, o compilador tenta assegurar que você as utiliza corretamente. As instruções `import` ajudam o compilador a localizar as classes que você pretende utilizar. Para cada nova classe da Java API que usamos, indicamos o pacote no qual você pode encontrar aquela classe. Esta informação sobre os pacotes é importante. Ela o ajuda a localizar as descrições de cada pacote e de cada classe na *documentação da Java API*. Uma versão baseada na Web desta documentação pode ser encontrada na página

java.sun.com/j2se/1.3/docs/api/index.html

Além disso, você pode baixar esta documentação para seu próprio computador da página

java.sun.com/j2se/1.3/docs.html

```

1 // Fig. 2.6: Welcome4.java
2 // Imprimindo múltiplas linhas em uma caixa de diálogo
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane; // importa a classe JOptionPane
6
7 public class Welcome4 {
8
9     // método main começa execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        JOptionPane.showMessageDialog(
13            null, "Welcome\nto\nJava\nProgramming!" );
14
15        System.exit( 0 ); // termina aplicativo
16
17    } // fim do método main
18
19 } // fim da classe Welcome4

```

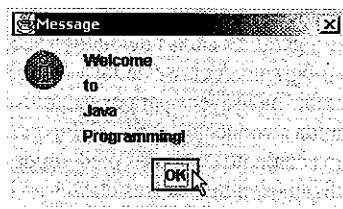


Fig. 2.6 Exibindo múltiplas linhas em uma caixa de diálogo.

Vamos fornecer uma visão geral do uso desta documentação com os recursos e programas para baixar para *Java Como Programar, Quarta Edição*, em nosso site, www.deitel.com. Os pacotes são discutidos em detalhes no Capítulo 8.



Erro comum de programação 2.7

Todas as instruções `import` devem aparecer antes da definição de classe. Colocar uma instrução `import` dentro do corpo de definição de uma classe ou depois de uma definição de classe é um erro de sintaxe.

A linha 5 diz ao compilador para carregar a classe `JOptionPane` do pacote `javax.swing`. Esse pacote contém muitas classes que ajudam os programadores de Java a definir *interfaces gráficas com o usuário (GUIs – graphical user interfaces)* para seu aplicativo. Os *componentes GUI* facilitam a entrada de dados pelo usuário de seu programa e a formatação ou apresentação de dados de saída para o usuário de seu programa. Por exemplo, a Fig. 2.7 contém uma janela do Netscape Navigator. Na janela, há uma barra contendo *menus* (`File`, `Edit`, `View`, etc.), chamada de *barra de menu*. Abaixo da barra de menus há um conjunto de *botões* e cada um deles tem uma tarefa definida no Netscape Navigator. Abaixo dos botões, há um *campo de texto*, em que o usuário pode digitar o nome do *site* da World Wide Web a visitar. Menus, botões, campos de texto e rótulos fazem parte da GUI do Netscape Navigator. Eles permitem que você interaja com o programa. Java contém classes que implementam os componentes GUI descritos aqui e outros que serão descritos no Capítulo 12 e no Capítulo 13.

No método `main` da Fig. 2.6, as linhas 12 e 13

```
JOptionPane.showMessageDialog(
    null, "Welcome\\nto\\nJava\\nProgramming!" );
```

indicam uma chamada para o método `showMessageDialog` da classe `JOptionPane`. O método exige dois argumentos. Quando um método exige múltiplos argumentos, os argumentos são separados por vírgulas (,). Até discutirmos `JOptionPane` em detalhes no Capítulo 13, o primeiro argumento sempre será a palavra-chave `null`. O segundo argumento é o *string* a exibir. O primeiro argumento ajuda o aplicativo Java a determinar onde posicionar a caixa de diálogo. Quando o primeiro argumento é `null`, a caixa de diálogo aparece no centro da tela do computador. A maioria dos aplicativos que você usa em seu computador é executada em sua própria janela (p. ex., programas de correio eletrônico), navegadores e processadores de texto). Quando uma destas aplicações exibe uma caixa de diálogo, ela normalmente aparece no centro da janela da aplicação, que não é necessariamente o centro da tela. Mais adiante neste livro, você verá aplicativos mais elaborados nos quais o primeiro argumento para o método `showMessageDialog` fará a caixa de diálogo aparecer no centro da janela do aplicativo, em vez de no centro da tela.

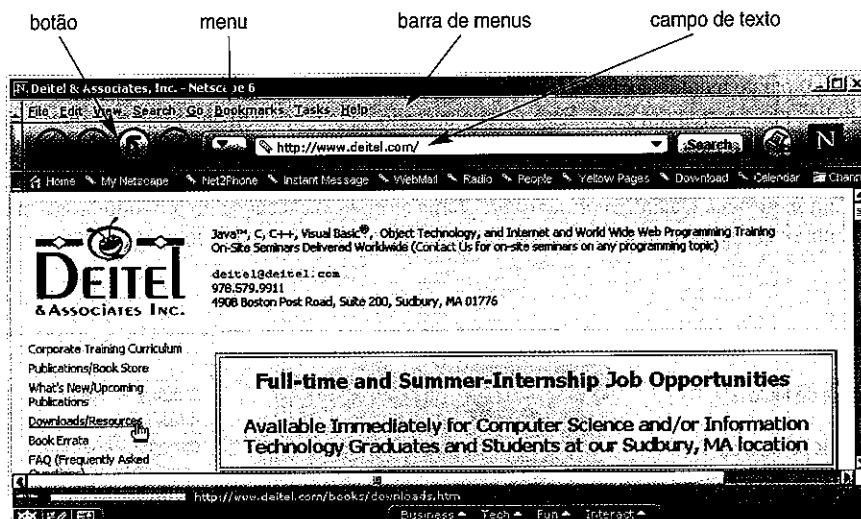


Fig. 2.7 Uma janela de exemplo do Netscape Navigator com os componentes GUI.



Boa prática de programação 2.11

Coloque um espaço depois de cada vírgula (,) em uma lista de argumentos para tornar os programas mais legíveis.

O método `JOptionPane.showMessageDialog` é um método especial da classe `JOptionPane` e é chamado de **método static**. Tais métodos sempre são chamados utilizando-se seu nome de classe seguido por um operador ponto (.) e pelo nome do método, como em

`NomeClasse.nomeMétodo(argumentos)`

Muitos dos métodos predefinidos que apresentamos no início deste livro são métodos **static**. Pedimos que você imite esta sintaxe para chamar métodos **static** até que os discutamos em detalhes no Capítulo 8.

A execução da instrução nas linhas 12 e 13 exibe a caixa de diálogo mostrada na Fig. 2.8. A *barra de título* do diálogo contém o *string Message* para indicar que o diálogo está apresentando uma mensagem para o usuário. A caixa de diálogo automaticamente inclui um botão **OK** que permite ao usuário *fechar (ocultar) o diálogo* pressionando o botão. Isso é realizado posicionando-se o *cursor do mouse* (também chamado de *ponteiro do mouse*) sobre o botão **OK** e clicando com o botão esquerdo do mouse.

Lembre-se de que todas as instruções em Java terminam com um ponto-e-vírgula (;). Portanto, as linhas 12 e 13 representam uma instrução. Java permite que instruções grandes sejam divididas em muitas linhas. Entretanto, você não pode dividir uma instrução no meio de um identificador ou no meio de um *string*.



Erro comum de programação 2.8

Dividir uma instrução no meio de um identificador ou de um string é um erro de sintaxe.

A linha 15

```
System.exit( 0 ); // termina o aplicativo
```

utiliza o método **static exit** da classe `System` para terminar o aplicativo. Em qualquer aplicativo que exibe uma interface gráfica com o usuário, esta linha é necessária para terminar o aplicativo. Observe mais uma vez a sintaxe utilizada para chamar o método – o nome da classe (`System`), um ponto (.) e o nome do método (`exit`). Lembre-se de que os identificadores que iniciam com a primeira letra maiúscula normalmente representam nomes de classe. Então, você pode supor que `System` é uma classe. A classe `System` faz parte do pacote `java.lang`. Observe que a classe `System` não é importada com uma instrução `import` no início do programa. Por *default*, o pacote `java.lang` é importado em todos os programas Java. O pacote `java.lang` é o único pacote na Java API para o qual você não precisa especificar uma instrução `import`.

O argumento 0 para o método `exit` indica que o aplicativo terminou com sucesso (um valor diferente de zero normalmente indica que ocorreu um erro). Esse valor é passado à janela de comando que executou o programa. Isso é útil se o programa é executado a partir de um arquivo de lote (em sistemas Windows 95/98/ME/NT/2000) ou de um *script de shell* (em sistemas UNIX/Linux). Os arquivos de lote e os scripts de *shell* freqüentemente executam vários programas em sequência. Quando o primeiro programa termina, a execução do próximo programa inicia automaticamente. É possível usar o argumento para o método `exit` em um arquivo de lote ou *script de shell* para deter-

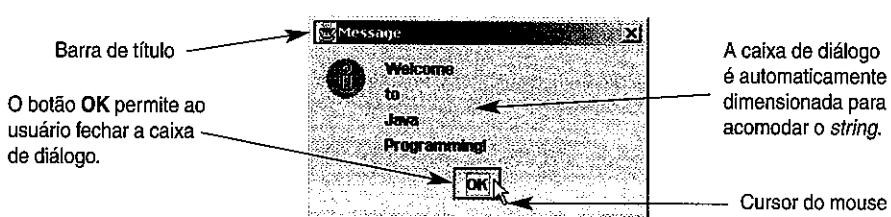


Fig. 2.8 Caixa de diálogo de mensagem.

minar se outros programas devem ser executados. Para obter mais informações sobre arquivos de lote ou *scripts* de *shell*, veja a documentação do seu sistema operacional.

Erro comum de programação 2.9



Esquecer de chamar `System.exit` em um aplicativo que exibe uma interface gráfica com o usuário impede o programa de terminar de forma adequada. Esta omissão normalmente faz com que você fique impedido de digitar quaisquer outros comandos na janela de comando. O Capítulo 14 discute em mais detalhes as razões pelas quais `System.exit` é obrigatório em aplicativos baseados em GUI.

2.5 Outro aplicativo Java: adicionando inteiros

Nosso próximo aplicativo lê dois *inteiros* (números inteiros, como -22, 7 e 1024) digitados pelo usuário no teclado, calcula a soma desses valores e exibe o resultado. Este programa utiliza outra caixa de diálogo predefinida da classe `JOptionPane`, chamada de *diálogo de entrada*, que permite ao usuário fornecer um valor para utilização no programa. O programa também utiliza um diálogo de mensagem para exibir os resultados da adição dos inteiros. A Figura 2.9 mostra o aplicativo e as reproduções de telas de exemplo.

```

1 // Fig. 2.9: Addition.java
2 // Um programa de adição
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane; // importa a classe JOptionPane
6
7 public class Addition {
8
9     // método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        String firstNumber, // primeiro string inserido pelo usuário
13        secondNumber; // segundo string inserido pelo usuário
14        int number1, // primeiro número a somar
15        int number2, // segundo número a somar
16        int sum; // soma de number1 e number2
17
18        // lê o primeiro número do usuário como um string
19        firstNumber =
20            JOptionPane.showInputDialog( "Enter first integer" );
21
22        // lê o segundo número do usuário como um string
23        secondNumber =
24            JOptionPane.showInputDialog( "Enter second integer" );
25
26        // converte os números do tipo String para o tipo int
27        number1 = Integer.parseInt( firstNumber );
28        number2 = Integer.parseInt( secondNumber );
29
30        // adiciona os números
31        sum = number1 + number2;
32
33        // exibe os resultados
34        JOptionPane.showMessageDialog(
35            null, "The sum is " + sum, "Results",
36            JOptionPane.PLAIN_MESSAGE );
37
38        System.exit( 0 ); // termina o aplicativo

```

Fig. 2.9 Um programa de adição “em ação” (parte 1 de 2).

```

39
40 } // fim do método main
41
42 } // fim da classe Addition

```

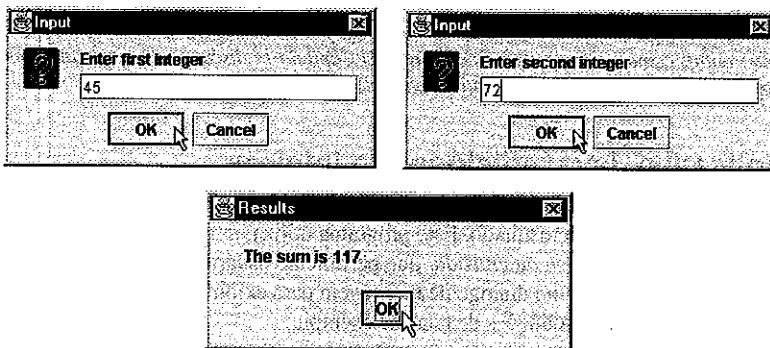


Fig. 2.9 Um programa de adição “em ação” (parte 2 de 2).

As linhas 1 e 2

```

// Fig. 2.9: Addition.java
// Um programa de adição

```

são comentários de uma única linha declarando o número da figura, o nome do arquivo e o propósito do programa.

A linha 4

```
// Pacotes de extensão de Java
```

é um comentário de uma única linha que indica que a próxima linha importa uma classe dos pacotes de extensão de Java.

A linha 5

```
import javax.swing.JOptionPane; // importa a classe JOptionPane
```

indica que o compilador deve carregar a classe **JOptionPane** para utilizar nesse aplicativo.

Como afirmado anteriormente, cada programa Java consiste em pelo menos uma definição de classe. A linha 7

```
public class Addition {
```

inicia as definições da classe **Addition**. O nome de arquivo para essa classe **public** deve ser **Addition.java**.

Lembre-se de que todas as definições de classe iniciam com uma chave esquerda de abertura (no fim da linha 7), {, e terminam com uma chave direita de fechamento, } (na linha 42).

Como declarado anteriormente, cada aplicativo inicia a execução com o método **main** (linhas 10 a 40). A chave esquerda (linha 11) marca o começo do corpo de **main** e a chave direita correspondente (linha 40) marca o fim do corpo de **main**.

As linhas 12 e 13

```
String firstNumber; // primeiro string inserido pelo usuário
String secondNumber; // segundo string inserido pelo usuário
```

são *declarações*. As palavras **firstNumber** e **secondNumber** são nomes de *variáveis*. A variável é uma posição na memória do computador na qual um valor pode ser armazenado para utilização por um programa. Todas as variáveis devem ser declaradas com um nome e um tipo de dado antes de poderem ser utilizadas em um programa. Essa declaração especifica que as variáveis **firstNumber** e **secondNumber** são dados do tipo **String** (localizado no pacote **java.lang**), o que significa que as variáveis armazenarão *strings*. Um nome de variável pode ser qualquer identificador válido. Assim como instruções, as declarações terminam com um ponto-e-vírgula (;). Observe os comentários de uma só linha no fim de cada linha. Este uso e posicionamento dos comentários é uma prática comum usada pelos programadores para indicar a finalidade de cada variável no programa.



Boa prática de programação 2.12

Escolher nomes de variáveis significativos ajuda um programa a ser autodocumentado (isto é, torna mais fácil entender um programa simplesmente lendo-o em vez de ter de ler o manual ou utilizar comentários excessivos).



Boa prática de programação 2.13

Por convenção, os identificadores de nome de variável começam com letra minúscula. Como ocorre com os nomes de classe, toda palavra no nome depois da primeira palavra deve iniciar com uma letra maiúscula. Por exemplo, o identificador `firstNumber` tem um N maiúsculo em sua segunda palavra, `Number`.



Boa prática de programação 2.14

Alguns programadores preferem declarar cada variável em uma linha separada. Esse formato permite a inserção fácil de um comentário descritivo ao lado de cada declaração.



Observação de engenharia de software 2.2

Java importa automaticamente classes do pacote `java.lang`, como a classe `String`. Portanto, não são necessárias instruções `import` para as classes que estão no pacote `java.lang`.

As declarações podem ser divididas ao longo de várias linhas, com cada variável na declaração separada por uma vírgula (isto é, uma lista separada por vírgulas de nomes de variável). Várias variáveis do mesmo tipo podem ser declaradas em uma declaração ou em múltiplas declarações. As linhas 12 e 13 também podem ser escritas como segue:

```
String firstNumber,      // primeiro string digitado pelo usuário
       secondNumber;    // segundo string digitado pelo usuário
```

As linhas 14 a 16,

```
int number1,           // primeiro número a somar
    int number2,         // segundo número a somar
    int sum;             // soma de number1 e number2
```

declararam que as variáveis `number1`, `number2` e `sum` são dados do tipo `int`, o que significa que essas variáveis armazenarão valores *inteiros* (isto é, números inteiros como 7, -11, 0, 31914). Em breve discutiremos os tipos de dados `float` e `double` para especificar números reais (números com casas decimais como 3,4, 0,0, -11,19) e variáveis do tipo `char`, para especificar dados formados por caracteres. Uma variável `char` pode armazenar apenas uma única letra minúscula, uma única letra maiúscula, um único dígito ou um único caractere especial (como `x`, `$`, `7` e `*`) e seqüências de escape (como o caractere de nova linha `\n`). Java também pode representar caracteres de muitos outros idiomas. Tipos como `int`, `double` e `char` são freqüentemente chamados de *tipos de dados primitivos* ou *tipos de dados embutidos*. Os nomes de tipos primitivos são palavras-chave e, portanto, devem ser formados apenas por letras minúsculas. O Capítulo 4 resume os oito tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`).

A linha 18 é um comentário de uma só linha que indica que a próxima instrução lê o primeiro número fornecido pelo usuário. As linhas 19 e 20

```
firstNumber =
JOptionPane.showInputDialog( "Enter first integer" );
```

lêem um `String` fornecido pelo usuário representando o primeiro dos dois inteiros a adicionar. O método `JOptionPane.showInputDialog` exibe o diálogo de entrada na Fig. 2.10.

O argumento para `showInputDialog` indica o que o usuário deve digitar no campo de texto. Essa mensagem é chamada de *prompt* porque direciona o usuário para uma ação específica. O usuário digita caracteres no campo de texto e, a seguir, clica no botão **OK** ou pressiona a tecla `Enter` para devolver o `string` para o programa. (Se você digitar e nada aparecer no campo de texto, posicione o ponteiro do mouse no campo de texto e clique no botão esquerdo do mouse para ativar o campo de texto.) Infelizmente, Java não fornece uma forma simples de entrada que seja análoga a exibir a saída na janela de comando com os métodos `print` e `println` de `System.out`. Por essa razão, normalmente recebemos entrada de um usuário por um componente GUI (um diálogo de entrada nesse programa).

Tecnicamente, o usuário pode digitar qualquer coisa no campo de texto da entrada. Nossa programa supõe que o usuário segue as instruções e digita um valor inteiro válido. Neste programa, se o usuário digitar um valor diferen-

te de um inteiro ou clicar no botão **Cancel** no diálogo de entrada, ocorrerá um erro de lógica durante a execução. O Capítulo 14 discute como tornar seus programas mais robustos capacitando-os a tratar esses erros. Isto também é conhecido como tornar seu programa *tolerante a falhas*.

O resultado do método `showInputDialog` de `JOptionPane` (um `String` que contém os caracteres digitados pelo usuário) é atribuído à variável `firstNumber` com o operador de atribuição `=`. A instrução (linhas 19 e 20) é lida como “`firstNumber` recebe o valor de `JOptionPane.showInputDialog("Enter first integer")`”. O operador `=` é chamado de *operador binário* porque tem dois *operandos* – `firstNumber` e o resultado da expressão `JOptionPane.showInputDialog("Enter first integer")`. Essa instrução inteira é chamada de *instrução de atribuição* porque atribui um valor a uma variável. A expressão do lado direito do operador de atribuição `=` é sempre avaliada primeiro. Neste caso, o programa chama o método `showInputDialog` e o valor digitado pelo usuário é atribuído a `firstNumber`.

A linha 22 é um comentário de uma só linha que indica que a próxima instrução lê o segundo número fornecido pelo usuário.

As linhas 23 e 24

```
secondNumber =
JOptionPane.showInputDialog( "Enter second integer" );
```

exibem um diálogo de entrada em que o usuário digita um `String` para representar o segundo dos dois inteiros a adicionar.

As linhas 27 e 28

```
number1 = Integer.parseInt( firstNumber );
number2 = Integer.parseInt( secondNumber );
```

converterem os dois `Strings` fornecidos pelo usuário em valores `int` que o programa pode utilizar em um cálculo. O método `Integer.parseInt` (um método `static` da classe `Integer`) converte seu argumento `String` em um inteiro. A classe `Integer` está definida no pacote `java.lang`. A linha 27 atribui à variável `number1` o valor `int` (inteiro) que `Integer.parseInt` devolve. A linha 28 atribui à variável `number2` o valor `int` (inteiro) que `Integer.parseInt` devolve.

A linha 31,

```
sum = number1 + number2;
```

é uma instrução de atribuição que calcula a soma das variáveis `number1` e `number2` e atribui o resultado à variável `sum` utilizando o operador de atribuição `=`. A instrução é lida como “`sum` recebe o valor de `number1 + number2`”. A maioria dos cálculos é realizada em instruções de atribuição. Quando o programa encontra a operação de adição, ele usa os valores armazenados nas variáveis `number1` e `number2` para executar o cálculo. Na instrução precedente, o operador de adição é um operador binário: seus dois operandos são `number1` e `number2`.

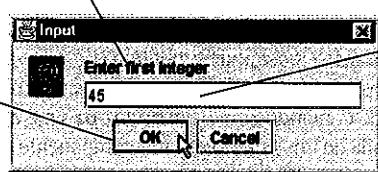


Boa prática de programação 2.15

Coloque espaços em ambos os lados de um operador binário. Isso destaca o operador e torna o programa mais legível.

Quando o usuário clica em **OK**, o **45** digitado pelo usuário é retornado para o programa como um `String`. O programa deve converter o `String` em um número.

Esse é o *prompt* para o usuário.



Esse é o campo de texto em que o usuário digita o valor.

Fig. 2.10 Caixa de diálogo de entrada.

Depois que o cálculo foi executado, as linhas 34 a 36

```
JOptionPane.showMessageDialog(
    null, "The sum is " + sum, "Results",
    JOptionPane.PLAIN_MESSAGE );
```

utilizam o método `JOptionPane.showMessageDialog` para exibir o resultado da adição. Esta nova versão do método `showMessageDialog` de `JOptionPane` exige quatro argumentos. Como na Fig. 2.6, o primeiro argumento `null` indica que o diálogo de mensagem aparecerá no centro da tela. O segundo argumento é a mensagem a ser exibida. Neste caso, o segundo argumento é a expressão

`"The sum is " + sum`

que usa o operador `+` para “somar” um `String` (o literal `"The sum is "`) e o valor da variável `sum` (a variável `int` que contém o resultado da soma na linha 31). Java tem uma versão do operador `+` para a *concatenação de strings* que concatena um `String` e um valor de outro tipo de dados (incluindo outro `String`); o resultado dessa operação é um `String` novo (e normalmente mais longo). Se assumirmos que `sum` contém o valor 117, a expressão é avaliada da seguinte maneira:

1. Java determina que os dois operandos do operador `+` (o `string` `"The sum is "` e o inteiro `sum`) são de tipos diferentes e um deles é um `String`.
2. Java converte `sum` para um `String`.
3. Java acrescenta a representação de `sum` sob a forma de `String` no fim de `"The sum is "`, o que resulta no `String` `"The sum is 117"`.

O método `showMessageDialog` exibe o `String` resultante na caixa de diálogo. Observe que a conversão automática do inteiro `sum` ocorre somente porque o operador de adição concatena o `String` literal `"The sum is "` e `sum`. Também observe que o espaço entre `is` e 117 faz parte do `string` `"The sum is"`. A concatenação de `strings` é discutida em detalhes no Capítulo 10.



Erro comum de programação 2.10

Confundir o operador `+` utilizado para concatenação de strings com o operador `+` utilizado para adição pode levar a resultados estranhos. Por exemplo, suponha que a variável `inteira y` tem o valor 5, a expressão `"y + 2 = "` + `y + 2` resulta no string `"y + 2 = 52"`, não em `"y + 2 = 7"`, porque o primeiro valor de `y` é concatenado com o string `"y + 2 = "`, então o valor 2 é concatenado com o string novo e maior `"y + 2 = 5"`. A expressão `"y + 2 = "` + `(y + 2)` produz o resultado desejado.

O terceiro e quarto argumentos do método `showMessageDialog` na Fig. 2.9 representam o `string` que deveria aparecer na barra de título da caixa de diálogo e o tipo de caixa de diálogo, respectivamente. O quarto argumento – `JOptionPane.PLAIN_MESSAGE` – é um valor que indica o tipo de diálogo de mensagem a exibir. Esse tipo de diálogo de mensagem não exibe um ícone à esquerda da mensagem. A Fig. 2.11 ilustra o segundo e terceiro argumentos e mostra que não há nenhum ícone na janela.

Os tipos de diálogo de mensagem são mostrados na Fig. 2.12. Todos os tipos de diálogo de mensagem exceto os diálogos `PLAIN_MESSAGE` exibem um ícone para o usuário, que indica o tipo de mensagem.

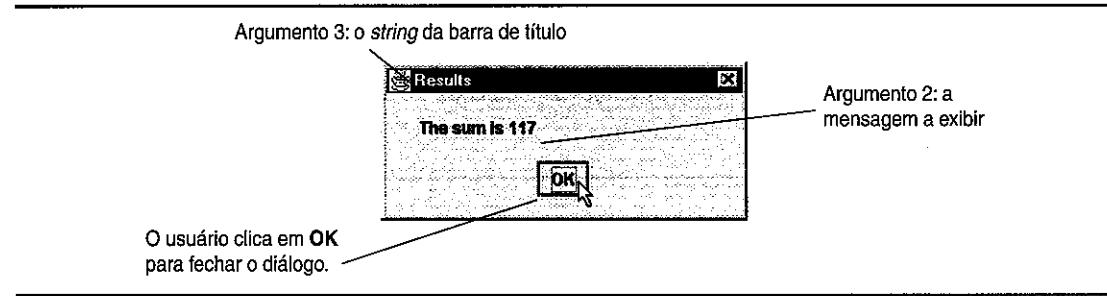


Fig. 2.11 Caixa de diálogo de mensagem personalizada com a versão de quatro argumentos do método `showMessageDialog`.

Tipo de diálogo de mensagem	Ícone	Descrição
JOptionPane.ERROR_MESSAGE		Exibe um diálogo que indica um erro para o usuário.
JOptionPane.INFORMATION_MESSAGE		Exibe um diálogo com uma mensagem com informações para o usuário – o usuário simplesmente pode dispensar o diálogo.
JOptionPane.WARNING_MESSAGE		Exibe um diálogo que adverte o usuário sobre um problema em potencial.
JOptionPane.QUESTION_MESSAGE		Exibe um diálogo que impõe uma pergunta para o usuário. Este diálogo normalmente exige uma resposta, tal como clicar em um botão Yes ou No.
JOptionPane.PLAIN_MESSAGE	Sem ícone	Exibe um diálogo que simplesmente contém uma mensagem, sem nenhum ícone.

Fig. 2.12 Constantes de JOptionPane para o diálogo de mensagem.

2.6 Conceitos de memória

Nomes de variáveis, como `number1`, `number2` e `sum`, na verdade correspondem a *posições* na memória do computador. Cada variável tem um *nome*, um *tipo*, um *tamanho* e um *valor*.

No programa de adição da Fig. 2.9, quando a instrução

```
number1 = Integer.parseInt( firstNumber );
```

é executada, o *string* previamente digitado pelo usuário no diálogo de entrada e armazenado em `firstNumber` é convertido em um `int` e colocado em uma posição da memória à qual o nome `number1` foi atribuído pelo compilador. Suponha que o usuário digite o *string* `45` como o valor para `firstNumber`. O programa converte `firstNumber` em um `int` e o computador coloca o valor inteiro `45` na posição `number1`, como mostrado na Fig. 2.13.

Sempre que um valor é colocado em uma posição da memória, esse valor substitui o valor que havia antes nessa posição. O valor anterior é destruído (i. e., perdido).

Quando a instrução

```
number2 = Integer.parseInt( secondNumber );
```

é executada, suponha que o usuário digite o *string* `72` como o valor para `secondNumber`. O programa converte `secondNumber` em um `int`, o computador coloca o valor inteiro `72` na posição `number2`. A memória fica como mostrado na Fig. 2.14.

Depois que o programa da Fig. 2.9 obtém os valores para `number1` e `number2`, ele soma os valores e coloca a soma na variável `sum`. A instrução

A diagram illustrating memory at address `number1`. The address is labeled "number1" and points to a box containing the value "45".

Fig. 2.13 Posição da memória que mostra o nome e o valor da variável `number1`.

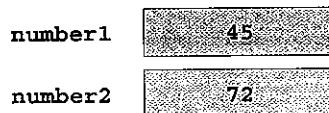


Fig. 2.14 Posições da memória depois que os valores para `number1` e `number2` foram armazenados.

```
sum = number1 + number2;
```

realiza a adição e também substitui o valor anterior de `sum`. Depois que `sum` foi calculada, a memória fica como mostrado na Fig. 2.15. Observe que os valores de `number1` e `number2` aparecem exatamente como eram antes de serem utilizados no cálculo de `sum`. Esses valores foram utilizados, mas não destruídos, enquanto o computador realizou o cálculo. Portanto, quando um valor é lido a partir de uma posição da memória, o processo é do tipo não-destrutivo.

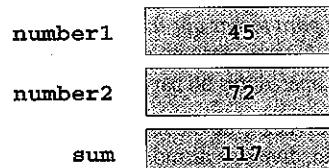


Fig. 2.15 Posições da memória após calcular a soma de `number1` e `number2`.

2.7 Aritmética

A maioria dos programas realiza cálculos aritméticos. Os *operadores aritméticos* são resumidos na Fig. 2.13. Observe o uso de vários símbolos especiais não utilizados em álgebra. O *asterisco* (*) indica multiplicação e o *sinal de porcentagem* (%) é o *operador módulo*, que é discutido brevemente. Os operadores aritméticos na Fig. 2.16 são operadores binários porque cada um deles opera sobre dois operandos. Por exemplo, a expressão `sum + value` contém o operador binário + e os dois operandos `sum` e `value`.

A *divisão inteira* produz um quociente inteiro; por exemplo, a expressão `7 / 4` é avaliada como 1 e a expressão `17 / 5` é avaliada como 3. Observe que qualquer parte fracionária na divisão de inteiro simplesmente é descartada (isto é, truncada) – não ocorre nenhum arredondamento. Java fornece o operador módulo, %, que produz o resto depois da divisão de inteiro. A expressão `x % y` produz o resto depois que `x` é dividido por `y`. Portanto, `7 % 4` produz 3 e `17 % 5` produz 2. Esse operador é mais comumente utilizado com operandos inteiros, mas também pode ser utilizado com outros tipos aritméticos. Em capítulos posteriores, analisamos muitas aplicações interessantes do operador módulo, como determinar se um número é um múltiplo de outro. Não há operador aritmético para exponenciação em Java. O Capítulo 5 mostra como realizar exponenciação em Java. [Nota: o operador módulo pode ser usado tanto com números inteiros quanto com números em ponto flutuante.]

As expressões aritméticas em Java devem ser escritas na *forma de linha reta* para facilitar a inserção de programas no computador. Portanto, as expressões como “`a dividido por b`” devem ser escritas como `a / b`, de modo que todas as constantes, variáveis e operadores apareçam em uma mesma linha. A notação algébrica geralmente não é aceitável para compiladores:

$$\frac{a}{b}$$

Operação de Java	Operador aritmético	Expressão algébrica	Expressão em Java
Adição	+	$f + 7$	<code>f + 7</code>
Subtração	-	$p - c$	<code>p - c</code>
Multiplicação	*	bm	<code>b * m</code>
Divisão	/	$x / y \text{ ou } \frac{x}{y} \text{ ou } x + y$	<code>x / y</code>
Módulo	%	$r \bmod s$	<code>r % s</code>

Fig. 2.16 Operadores aritméticos.

Os parênteses são utilizados em expressões em Java da mesma maneira que nas expressões algébricas. Por exemplo, para multiplicar a pela quantidade $b + c$ escrevemos:

$$a * (b + c)$$

Java aplica os operadores em expressões aritméticas em uma seqüência precisa determinada pelas seguintes *regras de precedência de operadores*, que são geralmente as mesmas que aquelas seguidas em álgebra:

1. Os operadores em expressões contidas dentro de pares de parênteses são avaliados primeiro. Portanto, os *parênteses podem ser utilizados para forçar que a ordem de avaliação ocorra em qualquer seqüência desejada pelo programador*. Os parênteses estão no nível “mais alto de precedência”. Em casos de parênteses aninhados ou embutidos, os operadores no par mais interno de parênteses são aplicados primeiro.
2. As operações de multiplicação, divisão e de módulo são aplicadas em seguida. Se uma expressão contém várias operações de multiplicação, divisão e de módulo, os operadores são aplicados da esquerda para a direita. Os operadores de multiplicação, divisão e módulo têm o mesmo nível de precedência.
3. As operações de adição e de subtração são aplicadas por último. Se uma expressão contém várias operações de adição e subtração, os operadores são aplicados da esquerda para a direita. A adição e a subtração têm o mesmo nível de precedência.

As regras de precedência de operadores permitem a Java aplicar operadores na ordem correta. Quando dizemos que os operadores são aplicados da esquerda para a direita, estamos nos referindo à *associatividade* dos operadores. Vemos que alguns operadores associam da direita para a esquerda. A Figura 2.17 resume essas regras de precedência de operadores. Essa tabela será expandida à medida que os operadores adicionais de Java forem introduzidos. Há um gráfico de precedência completo no Apêndice C.

Agora vamos considerar várias expressões à luz das regras de precedência de operadores. Cada exemplo lista uma expressão algébrica e seu equivalente Java.

O seguinte exemplo é uma média aritmética de cinco termos:

Álgebra: $m = \frac{a + b + c + d + e}{5}$

Java: `m = (a + b + c + d + e) / 5;`

Os parênteses são necessários porque a divisão tem precedência mais alta que aquela da adição. A soma total ($a + b + c + d + e$) será dividida por 5. Se os parênteses são omitidos erroneamente, obtemos $a + b + c + d + e / 5$, que é avaliado como

$$a + b + c + d + \frac{e}{5}$$

O seguinte exemplo é uma equação de uma linha reta:

Álgebra: $y = mx + b$

Operador(es)	Operação(ões)	Ordem de avaliação (precedência)
()	Parênteses	Avaliados primeiro. Se os parênteses estão aninhados, a expressão no par mais interno é avaliada primeiro. Se há vários pares de parênteses no mesmo nível (isto é, não aninhados), eles são avaliados da esquerda para a direita.
*, / e %	Multiplicação	Avaliados em segundo lugar. Se houver vários operadores deste tipo, eles são avaliados da esquerda para a direita.
+ ou -	Adição	Avaliados por último. Se houver vários operadores deste tipo, eles são avaliados da esquerda para a direita.

Fig. 2.17 Precedência de operadores aritméticos.

Java: $y = m * x + b;$

Não é necessário parêntese. O operador de multiplicação é aplicado primeiro porque a multiplicação tem uma precedência mais alta do que aquela da adição. A atribuição ocorre por último porque tem uma precedência mais baixa que a multiplicação e a adição.

O exemplo seguinte contém operações de módulo (%), multiplicação, divisão, adição e subtração:

Álgebra: $z = pr \% q + w/x - y$

Java: $z = p * r \% q + w / x - y;$

Os números dentro dos círculos embaixo da instrução indicam a ordem em que Java aplica os operadores. As operações de multiplicação, o módulo e a divisão são avaliados primeiro na ordem da esquerda para a direita (isto é, eles associam da esquerda para a direita), porque têm precedência mais alta do que aquela da adição e a subtração. A adição e a subtração são avaliadas a seguir. Estas operações também são aplicadas da esquerda para a direita.

Nem todas as expressões com vários pares de parênteses contêm parênteses aninhados. Por exemplo, a expressão

$a * (b + c) + c * (d + e)$

não contém parênteses aninhados. Em vez disso, dizemos que os parênteses estão no mesmo nível.

Para desenvolver uma melhor compreensão das regras de precedência de operadores, considere a avaliação de um polinômio de segundo grau ($y = ax^2 + bx + c$):

$y = a * x * x + b * x + c;$



Os números dentro dos círculos embaixo da instrução indicam a ordem em que Java aplica os operadores. Não há operador aritmético para exponenciação em Java; x^2 é representado como $x * x$.

Suponha que a, b, c e x sejam inicializados como segue: $a = 2, b = 3, c = 7$ e $x = 5$. A Fig. 2.18 ilustra a ordem em que os operadores são aplicados no polinômio de segundo grau precedente.

Como em álgebra, é aceitável colocar parênteses desnecessários em uma expressão para tornar a expressão mais clara. Esses parênteses desnecessários também são chamados de *parênteses redundantes*. Por exemplo, a instrução de atribuição precedente pode ser escrita com parênteses assim:

$y = (a * x * x) + (b * x) + c;$

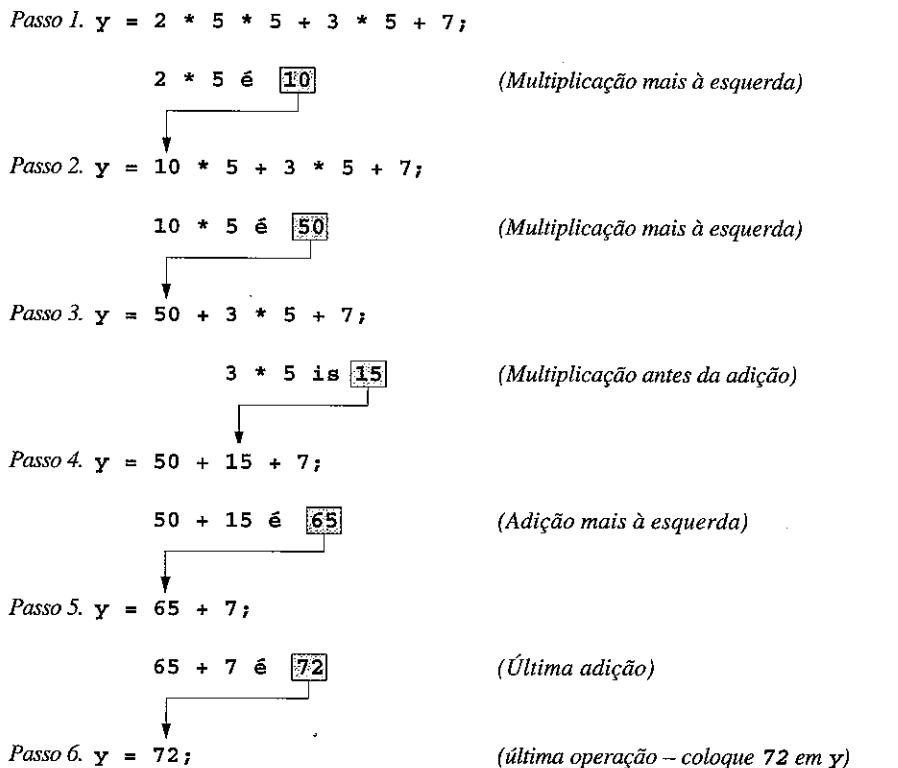


Fig. 2.18 Ordem em que um polinômio de segundo grau é avaliado.



Boa prática de programação 2.16

Utilizar parênteses para expressões aritméticas complexas, mesmo quando os parênteses não são necessários, pode tornar as expressões aritméticas mais fáceis de ler.

2.8 Tomada de decisão: operadores de igualdade e operadores relacionais

Esta seção introduz uma versão simples da estrutura `if` de Java que permite a um programa tomar uma decisão com base na verdade ou falsidade de alguma *condição*. Se a condição for atendida (isto é, a condição for *verdadeira*), a instrução no corpo da estrutura `if` será executada. Se a condição não for atendida (isto é, a condição for *falsa*), a instrução no corpo não será executada. Veremos um exemplo em seguida.

As condições em estruturas `if` podem ser formadas utilizando-se *operadores de igualdade e operadores relacionais*, resumidos na Fig. 2.19. Todos os operadores relacionais têm o mesmo nível de precedência e associam da esquerda para a direita. Os operadores de igualdade têm o mesmo nível de precedência, que é mais baixo que a precedência dos operadores relacionais. Os operadores de igualdade também associam da esquerda para a direita.



Erro comum de programação 2.11

É um erro de sintaxe se os operadores `==`, `!=`, `>=` e `<=` contiverem espaços entre seus símbolos, como em `= =`, `! =`, `> =` e `< =`, respectivamente.

Operador algébrico de igualdade padrão ou operador relacional	Operador de igualdade ou relacional em Java	Exemplo de condição em Java	Significado da condição em Java
<i>Operadores de igualdade</i>			
=	==	x == y	x é igual a y
≠	!=	x != y	x não é igual a y
<i>Operadores relacionais</i>			
>	>	x > y	x é maior que y
<	<	x < y	x é menor que y
≥	≥	x ≥ y	x é maior que ou igual a y
≤	≤	x ≤ y	x é menor que ou igual a y

Fig. 2.19 Operadores de igualdade e operadores relacionais.



Erro comum de programação 2.12

Inverter os operadores !=, ≥ e <= como em !=, => e =<, é um erro de sintaxe.



Erro comum de programação 2.13

Confundir o operador de igualdade == com o operador de atribuição, =, pode ser um erro de lógica ou de sintaxe. O operador de igualdade deve ser lido “é igual a” e o operador de atribuição deve ser lido “torna-se” ou “adquire o valor de”. Algumas pessoas preferem ler o operador de igualdade como “iguais duplos” ou “iguais iguais”.

O próximo exemplo utiliza seis estruturas if para comparar dois números digitados pelo usuário em campos de texto. Se a condição em qualquer uma dessas instruções if for verdadeira, a instrução de atribuição associada com aquela estrutura if é executada. O usuário insere dois valores por meio de diálogos de entrada. Em seguida, o programa converte os valores de entrada para inteiros e os armazena nas variáveis number1 e number2. Então o programa compara os números e exibe os resultados da comparação em um diálogo de informações. O programa e as saídas de exemplo são mostrados na Fig. 2.20.

```

1 // Fig. 2.20: Comparison.java
2 // Compara inteiros utilizando instruções if, operadores
3 // relacionais e operadores de igualdade
4
5 // Pacotes de extensão de Java
6 import javax.swing.JOptionPane;
7
8 public class Comparison {
9
10    // método main inicia a execução do aplicativo Java
11    public static void main( String args[] )
12    {
13        String firstNumber;    // primeiro string digitado pelo usuário
14        String secondNumber;   // segundo string digitado pelo usuário
15        String result;         // um string contendo a saída
16        int number1;           // primeiro número a comparar
17        int number2;           // segundo número a comparar
18

```

Fig. 2.20 Utilizando operadores de igualdade e operadores relacionais (parte 1 de 3).

```

19     // lê o primeiro número do usuário como um string
20     firstNumber =
21         JOptionPane.showInputDialog( "Enter first integer:" );
22
23     // lê o segundo número do usuário como um string
24     secondNumber =
25         JOptionPane.showInputDialog( "Enter second integer:" );
26
27     // converte os números do tipo String para o tipo int
28     number1 = Integer.parseInt( firstNumber );
29     number2 = Integer.parseInt( secondNumber );
30
31     // inicializa o resultado com o string vazio
32     result = "";
33
34     if ( number1 == number2 )
35         result = result + number1 + " == " + number2;
36
37     if ( number1 != number2 )
38         result = result + number1 + " != " + number2;
39
40     if ( number1 < number2 )
41         result = result + "\n" + number1 + " < " + number2;
42
43     if ( number1 > number2 )
44         result = result + "\n" + number1 + " > " + number2;
45
46     if ( number1 <= number2 )
47         result = result + "\n" + number1 + " <= " + number2;
48
49     if ( number1 >= number2 )
50         result = result + "\n" + number1 + " >= " + number2;
51
52     // Exibe os resultados
53     JOptionPane.showMessageDialog(
54         null, result, "Comparison Results",
55         JOptionPane.INFORMATION_MESSAGE );
56
57     System.exit( 0 );    // termina o aplicativo
58
59 } // termina o método main
60
61 } // termina a classe Comparison

```

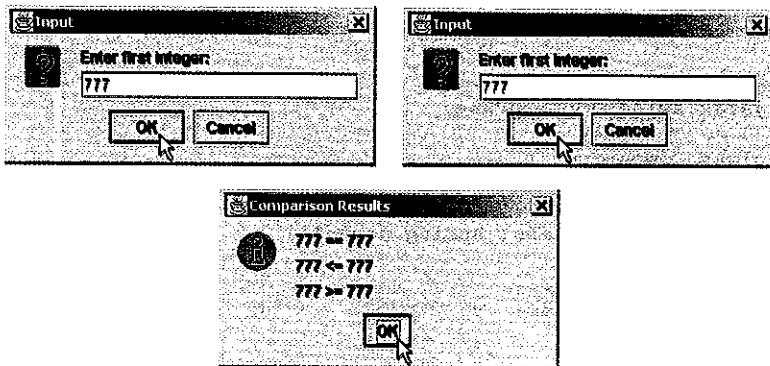


Fig. 2.20 Utilizando operadores de igualdade e operadores relacionais (parte 2 de 3).

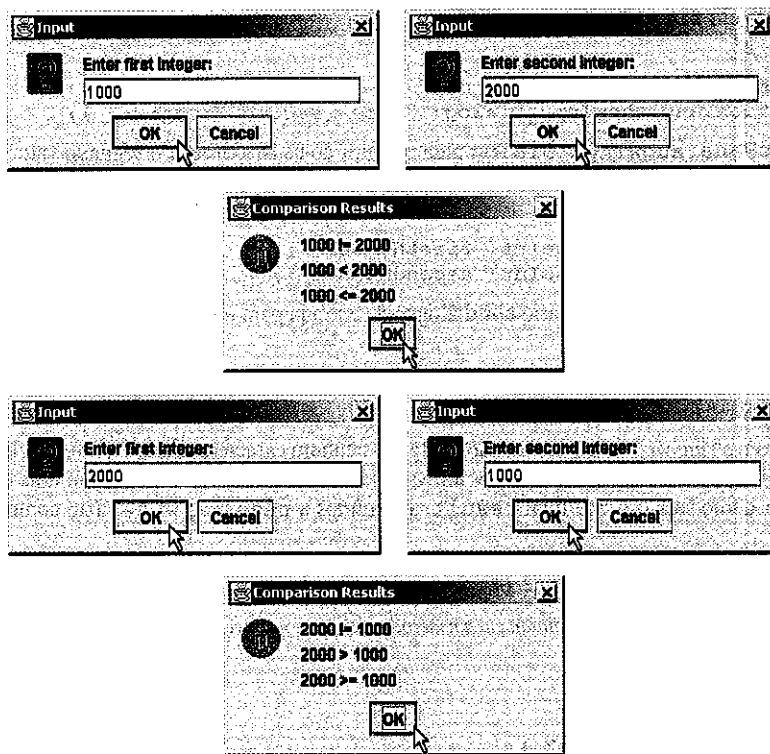


Fig. 2.20 Utilizando operadores de igualdade e operadores relacionais (parte 3 de 3).

A definição de classe do aplicativo **Comparison** inicia na linha 8

```
public class Comparison {
```

Como discutido anteriormente, o método **main** (linhas 11 a 59) inicia a execução de cada aplicativo Java. As linhas 13 a 17

```
String firstNumber; // primeiro string digitado pelo usuário
String secondNumber; // segundo string digitado pelo usuário
String result; // um string contendo a saída
int number1; // primeiro número a comparar
int number2; // segundo número a comparar
```

declaram as variáveis utilizadas no método **main**. Observe que há três variáveis do tipo **String** e duas variáveis do tipo **int**. Lembre-se de que as variáveis do mesmo tipo podem ser declaradas em uma declaração ou em múltiplas declarações. Se mais de um nome é declarado em uma declaração, os nomes são separados por vírgulas (,), como em

```
String firstNumber, secondNumber, result;
```

ou em

```
String firstNumber,
secondNumber,
result;
```

Este conjunto de nomes é chamado de *lista separada por vírgulas*. Mais uma vez, observe o comentário no fim de cada declaração nas linhas 13 a 17, indicando o propósito de cada variável no programa.

As linhas 20 e 21,

```
firstNumber =
JOptionPane.showInputDialog( "Enter first integer: " );
```

utilizam `JOptionPane.showInputDialog` para permitir ao usuário inserir o primeiro valor inteiro como um *string* e armazená-lo em `firstNumber`.

As linhas 24 e 25,

```
secondNumber =
    JOptionPane.showInputDialog( "Enter second integer:" );
```

utilizam `JOptionPane.showInputDialog` para permitir ao usuário inserir o segundo valor inteiro como um *string* e armazená-lo em `secondNumber`.

As linhas 28 e 29,

```
number1 = Integer.parseInt( firstNumber );
number2 = Integer.parseInt( secondNumber );
```

convertem cada *string* digitado pelo usuário nos diálogos de entrada para o tipo `int` e atribuem os valores a `number1` e `number2`.

A linha 32

```
result = "";
```

atribui a `result` um *string* vazio – um *string* não contendo nenhum caractere. Cada variável declarada em um método (como `main`) deve ser *inicializada* (receber um valor) antes de poder ser utilizada em uma expressão. Como ainda não sabemos como será o *string* final `result`, atribuímos a `result` o *string* vazio como um valor inicial temporário.



Erro comum de programação 2.14

Não inicializar uma variável definida em um método antes que variável seja utilizada no corpo do método é um erro de sintaxe.

As linhas 34 e 35

```
if ( number1 == number2 )
    result = result + number1 + " == " + number2;
```

definem uma *estrutura if* que compara os valores das variáveis `number1` e `number2` para determinar se eles são iguais. A estrutura `if` sempre inicia com a palavra-chave `if`, seguida por uma condição entre parênteses. A estrutura `if` espera encontrar uma instrução em seu corpo. O recuo mostrado aqui não é obrigatório, mas melhora a legibilidade do programa, enfatizando que a instrução na linha 35 faz parte da estrutura `if` que começa na linha 34.



Boa prática de programação 2.17

Recue a instrução no corpo de uma estrutura if para fazer o corpo da estrutura se destacar e para melhorar a legibilidade de programa.



Boa prática de programação 2.18

Coloque apenas uma instrução por linha em um programa. Isso aprimora a legibilidade do programa.

Na estrutura `if` precedente, se os valores das variáveis `number1` e `number2` forem iguais, a linha 35 atribui a `result` o valor de `result + number1 + " == " + number2`. Como discutido na Fig. 2.9, o operador nessa expressão executa a concatenação de *strings*. Para essa discussão, assumimos que cada uma das variáveis `number1` e `number2` tem o valor 123. Primeiro, a expressão converte o valor de `number1` para um *string* e o acrescenta a `result` (que atualmente contém o *string* vazio) para produzir um *string* "123". Em seguida, a expressão acrescenta " == " a "123" para produzir o *string* "123 == ". Por fim, a expressão acrescenta `number2` a "123 == " para produzir o *string* "123 == 123". O `String result` se torna mais longo à medida que o programa prossegue pelas estruturas `if` e executa mais concatenações. Por exemplo, dado o valor 123 para `number1` e `number2` nessa discussão, as condições de `if` nas linhas 46 e 47 (`<=`) e 49 e 50 (`>=`) também serão verdadeiras. Assim, o programa exibe o resultado

```
123 == 123
123 <= 123
123 >= 123
```

em um diálogo de mensagem.



Erro comum de programação 2.15

Substituir o operador `==` na condição de uma estrutura `if` como `if (x == 1)` pelo operador `=` como em `if (x = 1)` é um erro de sintaxe.



Erro comum de programação 2.16

Esquecer os parênteses esquerdos e direitos para a condição em uma estrutura `if` é um erro de sintaxe. Os parênteses são necessários.

Observe que não há ponto-e-vírgula (`;`) no final da primeira linha de cada estrutura `if`. Esse ponto-e-vírgula resultaria em um erro de lógica durante a execução. Por exemplo,

```
if ( number1 == number2 );      // erro de lógica
    result = result + number1 + " == " + number2;
```

na realidade seria interpretado por Java como

```
if ( number1 == number2 )
;

result = result + number1 + " == " + number2;
```

em que o ponto-e-vírgula sozinho na linha – chamada de *instrução vazia* – é a instrução a executar se a condição na estrutura `if` for verdadeira. Quando a instrução vazia é executada, nenhuma tarefa é realizada no programa. O programa então continua com a instrução de atribuição que é executada independentemente da condição ser verdadeira ou falsa.



Erro comum de programação 2.17

Colocar um ponto-e-vírgula imediatamente depois do parêntese direito da condição em uma estrutura `if` é normalmente um erro de lógica. O ponto-e-vírgula fará com que o corpo da estrutura `if` fique vazio, então a estrutura `if` em si não executará nenhuma ação, independentemente de sua condição ser verdadeira ou não. Pior ainda, a pretendida instrução do corpo da estrutura `if` torna-se agora uma instrução em seqüência com a estrutura `if` e será sempre executada.

Observe o uso de espaçamento na Fig. 2.20. Lembre-se de que os caracteres de espaço em branco como tabulações, nova linha e espaços normalmente são ignorados pelo compilador. Portanto, as instruções podem ser divididas em várias linhas e podem ser espaçadas de acordo com as preferências do programador sem afetar o significado de um programa. É incorreto dividir identificadores e literais de *string*. Idealmente, as instruções devem ser mantidas pequenas, mas nem sempre é possível fazer isso.



Boa prática de programação 2.19

Uma instrução longa pode se estender por várias linhas. Se uma única instrução deve ser dividida em várias linhas, escolha dividi-la em pontos que fazem sentido, como depois de uma vírgula em uma lista separada por vírgulas ou depois de um operador em uma expressão longa. Se uma instrução for dividida em duas ou mais linhas, recue todas as linhas subsequentes.

A tabela na Fig. 2.21 mostra a precedência dos operadores introduzidos neste capítulo. Os operadores são mostrados de cima para baixo em ordem decrescente de precedência. Observe que todos esses operadores, com a exceção do operador de atribuição `=`, associam da esquerda para a direita. A adição associa da esquerda para a direita, assim uma expressão como `x + y + z` é avaliada como se tivesse sido escrita `(x + y) + z`. O operador de atribuição `=` associa da direita para a esquerda, assim uma expressão como `x = y = 0` é avaliada como se tivesse sido escrita como `x = (y = 0)`, que, como logo veremos, primeiro atribui o valor 0 à variável `y` e depois atribui o resultado dessa atribuição, 0, a `x`.



Boa prática de programação 2.20

Consulte a tabela de precedência de operadores (veja a tabela completa no Apêndice C) ao escrever expressões que contêm muitos operadores. Confirme se os operadores na expressão são executados na ordem em que você espera. Se você não tiver certeza sobre a ordem de avaliação em uma expressão complexa, utilize parênteses para forçar a ordem, exatamente como faria em expressões algébricas. Certifique-se de observar que alguns operadores, como a atribuição (`=`), associam da direita para a esquerda, e não da esquerda para a direita.

Operadores	Associatividade	Tipo
()	da esquerda para a direita	parênteses
* / %	da esquerda para a direita	de multiplicação
+ -	da esquerda para a direita	de adição
< <= > >=	da esquerda para a direita	relacional
== !=	da esquerda para a direita	de igualdade
=	da direita para a esquerda	de atribuição

Fig. 2.21 Precedência e associatividade dos operadores discutidos até agora.

Apresentamos muitos recursos importantes de Java neste capítulo, incluindo a exibição de dados na tela, a inserção de dados a partir do teclado, a realização de cálculos e a tomada de decisão. Devemos salientar que todos estes aplicativos visam a apresentar ao leitor os conceitos básicos de programação. Como você verá nos capítulos mais adiante, aplicativos Java mais substanciais contêm somente umas poucas linhas de código no método `main`, que cria os objetos que executam o trabalho do aplicativo. No Capítulo 3, demonstraremos muitas técnicas semelhantes quando introduzimos a programação de *applets* Java. No Capítulo 4, baseamo-nos nas técnicas dos Capítulos 2 e 3 quando introduzimos a *programação estruturada*. Você estará mais familiarizado com as técnicas de recuo ou indentação. Estudaremos como especificar e variar a ordem em que as instruções são executadas; essa ordem é chamada de *fluxo de controle*.

2.9 (Estudo de caso opcional) Pensando em objetos: examinando a definição do problema

Agora iniciamos nosso estudo de caso opcional de projeto e implementação orientados a objetos. As seções “Pensando em objetos”, nos finais deste e dos próximos capítulos, vão deixá-lo à vontade com a orientação a objetos, examinando um estudo de caso de simulação de elevador. Este estudo de caso vai lhe proporcionar uma experiência de projeto e uma implementação completa, substancial e cuidadosamente cadenciada. Nos Capítulos 3 a 13, Capítulo 15 e Capítulo 22, executaremos as várias etapas de um projeto orientado a objetos (OOD) usando a UML ao mesmo tempo em que as relacionamos com os conceitos de orientação a objetos discutidos nos capítulos. Nos Apêndices G, H e I, implementaremos o simulador de elevador usando as técnicas de programação orientada a objetos (OOP) em Java. Apresentamos a solução completa do estudo de caso. Isto não é um exercício; em vez disso, é uma experiência de aprendizado de ponta a ponta que termina com um *walkthrough** detalhado do código real em Java. Oferecemos este estudo de caso para que você se acostume com os tipos de problemas de porte que são enfrentados em empresas. Esperamos que você goste desta experiência.

Definição do problema

Uma empresa pretende construir um edifício comercial de dois andares e equipá-lo com um elevador. Ela quer que você desenvolva em Java um *aplicativo de simulação de software* orientado a objetos que modele a operação do elevador para determinar se ele atenderá às necessidades da empresa. A empresa quer que a simulação contenha um sistema de elevador. O aplicativo consiste em três partes. A primeira e mais substancial é o simulador, que modela a operação do sistema de elevador. A segunda parte é a exibição deste modelo na tela, de modo que o usuário pode visualizá-lo graficamente. A parte final é a *interface gráfica com o usuário*, ou *GUI*, que permite ao usuário controlar a simulação. Nossa projeto e implementação seguirão a chamada *arquitetura Model-View-Controller*, que vamos estudar na Seção 13.17.

O sistema do elevador consiste em um poço de elevador e de um carro de elevador. em nossa simulação, modelamos pessoas que andam no carro do elevador (chamado de “o elevador”) para se movimentar entre os andares no poço do elevador, como mostrados nas Figuras 2.22, 2.23 e 2.24.

* N. de R.T.: termo usado para descrever uma das atividades de desenvolvimento de *software*, que consiste em reunir a equipe de desenvolvimento para analisar o código de um programa e discutir a implementação e as decisões tomadas pelos programadores; usada como ferramenta de treinamento e como parte de um processo de qualidade no desenvolvimento de *softwares*.

O elevador contém uma porta (chamada de “porta do elevador”) que se abre quando o elevador chega em um andar e fecha quando o elevador sai daquele andar. A porta do elevador permanece fechada durante os deslocamentos entre os andares para evitar que o passageiro se machuque raspando contra a parede do poço do elevador. Além disso, o poço do elevador se conecta a uma porta em cada andar (as duas “portas dos andares”), de modo que as pessoas não possam cair dentro do poço quando o elevador não está em um andar. Observe que não exibimos as portas dos andares nas figuras, porque elas iriam obstruir a visão do interior do elevador (usamos uma porta de tela para representar a porta do elevador porque a tela nos permite ver dentro do elevador). A porta do andar abre ao mesmo tempo em que a porta do elevador, de modo que parece que as duas portas abrem simultaneamente. Uma pessoa vê somente uma porta, dependendo da posição em que está. Quando a pessoa está dentro do elevador, a pessoa vê a porta do elevador e pode sair do elevador quando esta porta abre; quando a pessoa está fora do elevador, a pessoa vê a porta do andar e pode entrar no elevador quando a porta abre¹.

O elevador começa no primeiro andar, com todas as portas fechadas. Para poupar energia, o elevador só se movimenta quando necessário. Para simplificar, tanto o elevador como os andares têm capacidade para apenas uma pessoa².

O usuário de nosso aplicativo deve, a qualquer momento, ser capaz de criar uma nova pessoa na simulação e situar aquela pessoa no primeiro ou no segundo andar (Fig. 2.22³). Quando criada, a pessoa caminha pelo andar até o elevador. A pessoa então pressiona um botão no andar, próximo ao poço do elevador (um “botão de andar”). Quando pressionado, aquele botão de andar se ilumina e chama o elevador. Quando chamado, o elevador se movimenta até o andar em que está a pessoa. Se o elevador já está no andar daquela pessoa, o elevador não se movimenta. Quando chega, o elevador desliga o botão dentro do elevador (chamado de “botão do elevador”), faz soar a campainha dentro do elevador e, então, abre a porta do elevador (a qual abre a porta do andar naquele andar). O elevador então notifica o poço do elevador da chegada. O poço do elevador, ao receber esta mensagem, desliga o botão do andar e acende a luz naquele andar.

De vez em quando, uma pessoa chama o elevador quando ele está em movimento. Se a chamada foi gerada no andar do qual o elevador recém partiu, o elevador precisa “lembra” de voltar àquele andar depois de transportar o passageiro atual para o outro andar.

Quando a porta do andar se abre, a pessoa entra no elevador depois que o passageiro do elevador (se houver um) sai. Se uma pessoa não entra nem chama o elevador, o elevador fecha a porta e permanece naquele andar até que a próxima pessoa pressione um botão de andar para chamar o elevador. Quando uma pessoa entra no elevador, ela pressiona o botão do elevador, que também se ilumina quando pressionado. O elevador fecha a porta (a qual também fecha a porta do andar) e se movimenta para o outro andar. O elevador leva cinco segundos para viajar entre um andar e outro. Quando o elevador chega ao andar de destino, a porta do elevador se abre (junto com a porta do andar) e a pessoa sai do elevador.

O usuário do aplicativo introduz uma pessoa no primeiro ou no segundo andar pressionando o botão **First Floor** ou o botão **Second Floor**, respectivamente. Quando o usuário pressiona o botão **First Floor**, uma pessoa deve ser criada (pela simulação do elevador) e posicionada no primeiro andar do prédio. Quando o usuário pressiona o botão **Second Floor**, uma pessoa deve ser criada e posicionada no segundo andar. Ao longo do tempo, o usuário pode criar uma quantidade qualquer de pessoas na simulação, mas o usuário não pode criar uma nova pessoa em um andar ocupado. Por exemplo, a Fig. 2.22 mostra que o botão **First Floor** está desabilitado para evitar que o usuário crie mais de uma pessoa no primeiro andar. A Fig. 2.23 mostra que este botão é novamente habilitado quando a pessoa entra no elevador.

A empresa quer que exibamos os resultados da simulação graficamente, como mostrado nas Figuras 2.22, 2.23 e 2.24. Em diversos instantes no tempo, a tela deve exibir uma pessoa caminhando para o elevador, pressionando um botão e entrando, andando e saindo do elevador. O vídeo também deve mostrar o elevador se movendo, as portas abrindo, as luzes acendendo e apagando, os botões se iluminando quando pressionados e os botões escurecendo quando são desligados.

1 A maioria das pessoas não pensa nisto quando anda em um elevador – elas realmente pensam em uma “porta do elevador”, quando, na verdade, existe uma porta no elevador e uma porta no andar e estas portas abrem e fecham em conjunto.

2 Depois que você tiver estudado este estudo de caso, você pode querer modificá-lo para permitir que mais de uma pessoa ande no elevador ao mesmo tempo e mais de uma pessoa espere em cada andar ao mesmo tempo.

3 Para criar partes dos gráficos para a simulação de elevador, usamos imagens que a Microsoft oferece gratuitamente para download no endereço de msdn.microsoft.com/downloads/default.asp. Criamos outros gráficos com o Paint Shop Pro™ da Jasc® Software.

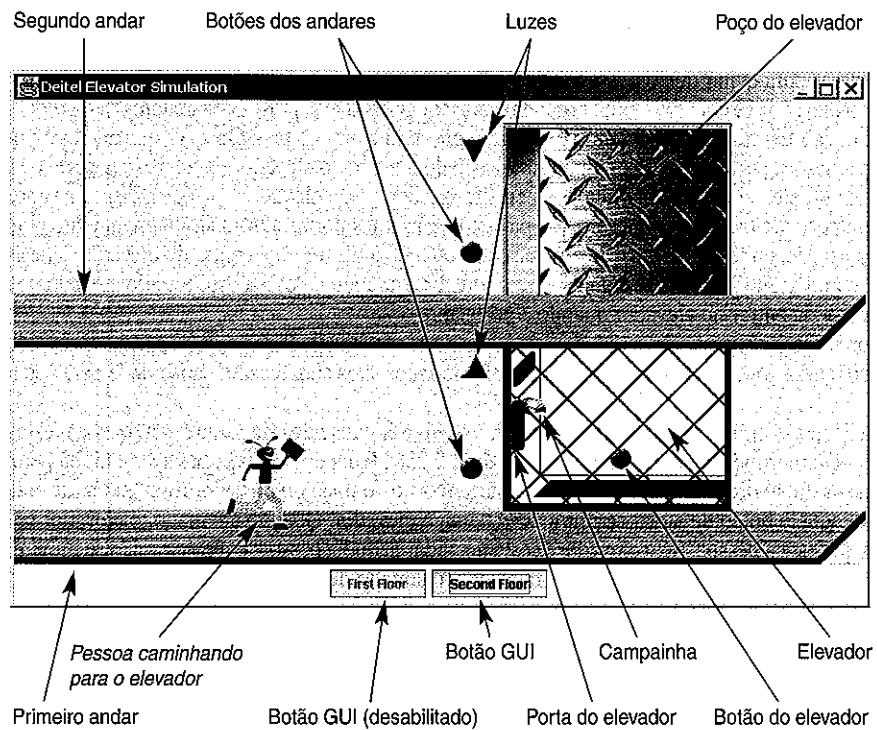


Fig. 2.22 Pessoa caminhando em direção ao elevador no primeiro andar.

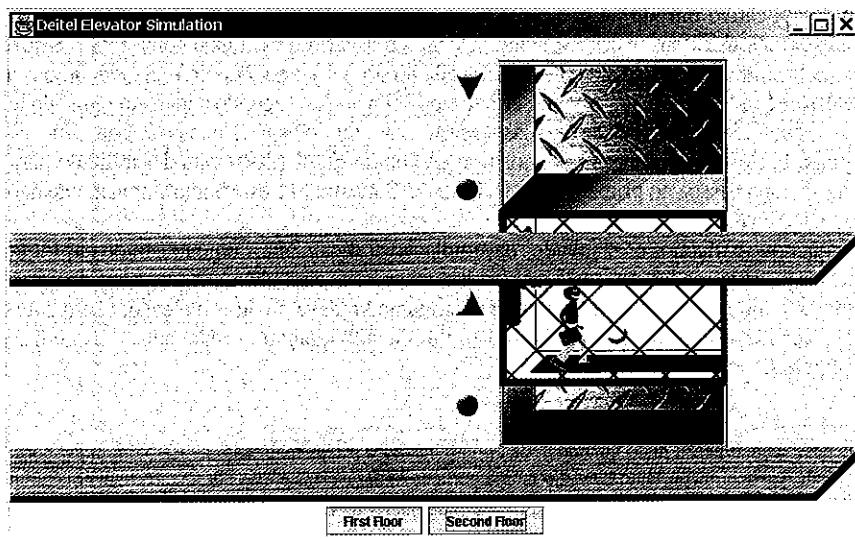


Fig. 2.23 Pessoa andando no elevador para o segundo andar.

A empresa quer que seja integrado áudio à simulação. Por exemplo, quando uma pessoa caminha, o usuário do aplicativo deve ouvir as passadas. Cada vez que um botão de andar ou do elevador é pressionado ou desligado, o usuário deve ouvir um clique. A campainha deve soar quando da chegada do elevador e as portas devem ranger quando elas abrem e fecham. Finalmente, deve tocar “música de elevador” enquanto o elevador se movimenta entre os andares.

Analisando e projetando o sistema de elevador

Precisamos analisar e projetar nosso sistema antes de implementarmos seu código em Java. A saída da fase de análise se destina a especificar claramente em um *documento de requisitos* o que o sistema deve fazer. O documento de requisitos para este estudo de caso é essencialmente a descrição do que o simulador de elevador deve fazer – apresentado informalmente nas páginas anteriores. A saída da fase de projeto deve especificar claramente *como* o sistema deve ser construído para fazer o que é necessário. Nas próximas seções “Pensando em objetos”, executamos as etapas de um projeto orientado a objetos (OOD) para o sistema de elevador. A UML é projetada para uso com qualquer processo OOD – existem muitos destes processos. Um método popular é o Rational Unified Process™ desenvolvido pela Rational Software Corporation. Para este estudo de caso, apresentamos nosso próprio processo de projeto simplificado. Para muitos de nossos leitores, esta será sua primeira experiência com OOD/UML.

Começamos agora a fase de projeto de nosso sistema de elevador, que irá se desenvolver ao longo dos Capítulos 2 a 13, Capítulo 15 e Capítulo 22, na qual desenvolvemos gradualmente o projeto. Os Apêndices G, H e I apresentam a implementação em Java completa.

Um sistema é um conjunto de componentes que interagem para resolver um problema. Em nosso estudo de caso, o aplicativo de simulação de elevador representa o sistema. O sistema pode conter “subsistemas”, que são “sistemas dentro de um sistema”. Os subsistemas simplificam o processo de projeto administrando subconjuntos de responsabilidades do sistema. Os projetistas de sistemas podem dividir as responsabilidades do sistema entre os subsistemas, projetar os subsistemas e, então, integrar os subsistemas com o sistema global. Nossa sistema de simulação de elevador contém três subsistemas, que estão definidos na definição do problema:

1. o modelo do simulador (que representa a operação do sistema de elevador);
2. a exibição deste modelo na tela (de modo que o usuário possa visualizá-lo de forma gráfica); e
3. a interface gráfica com o usuário (que permite ao usuário controlar a simulação).

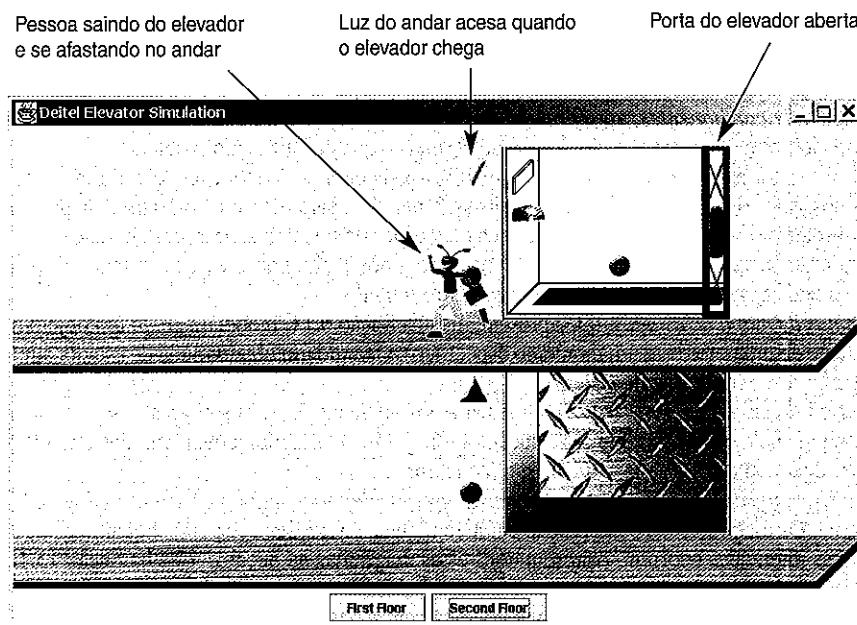


Fig. 2.24 Pessoa se afastando do elevador.

Desenvolvemos o modelo do simulador gradualmente até o Capítulo 15 e apresentamos o modelo implementado no Apêndice H. Discutimos os componentes GUI que permitem ao usuário controlar o modelo no Capítulo 12 e apresentamos como os subsistemas trabalham em conjunto para formar o sistema no Capítulo 13. Finalmente, apresentamos como exibir o modelo do simulador na tela no Capítulo 22 e terminamos a exibição no Apêndice I.

A *estrutura* do sistema descreve os objetos do sistema e seus inter-relacionamentos. O *comportamento* do sistema descreve como o sistema muda à medida que seus objetos interagem uns com os outros. Todos os sistemas têm estrutura e comportamento – precisamos projetar ambos. Entretanto, existem diversos *tipos* distintos de estruturas e comportamentos de sistemas. Por exemplo, a interação entre objetos no sistema é diferente da interação entre o usuário e o sistema, embora ambas sejam interações que constituem o comportamento do sistema.

A UML especifica nove tipos de diagramas para modelagem de sistemas. Cada diagrama modela uma característica distinta da estrutura ou do comportamento de um sistema – os primeiros quatro diagramas dizem respeito à estrutura do sistema; os cinco diagramas restantes dizem respeito ao comportamento do sistema:

1. Diagrama de classes
2. Diagrama de objetos
3. Diagrama de componentes
4. Diagrama de instalação
5. Diagrama de atividades
6. Diagrama de mapa de estados
7. Diagrama de colaborações
8. Diagrama de seqüência
9. Diagrama de casos de uso

Os *diagramas de classes*, que explicamos na Seção 3.8 de “Pensando em objetos”, modelam as classes, ou “blocos de construção”, usados para construir um sistema. Cada entidade na definição do problema é um candidato a ser uma classe no sistema (i.e., **Person**, **Elevator**, **Floor**, etc.).

Os *diagramas de objetos*, que também explicamos na Seção 3.8, modelam uma “fotografia” do sistema através da modelagem dos objetos de um sistema e seus relacionamentos em um momento específico no tempo. Cada objeto representa uma instância de uma classe do diagrama de classes (p. ex., o objeto elevador é uma instância da classe **Elevator**) e podem existir diversos objetos criados a partir de uma classe (p. ex., tanto o objeto botão do primeiro andar como o objeto botão do segundo andar são criados a partir da classe **FloorButton**).

Os *diagramas de componentes*, apresentados na Seção 13.17, modelam os *componentes* – recursos (que incluem gráficos e áudio) e *pacotes* (que são grupos de classes) – que constituem o sistema.

Os *diagramas de instalação* modelam os requisitos do sistema durante a execução (como o computador ou computadores nos quais o sistema irá ficar), requisitos de memória para o sistema ou outros dispositivos que o sistema requeira durante a execução. Não apresentamos diagramas de instalação neste estudo de caso, porque não estamos projetando um sistema para um *hardware* específico – nossa simulação exige somente um computador que contém o ambiente de execução Java 2 no qual possa ser executado.

Os *diagramas de mapa de estados*, que apresentamos na Seção 5.11, modelam *como* um objeto muda de *estado* (i.e., a condição de um objeto em um momento específico). Quando um objeto muda de estado, aquele objeto pode se comportar de forma diferente no sistema.

Os *diagramas de atividades*, que também apresentamos na Seção 5.11, modelam a *atividade* de um objeto – o fluxo de trabalho daquele objeto durante a execução do programa. O diagrama de atividades é um fluxograma que modela as ações que o objeto vai ser executado e em que ordem.

Tanto os *diagramas de colaborações* quanto os *diagramas de seqüência* modelam as interações entre os objetos de um sistema. Os diagramas de colaboração enfatizam quais interações ocorrem, enquanto os de seqüência enfatizam quando as interações ocorrem. Apresentamos estes diagramas na Seção 7.10 e na Seção 15.12, respectivamente.

Os *diagramas de casos de uso* representam a interação entre o usuário e nosso sistema (i.e., todas as ações que o usuário pode executar no sistema). Apresentamos diagramas de casos de uso na Seção 12.16, na qual discutimos aspectos da interface com o usuário.

Na Seção 3.17 de “Pensando em objetos”, continuamos a projetar nosso sistema de elevador identificando as classes na definição do problema. Usando estas classes, desenvolvemos um diagrama de classes que modela a estrutura de nosso sistema de simulação de elevador.

Recursos na Internet e na World Wide Web

A seguir estão listados URLs e livros sobre projeto orientado a objetos com a UML – você poderá achar estas referências úteis à medida que estudar as seções restantes de nossa apresentação de estudo de caso.

www.omg.org/technology/uml/index.htm

Esta é a página de recursos para UML do Object Management Group, que fornece especificações para várias tecnologias orientadas a objetos, como a UML.

www.smartdraw.com/drawing/software/indexUML.asp

Este site mostra como desenhar diagramas de UML sem o uso de ferramentas de modelagem.

www.rational.com/uml/index.jsp

Esta é a página de recursos para UML da Rational Software Corporation – a empresa que criou a UML.

microgold.com/Stage/UML_FAQ.html

Fornecê a FAQ (lista de perguntas freqüentes) sobre UML mantida por Rational Software.

www.softdocwiz.com/Dictionary.htm

Hospeda o Unified Modeling Language Dictionary, que lista e define todos os termos usados na UML.

www.embarcadero.com

Fornecê uma licença de 30 dias para baixar uma versão experimental de Describe™ – a nova ferramenta de modelagem de UML da Embarcadero Technologies®.

www.ics.uci.edu/pub/arch/uml/uml_books_and_tools.html

Relaciona livros sobre a UML e ferramentas de software que usam a UML, como o Rational Rose™ e o Embarcadero Describe™.

www.ootips.org/ood-principles.html

Fornecê respostas para a pergunta “o que torna um projeto orientado a objetos bom?”

wdvl.internet.com/Authoring/Scripting/Tutorial/oo_design.html

Este site apresenta OOD e fornece recursos para OOD.

Bibliografia

Booch, G., *Object-Oriented Analysis and Design with Applications*. Addison-Wesley. Massachusetts; 1994.

Folwer, M., and Scott, K., *UML Distilled Second Edition; A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley. Massachusetts; 1999.

Larman, C., *Applying UML and Patterns; An Introduction to Object-Oriented Analysis and Design*. Prentice Hall. New Jersey; 1998.

Page-Jones, M., *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley. Massachusetts; 1999.

Rumbaugh, J.; Jacobson, I.; and Booch, G., *The Unified Modeling Language Reference Manual*. Addison-Wesley. Massachusetts; 1999.

Rumbaugh, J.; Jacobson, I.; and Booch, G., *The Unified Modeling Language User Guide*. Addison-Wesley. Massachusetts; 1999.

Rumbaugh, J.; Jacobson, I.; and Booch, G., *The Complete UML Training Course*. Prentice Hall. New Jersey; 2000.

Rumbaugh, J.; Jacobson, I.; and Booch, G., *The Unified Software Development Process*. Addison-Wesley. Massachusetts; 1999.

Rosenburg, D., and Scott, K., *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Addison-Wesley. Massachusetts; 2001.

Schach, S., *Object-Oriented and Classical Software Engineering*. McGraw Hill. New York; 2001.

Schneider, G., and Winters, J., *Applying Use Cases*. Addison-Wesley. Massachusetts; 1998.

Scott, K., *UML Explained*. Addison-Wesley. Massachusetts; 2001.

Stevens, P., and Pooley, R.J., *Using UML: Software Engineering with Objects and Components Revised Edition*. Addison-Wesley; 2000.

Resumo

- O aplicativo é um programa que é executado com o interpretador **java**.
- O comentário que começa com `//` é chamado de comentário de uma única linha. Os programadores inserem comentários para documentar programas e aprimorar a legibilidade do programa.
- O *string* de caracteres contido entre aspas duplas é chamado de *string*, *string* de caractere, mensagem ou literal de *string*.
- Linhas em branco, caracteres de espaço, caracteres de nova linha e caracteres de tabulação são conhecidos como caracteres de espaço em branco. Os caracteres de espaço em branco fora dos *strings* são ignorados pelo compilador.
- A palavra-chave **class** introduz uma definição de classe e é imediatamente seguida pelo nome da classe.
- As palavras-chave (ou palavras reservadas) são reservadas para uso de Java. As palavras-chave devem aparecer inteiramente em letras minúsculas.
- Por convenção, todos os nomes de classe em Java iniciam com uma primeira letra em maiúscula. Se um nome de classe tiver mais de uma palavra, cada palavra deve ser iniciada com uma letra maiúscula.
- O identificador é uma série de caracteres que consiste em letras, dígitos, sublinhados (`_`) e sinais de cifrão (`$`), que não inicia com um dígito, não contém nenhum espaço e não é uma palavra-chave.
- Java diferencia letras maiúsculas de minúsculas – as letras minúsculas e maiúsculas são diferentes.
- A chave esquerda, `{`, inicia o corpo de cada definição de classe. A chave direita correspondente, `}`, termina cada definição de classe.
- Os aplicativos Java começam a rodar no método **main**.
- Os métodos são capazes de realizar tarefas e retornar informações quando completam suas tarefas.
- A primeira linha do método **main** deve ser definida como

```
public static void main( String args[] )
```

- A chave esquerda, `{`, inicia o corpo de uma definição de método. A chave direita correspondente, `}`, termina o corpo da definição de método.
- **System.out** é conhecido como o objeto de saída padrão. **System.out** permite aos aplicativos Java exibir *strings* e outros tipos de informações na janela de comando a partir da qual o aplicativo Java é executado.
- A seqüência de escape `\n` indica um caractere nova linha. Entre as seqüências de escape incluem-se `\t` (tabulação), `\r` (retorno de carro), `\\"` (barras invertidas) e `"` (aspas duplas).
- O método **println** do objeto **System.out** exibe (ou imprime) uma linha de informações na janela de comando. Quando **println** completa sua tarefa, ele automaticamente posiciona o cursor de saída no início da próxima linha na janela de comando.
- Cada instrução deve terminar com um ponto-e-vírgula (também conhecido como terminador de instrução).
- A diferença entre os métodos **print** e **println** de **System.out** é que **print** não posiciona o cursor no começo da próxima linha na janela de comando quando termina de exibir seu argumento. O próximo caractere exibido na janela de comando aparece imediatamente depois do último caractere exibido com **print**.
- Java contém muitas classes predefinidas que são agrupadas em categorias de classes relacionadas chamadas pacotes. Os pacotes são conhecidos coletivamente como biblioteca de classes Java ou interface de programas aplicativos Java (API Java).
- A classe **JOptionPane** é definida para nós em um pacote denominado **javax.swing**. A classe **JOptionPane** contém métodos que exibem caixas de diálogo.
- O compilador utiliza as instruções **import** para localizar classes necessárias para compilar um programa Java.
- O pacote **javax.swing** contém muitas classes que ajudam a definir uma interface gráfica com o usuário (GUI) para um aplicativo. Os componentes GUI facilitam a entrada de dados pelo usuário de um programa e as saídas de dados por um programa.
- O método **showMessageDialog** da classe **JOptionPane** exibe uma caixa de diálogo que contém uma mensagem para o usuário.
- O método **static** é chamado seguindo seu nome de classe por um ponto (`.`) e pelo nome do método.
- O método **exit** da classe **System** termina um aplicativo. A classe **System** está no pacote **java.lang**. Todos os programas Java importam o pacote **java.lang** por *default*.
- A variável é uma posição na memória do computador na qual um valor pode ser armazenado para sua utilização por um programa. O nome de uma variável é qualquer identificador válido.
- Todas as variáveis devem ser declaradas com um nome e um tipo de dados antes de poderem ser utilizadas em um programa.
- As declarações terminam com um ponto-e-vírgula (`;`) e podem ser divididas em várias linhas com cada variável na declaração separada por uma vírgula (isto é, uma lista de nomes de variáveis separados por vírgulas).

- Variáveis do tipo `int` armazenam valores inteiros (números inteiros como 7, -11, 0, 31914).
- Tipos como `int`, `float`, `double` e `char` são tipos de dados primitivos. Os nomes de tipos de primitivos são palavras-chave da linguagem de programação Java.
- O `prompt` orienta o usuário a realizar uma ação específica.
- O valor é atribuído a uma variável com uma instrução de atribuição, a qual usa o operador de atribuição `=`. O operador `=` é chamado de operador binário porque tem dois operandos.
- O método `Integer.parseInt` (um método `static` da classe `Integer`) converte seu argumento `String` em um valor `int`.
- Cada variável tem um nome, um tipo, um tamanho e um valor.
- Quando um valor é colocado em uma posição da memória, ele substitui o valor que havia antes nessa posição. Quando um valor é lido a partir de uma posição da memória, o valor da variável permanece inalterado.
- Os operadores aritméticos são operadores binários porque operam sobre dois operandos.
- A divisão de inteiros produz um resultado inteiro.
- As expressões aritméticas em Java devem ser escritas em linha reta para facilitar a entrada de programas no computador.
- Os operadores em expressões aritméticas são aplicados em uma seqüência precisa determinada pelas regras de precedência de operadores.
- Os parênteses podem ser utilizados para forçar a ordem de avaliação de operadores.
- Quando dizemos que os operadores são aplicados da esquerda para a direita, estamos nos referindo à associatividade dos operadores. Alguns operadores associam da direita para a esquerda.
- A estrutura `if` de Java permite a um programa tomar uma decisão com base na verdade ou falsidade de uma condição. Se a condição for atendida (i. e., a condição for verdadeira), a instrução no corpo da estrutura `if` é executada. Se a condição não for atendida (i. e., a condição for falsa), a instrução do corpo não é executada.
- As condições em estruturas `if` podem ser formadas com os operadores de igualdade e os operadores relacionais.
- O `string` vazio é um `string` que não contém nenhum caractere.
- Cada variável declarada em um método deve ser inicializada antes de poder ser utilizada em uma expressão.

Terminologia

<i>aplicativo</i>	<i>concatenação de string</i>
<i>applet</i>	<i>condição</i>
<i>argumento para um método</i>	<i>corpo de uma definição de classe</i>
<i>aritméticos</i>	<i>corpo de uma definição de método</i>
<i>associatividade da direita para a esquerda</i>	<i>cursor do mouse</i>
<i>associatividade de operadores</i>	<i>decisão</i>
<i>barra de título de um diálogo</i>	<i>declaração</i>
<i>biblioteca de classes Java</i>	<i>definição de classe</i>
<i>caixa de diálogo</i>	<i>diálogo</i>
<i>caractere de escape barra invertida (\)</i>	<i>diálogo de entrada</i>
<i>caractere de nova linha (\n)</i>	<i>diálogo de mensagem</i>
<i>caracteres de espaço em branco</i>	<i>distinção de letras maiúsculas e minúsculas</i>
<i>chave direita } termina o corpo de um método</i>	<i>divisão de inteiro</i>
<i>chave direita } termina o corpo de uma classe</i>	<i>documentar um programa</i>
<i>chave esquerda { inicia o corpo de um método</i>	<i>== “é igual a”</i>
<i>chave esquerda { inicia o corpo de uma classe</i>	<i>> “é maior que”</i>
<i>chaves ({ e })</i>	<i>>= “é maior que ou igual a”</i>
<i>classe</i>	<i>< “é menor que”</i>
<i>classe definida pelo programador</i>	<i><= “é menor que ou igual a”</i>
<i>classe definida pelo usuário</i>	<i>erro de compilação</i>
<i>classe Integer</i>	<i>erro de sintaxe</i>
<i>classe JOptionPane</i>	<i>erro durante a compilação</i>
<i>classe String</i>	<i>estrutura if</i>
<i>classe System</i>	<i>extensão de arquivo .class</i>
<i>comentário (//)</i>	<i>extensão de arquivo .java</i>
<i>comentário de documentação Java</i>	<i>falso</i>
<i>comentário de múltiplas linhas</i>	<i>ferramenta de comando</i>
<i>comentário de uma única linha</i>	<i>ferramenta de shell</i>
<i>compila erro</i>	<i>formato de linha reta</i>
<i>compilador</i>	<i>identificador</i>

instrução	operador
instrução de atribuição	operador binário
instrução import	operador de adição (+)
inteiro (int)	operador de atribuição (=)
interface de programas aplicativos Java (API)	operador de concatenação de string (+)
interface gráfica com usuário	operador de divisão (/)
interpretador	operador de módulo (%)
interpretador java	operador de multiplicação (*)
janela de comando	operador de subtração (-) (GUI)operadores
Java	operadores de igualdade
Java 2 Software Development Kit (J2SDK)	operadores relacionais
Java 2 Software Development Kit (J2SDK)	operando
JOptionPane.ERROR_MESSAGE	pacote
JOptionPane.INFORMATION_MESSAGE	pacote java.lang
JOptionPane.PLAIN_MESSAGE	pacote javax.swing
JOptionPane.QUESTION_MESSAGE	palavra-chave class
JOptionPane.WARNING_MESSAGE	palavra-chave public
lista separada por vírgulas	palavra-chave void
literal	palavras reservadas
memória	parênteses ()
mensagem	parênteses aninhados
método	ponteiro do mouse
método exit de System	posição da memória
método main	precedência
método parseInt da classe Integer	prompt
método showInputDialog de JOptionPane	Prompt do MS-DOS
método showMessageDialog de JOptionPane	regras de precedência do operador
método static	seqüência de escape
método System.out.print	string
método System.out.println	string vazio ("")
!= "não é igual a"	strings de caracteres
navegador Microsoft Internet Explorer	terminador de instrução (;)
navegador Netscape Navigator	terminador de instrução ponto-e-vírgula (;)
nome de classe	tipo de dados primitivo
nome de variável	tipo primitivo int
objeto	valor de variável
objeto de saída padrão	variável
objeto System.out	verdade

Exercícios de auto-revisão

2.1 Preencha as lacunas em cada uma das seguintes afirmações:

- A _____ inicia o corpo de cada método e a _____ termina o corpo de cada método.
- Cada instrução termina com um _____.
- A estrutura _____ é utilizada para tomada de decisão.
- _____ inicia um comentário de uma única linha.
- _____, _____, _____ e _____ são chamados de caracteres de espaçamento.
- A classe _____ contém métodos que exibem diálogos de mensagem e diálogos de entrada.
- As _____ são reservados para uso pelo Java.
- Os aplicativos Java iniciam a execução no método _____.
- Os métodos _____ e _____ exibem informações na janela de comando.
- O método _____ sempre é chamado com seu nome de classe seguido por um ponto (.) e o nome do método.

2.2 Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.

- Os comentários fazem com que o computador imprima o texto depois das // na tela quando o programa é executado.
- Todas as variáveis devem receber um tipo quando são declaradas.

- c) Java considera que as variáveis `number` e `NuMbEr` são idênticas.
- d) O operador de módulo (%) pode ser utilizado apenas com operandos inteiros.
- e) Os operadores aritméticos *, /, %, + e - têm o mesmo nível de precedência.
- f) O método `Integer.parseInt` converte um inteiro em um `String`.

- 2.3** Escreva instruções Java para realizar cada uma das seguintes tarefas:
- a) Declarar as variáveis `c`, `thisIsAVariable`, `q76354` e `number` como do tipo `int`.
 - b) Exibir um diálogo que solicita ao usuário para digitar um inteiro.
 - c) Converter um `String` em um inteiro e armazenar o valor convertido na variável inteira `age`. Suponha que o `String` seja armazenado em `value`.
 - d) Se a variável `number` não é igual a 7, exibir "The variable `number` is not equal to 7" em um diálogo de mensagem. (Dica: utilize a versão do diálogo de mensagem que exige dois argumentos.)
 - e) Imprimir a mensagem "This is a Java program" em uma linha na janela de comando.
 - f) Imprimir a mensagem "This is a Java program" em duas linhas na janela de comando; a primeira linha deve terminar com `Java`. Utilize apenas uma instrução.
- 2.4** Identifique e corrija os erros em cada uma das seguintes instruções:
- a) `if (c < 7);`
`JOptionPane.showMessageDialog(null,`
`"c is less than 7");`
 - b) `if (c => 7)`
`JOptionPane.showMessageDialog(null,`
`"c is equal to or greater than 7");`
- 2.5** Escreva uma instrução (ou um comentário) para realizar cada uma das seguintes tarefas:
- a) Declarar que um programa calculará o produto de três inteiros.
 - b) Declarar as variáveis `x`, `y`, `z` e `result` como do tipo `int`.
 - c) Declarar as variáveis `xVal`, `yVal` e `zVal` como do tipo `String`.
 - d) Solicitar ao usuário para inserir o primeiro valor, ler o valor do usuário e armazená-lo na variável `xVal`.
 - e) Solicitar ao usuário para inserir o segundo valor, ler o valor do usuário e armazená-lo na variável `yVal`.
 - f) Solicitar ao usuário para inserir o terceiro valor, ler o valor do usuário e armazená-lo na variável `zVal`.
 - g) Converter `xVal` em um `int` e armazenar o resultado na variável `x`.
 - h) Converter `yVal` em um `int` e armazenar o resultado na variável `y`.
 - i) Converter `zVal` em um `int` e armazenar o resultado na variável `z`.
 - j) Computar o produto dos três inteiros contidos nas variáveis `x`, `y` e `z` e atribuir o resultado à variável `result`.
 - k) Exibir um diálogo contendo a mensagem "The product is " seguido pelo valor da variável `result`.
 - l) Retornar um valor a partir do programa para indicar que o programa foi finalizado com sucesso.
- 2.6** Utilizando as instruções que você escreveu no Exercício 2.5, escreva um programa completo que calcula e imprime o produto de três inteiros.

Respostas aos exercícios de auto-revisão

- 2.1** a) Chave esquerda ({}, chave direita }). b) Ponto-e-vírgula (;). c) `if`. d) `//`. e) Linhas em branco, caracteres de espaço, caracteres de nova linha e caracteres de tabulação. f) `JOptionPane`. g) Palavras-chave. h) `main`. i) `System.out.print` e `System.out.println`. j) `static`.
- 2.2** a) Falsa. Os comentários não fazem com que nenhuma ação seja realizada quando o programa é executado. Eles são utilizados para documentar programas e melhorar sua legibilidade.
- b) Verdadeira.
- c) Falsa. Java diferencia letras maiúsculas de minúsculas, assim essas variáveis são diferentes.
- d) Falsa. O operador de módulo também pode ser usado com operandos não-inteiros em Java.
- e) Falsa. Os operadores *, / e % estão no mesmo nível de precedência e os operadores + e - estão em um nível mais baixo de precedência.
- f) Falsa. O método `Integer.parseInt` converte um `String` em um valor inteiro (`int`).
- 2.3** a) `int c, thisIsAVariable, q76354, number;`
b) `value = JOptionPane.showInputDialog("Enter an integer");`

```

c) age = Integer.parseInt( stringValue );
d) if ( number != 7 )
        JOptionPane.showMessageDialog( null,
            "The variable number is not equal to 7" );
e) System.out.println( "This is a Java program" );
f) System.out.println( "This is a Java\nprogram" );

```

2.4 As soluções para o exercício de auto-revisão 2.4 são as seguintes:

- a) Erro: ponto-e-vírgula depois do parêntese direito da condição na instrução `if`. Correção: remova o ponto-e-vírgula depois do parêntese direito. [Nota: o resultado desse erro é que a instrução de saída será executada independentemente da condição na instrução `if` for verdadeira. O ponto-e-vírgula depois do parêntese direito é considerado uma instrução vazia – uma instrução que não faz nada. Aprenderemos mais sobre a instrução vazia no próximo capítulo.]
- b) Erro: o operador relacional `=>` é incorreto. Correção: altere `=>` para `>=`.

2.5

```

a) // Calcula o produto de três inteiros
b) int x, y, z, result;
c) String xVal, yVal, zVal;
d) xVal = JOptionPane.showInputDialog(
        "Enter first integer:" );
e) yVal = JOptionPane.showInputDialog(
        "Enter second integer:" );
f) zVal = JOptionPane.showInputDialog(
        "Enter third integer:" );
g) x = Integer.parseInt( xVal );
h) y = Integer.parseInt( yVal );
i) z = Integer.parseInt( zVal );
j) result = x * y * z;
k) JOptionPane.showMessageDialog( null,
        "The product is " + result );
l) System.exit( 0 );

```

2.6 A solução para o Exercício 2.6 é a seguinte:

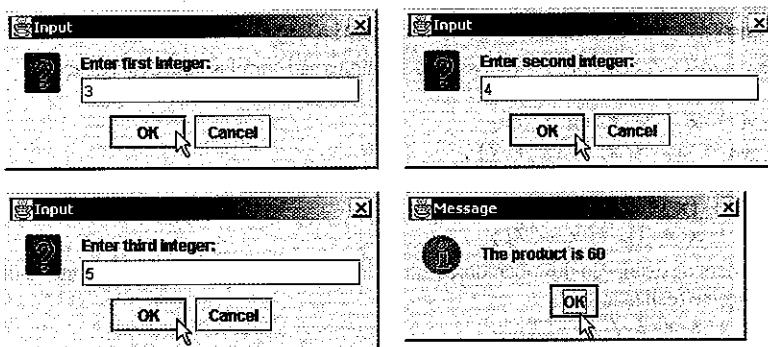
```

1 // Calcula o produto de três inteiros
2
3 // Pacotes de extensão de Java
4 import javax.swing.JOptionPane;
5
6 public class Product {
7
8     public static void main( String args[] )
9     {
10         int x, y, z, result;
11         String xVal, yVal, zVal;
12
13         xVal = JOptionPane.showInputDialog(
14             "Enter first integer:" );
15         yVal = JOptionPane.showInputDialog(
16             "Enter second integer:" );
17         zVal = JOptionPane.showInputDialog(
18             "Enter third integer:" );
19
20         x = Integer.parseInt( xVal );
21         y = Integer.parseInt( yVal );
22         z = Integer.parseInt( zVal );
23
24         result = x * y * z;
25         JOptionPane.showMessageDialog( null,
26             "The product is " + result );

```

```

27
28     System.exit( 0 );
29 }
30 }
```



Exercícios

- 2.7** Preencha as lacunas em cada uma das seguintes afirmações:
- São utilizados _____ para documentar um programa e aprimorar sua legibilidade.
 - O diálogo de entrada capaz de receber entrada do usuário é exibido com o método da classe _____.
 - Uma decisão pode ser tomada em um programa Java com uma _____.
 - Os cálculos normalmente são realizados pelas instruções _____.
 - O diálogo capaz de exibir uma mensagem para o usuário é exibido com o método da classe _____.
- 2.8** Escreva instruções em Java que realizam cada uma das seguintes tarefas:
- Exibir a mensagem "Enter two numbers" utilizando a classe `JOptionPane`.
 - Atribuir o produto das variáveis `b` e `c` para a variável `a`.
 - Declarar que um programa realiza um exemplo de cálculo de folha de pagamento (isto é, utiliza um texto que ajuda a documentar um programa).
- 2.9** Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- Os operadores Java são avaliados da esquerda para a direita.
 - Os seguintes nomes de variável são válidos: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales$`, `his$_account_total`, `a`, `b$`, `c`, `z`, `zz`.
 - Uma expressão aritmética Java válida sem parênteses é avaliada da esquerda para a direita.
 - Os seguintes nomes de variável são inválidos: `3g`, `87`, `67h2`, `h22`, `2h`.
- 2.10** Preencha as lacunas em cada uma das afirmações seguintes:
- Quais são as operações aritméticas que têm a mesma precedência que a multiplicação? _____.
 - Quando os parênteses são aninhados, qual conjunto de parênteses é avaliado primeiro em uma expressão aritmética? _____.
 - A posição na memória do computador que pode conter valores diferentes em vários momentos durante a execução de um programa é chamada de _____.
- 2.11** O que é exibido no diálogo de mensagem quando cada uma das seguintes instruções Java é executada? Pressuponha que `x = 2` e `y = 3`.
- `JOptionPane.showMessageDialog(null, "x = " + x);`
 - `JOptionPane.showMessageDialog(null,`
 `"The value of x + x is " + (x + x));`
 - `JOptionPane.showMessageDialog(null, "x =");`
 - `JOptionPane.showMessageDialog(null,`
 `(x + y) + " = " + (y + x));`

2.12 Quais das seguintes instruções Java contêm variáveis cujos valores são alterados ou substituídos?

- a) `p = i + j + k + 7;`
- b) `JOptionPane.showMessageDialog(null,
 "variáveis cujos valores são destruídos");`
- c) `JOptionPane.showMessageDialog(null, "a = 5");`
- d) `stringVal = JOptionPane.showInputDialog("Enter string: ");`

2.13 Dado que $y = ax^3 + 7$, quais das instruções Java seguintes são corretas para essa equação?

- a) `y = a * x * x * x + 7;`
- b) `y = a * x * x * (x + 7);`
- c) `y = (a * x) * x * (x + 7);`
- d) `y = (a * x) * x * x + 7;`
- e) `y = a * (x * x * x) + 7;`
- f) `y = a * x * (x * x + 7);`

2.14 Declare a ordem de avaliação dos operadores em cada uma das instruções Java seguintes e mostre o valor de `x` depois que cada instrução é realizada.

- a) `x = 7 + 3 * 6 / 2 - 1;`
- b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
- c) `x = (3 * 9 * (3 + (9 * 3 / (3))));`

2.15 Escreva um aplicativo que exiba os números 1 a 4 na mesma linha, com cada par de números adjacentes separados por um espaço. Escreva o programa utilizando os seguintes métodos:

- a) Utilizando uma instrução `System.out`.
- b) Utilizando quatro instruções `System.out`.

2.16 Escreva um aplicativo que solicita ao usuário para digitar dois números, obtém os dois números do usuário e imprime a soma, o produto, a diferença e o quociente (divisão) dos dois números. Utilize as técnicas mostradas na Fig. 2.9.

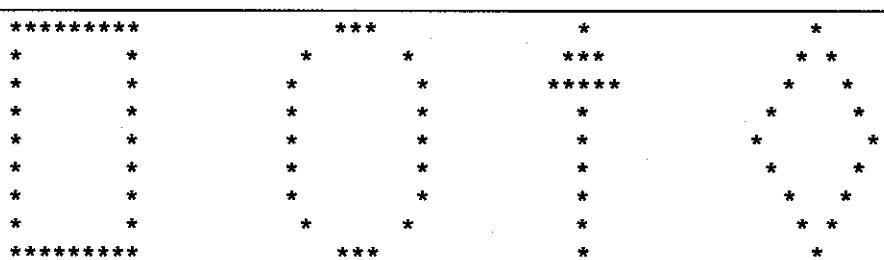
2.17 Escreva um aplicativo que solicita ao usuário para digitar dois inteiros, que obtém os números do usuário e exibe o maior número seguido pelas palavras “**is larger**” em um diálogo de mensagem de informação. Se os números forem iguais, ele imprime a mensagem “**These numbers are equal**”. Utilize as técnicas mostradas na Fig. 2.20.

2.18 Escreva um aplicativo que lê três inteiros digitados pelo usuário e exibe a soma, a média, o produto, o maior e o menor desses números em um diálogo de mensagem de informação. Utilize as técnicas de GUI mostradas na Fig. 2.20. [Nota: o cálculo da média nesse exercício deve resultar em uma representação da média na forma de inteiro. Então, se a soma dos valores é 7, a média será 2 e não 2,3333...]

2.19 Escreva um aplicativo que lê uma entrada do usuário definindo o raio de um círculo e que imprime o diâmetro, a circunferência e a área do círculo. Utilize o valor 3,14159 para π . Utilize as técnicas de GUI mostradas na Fig. 2.9. [Nota: você também pode utilizar a constante `Math.PI` predefinida para o valor de π . Essa constante é mais precisa que o valor 3,14159. A classe `Math` é definida no pacote `java.lang`, assim você não precisa importá-la.] Utilize as seguintes fórmulas (r é o raio):

$$\begin{aligned} \text{diâmetro} &= 2r \\ \text{circunferência} &= \pi r^2 \\ \text{área} &= \pi r^2 \end{aligned}$$

2.20 Escreva um aplicativo que exibe na janela de comando uma caixa, uma oval, uma seta e um losango utilizando asteriscos (*) como segue:



2.21 Modifique o programa que você criou no Exercício 2.20 para exibir as formas em um diálogo `JOptionPane`.`PLAIN_MESSAGE`. O programa exibe as formas exatamente como no Exercício 2.20?

2.22 O que o seguinte código imprime?

```
System.out.println( "*\n**\n***\n****\n*****" );
```

2.23 O que os seguintes códigos imprimem?

```
System.out.println( "**" );
System.out.println( "***" );
System.out.println( "*****" );
System.out.println( "****" );
System.out.println( "***" );
```

2.24 O que os seguintes códigos imprimem?

```
System.out.print( "**" );
System.out.print( "***" );
System.out.print( "*****" );
System.out.print( "****" );
System.out.println( "***" );
```

2.25 O que o seguinte código imprime?

```
System.out.print( "**" );
System.out.println( "***" );
System.out.println( "*****" );
System.out.print( "****" );
System.out.println( "***" );
```

2.26 Escreva um aplicativo que lê cinco inteiros e determina e imprime o maior e o menor inteiro do grupo. Utilize somente as técnicas de programação que você aprendeu neste capítulo.

2.27 Escreva um aplicativo que lê um inteiro e determina e imprime se ele é ímpar ou par. [Dica: utilize o operador de módulo. Um número par é um múltiplo de 2. Qualquer múltiplo de 2 deixa um resto zero quando dividido por 2.]

2.28 Escreva um aplicativo que lê dois inteiros e determina e imprime se o primeiro é um múltiplo do segundo. [Dica: utilize o operador de módulo.]

2.29 Escreva um aplicativo que exibe na janela de comando um padrão de tabuleiro de damas como segue:

```
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
```

2.30 Modifique o programa que você escreveu no Exercício 2.29 para exibir o padrão de tabuleiro de damas em um diálogo `JOptionPane`.`PLAIN_MESSAGE`. O programa exibe as formas exatamente como no Exercício 2.29?

2.31 Eis uma prévia do que virá futuramente. Neste capítulo você aprendeu sobre inteiros e o tipo de dados `int`. Java também pode representar letras maiúsculas, minúsculas e uma variedade considerável de símbolos especiais. Cada caractere tem uma representação correspondente de inteiro. O conjunto de caracteres que um computador utiliza e das representações correspondentes na forma de inteiro desses caracteres é chamado de *conjunto de caracteres* desse computador. Você pode indicar um valor de caractere em um programa simplesmente incluindo esse caractere entre aspas simples, como com '`A`'.

Você pode determinar o equivalente na forma de inteiro de um caractere precedendo esse caractere com `(int)`. Isso se chama coerção (falaremos mais sobre coerções no Capítulo 4):

```
(int) 'A'
```

A instrução abaixo enviará para a saída um caractere e seu equivalente na forma de inteiro:

```
System.out.println( "The character " + 'A' +
    " has the value " + ( int ) 'A' );
```

Quando a instrução acima é executada, ela exibe o caractere A e o valor 65 (do chamado conjunto de caracteres Unicode) como parte do *string*.

Escreva um aplicativo que exibe os equivalentes na forma de inteiro de algumas letras maiúsculas, minúsculas, dígitos e símbolos especiais. Exiba no mínimo os inteiros equivalentes aos seguintes: A B C a b c 0 1 2 \$ * + / e o caractere espaço em branco.

2.32 Escreva um aplicativo que lê um número que consiste em cinco dígitos digitados pelo usuário, separa o número em seus dígitos individuais e imprime os dígitos separados uns dos outros por três espaços cada. Por exemplo, se o usuário digitar o número 42339, o programa deve imprimir

4	2	3	3	9
---	---	---	---	---

[*Dica*: esse exercício é possível de ser realizado com as técnicas que você aprendeu neste capítulo. Você precisará utilizar tanto as operações de divisão como as de módulo para “selecionar” cada dígito.]

Suponha que o usuário digite a quantidade de dígitos correta. O que acontece quando você executa o programa e digita um número com mais de cinco dígitos? O que acontece quando você executa o programa e digita um número com menos de cinco dígitos?

2.33 Utilizando apenas as técnicas de programação que aprendeu neste capítulo, escreva um aplicativo que calcula os quadrados e cubos dos números de 0 a 10 e imprime os valores resultantes no formato de tabela como segue:

número	quadrado	cubo
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

[*Nota*: esse programa não exige nenhuma entrada do usuário.]

2.34 Escreva um programa que lê o nome e o sobrenome de um usuário como duas entradas separadas e concatena o nome e o sobrenome separados por um espaço. Exiba em um diálogo de mensagem o nome concatenado.

2.35 Escreva um programa que lê cinco números e determina e imprime quantos números negativos, quantos números positivos e quantos zeros foram lidos.

3

Introdução a *applets* Java

Objetivos

- Observar alguns dos fascinantes recursos de Java através dos *applets* de demonstração do Java 2 Software Development Kit.
- Diferenciar um *applet* de um aplicativo.
- Ser capaz de escrever *applets* Java simples.
- Ser capaz de escrever arquivos de Hypertext Markup Language (HTML) simples para carregar um *applet* no **appletviewer** ou em um navegador da World Wide Web.
- Entender a diferença entre variáveis e referências.
- Executar *applets* em navegadores da World Wide Web.

*He would answer to "Hi!" or to any loud cry
Such as "Fry me!" or "Fritter my wig!"
To "What-you-may-call-um!" or "What-was-his-name!"
But especially "Thing-um-a-jig!"*

Lewis Carroll

*A pintura é apenas uma ponte ligando a mente do pintor
com a do observador.*

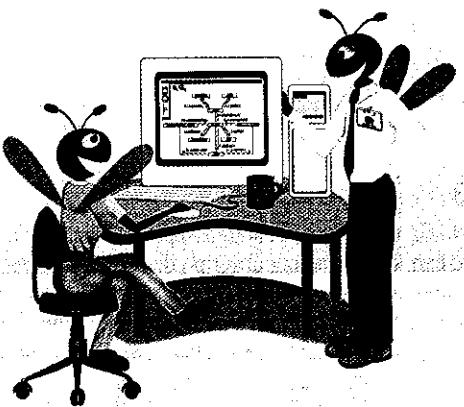
Eugène Delacroix

*Meu método é pegar o problema extremo para encontrar a
coisa certa a dizer e então dizê-la com a frivolidade
extrema.*

George Bernard Shaw

Embora isso seja loucura, ainda há método nela.

William Shakespeare



Sumário do capítulo

- 3.1 Introdução
- 3.2 Applets de exemplo do Java 2 Software Development Kit
 - 3.2.1 O applet Tic-Tac-Toe
 - 3.2.2 O applet DrawTest
 - 3.2.3 O applet Java2D
- 3.3 Um applet Java simples desenhando um string
 - 3.3.1 Compilando e executando o WelcomeApplet
- 3.4 Dois applets mais simples desenhando strings e linhas
- 3.5 Outro applet Java adicionando números de ponto flutuante
- 3.6 Visualizando applets em um navegador da Web
 - 3.6.1 Visualizando applets no Netscape Navigator 6
 - 3.6.2 Visualizando applets em outros navegadores usando o Java Plug-in
- 3.7 Recursos para applets Java na Internet e na World Wide Web
- 3.8 (Estudo de caso opcional) Pensando em objetos: identificando as classes em uma definição de problema

*Resumo • Diagramas • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão
Exercícios*

3.1 Introdução

No Capítulo 2, apresentamos a programação de aplicativos Java e vários aspectos importantes desses aplicativos. Esse capítulo introduz outro tipo de programa Java chamado *applet Java*. *Applets* são programas Java que podem ser embutidos em documentos *Hypertext Markup Language (HTML)* (i. e., páginas da Web). Quando um navegador carrega uma página da Web que contém um *applet*, o *applet* é baixado para o navegador e começa a ser executado.

O navegador que executa um *applet* é denominado genericamente *contêiner do applet*. O Java 2 Software Development Kit (J2SDK) inclui um contêiner de *applets* (chamado *appletviewer*) para testar *applets* antes de você embuti-los em uma página da Web. A maioria dos navegadores em uso atualmente não suporta o Java 2 diretamente. Por este motivo, normalmente demonstramos nossos *applets* usando o *appletviewer*. Um navegador que suporta Java 2 é o Netscape Navigator 6. Para executar *applets* em outros navegadores da Web como o Microsoft Internet Explorer ou versões anteriores do Netscape Navigator é necessário o *Java Plug-in*, que discutimos na Seção 3.6.2 deste capítulo.



Dica de portabilidade 3.1

A maioria dos navegadores da Web em uso atualmente não suporta applets escritos em Java 2. Para executar applets em tais navegadores, você deve usar o Java Plug-in (ver Seção 3.6.2).



Dica de teste e depuração 3.1

Teste seus applets no contêiner de applets *appletviewer* antes de executá-los em um navegador. Isto permite que você veja mensagens de erro que podem ocorrer. Além disso, depois que um applet está sendo executado em um navegador, às vezes é difícil carregar novamente o applet depois de fazer alterações na definição de classe do applet.



Dica de teste e depuração 3.2

Teste applets em todos os navegadores da Web nos quais os applets serão executados, para assegurar que eles operam corretamente em cada navegador.

Um de nossos objetivos neste capítulo é simular vários recursos apresentados no Capítulo 2. Isso fornece um reforço positivo de conceitos anteriores. Outro objetivo deste capítulo é começar a usar a terminologia de programação orientada a objetos apresentada na Seção 1.15.

Como no Capítulo 2, há alguns casos em que, por enquanto, não fornecemos todos os detalhes necessários para criar aplicativos e *applets* complexos em Java. É importante primeiro construir seu conhecimento sobre conceitos fundamentais de programação. No Capítulo 4 e no Capítulo 5, apresentamos um tratamento detalhado de *desenvolvimento de programas e controle de programas* em Java. À medida que prosseguimos pelo texto, apresentamos muitos aplicativos e *applets* substanciais.

3.2 Applets de exemplo do Java 2 Software Development Kit

Iniciamos considerando vários *applets* de exemplo fornecidos com a versão 1.2.1 do Java 2 Software Development Kit (J2SDK). Os *applets* que demonstramos lhe dão uma idéia das capacidades de Java. Cada um dos programas de exemplo fornecidos com o J2SDK também vem com o *código-fonte* (os arquivos `.java` que contém os programas do *applet* Java). Esse código-fonte é útil à medida que você aprimorar seu conhecimento de Java – você pode ler o código-fonte fornecido para aprender novos e estimulantes recursos de Java. Lembre-se, todos os programadores inicialmente aprendem novos recursos imitando sua utilização em programas existentes. O J2SDK vem com muitos programas e há um número imenso de recursos Java na Internet e na World Wide Web que incluem código-fonte Java.

Os programas de demonstração fornecidos com o J2SDK estão localizados em seu diretório de instalação do J2SDK em um subdiretório chamado `demo`. Para o Java 2 Software Development Kit versão 1.3, a localização padrão do diretório `demo` no Windows é

```
c:\jdk1.3\demo
```

No UNIX/Linux, é o diretório no qual você instala o J2SDK, seguido por `jdk1.3/demo` – por exemplo

```
/usr/local/jdk1.3/demo
```

Para outras plataformas, haverá uma estrutura semelhante de diretório (ou pasta). Para o propósito deste capítulo, estamos pressupondo no Windows que o J2SDK esteja instalado em `c:\jdk1.3` e no UNIX que o J2SDK esteja instalado em seu diretório inicial em `~/jdk1.3`. [Nota: você pode precisar atualizar essas localizações para indicar seu diretório de instalação e/ou unidade de disco escolhido ou uma versão mais nova do J2SDK.]

Se você estiver utilizando um Java Development Kit que não vem com as demos de Java da Sun, você pode baixar o J2SDK (com as demos) do site de Java da Sun Microsystems

```
http://java.sun.com/products/jdk/1.3/
```

3.2.1 O applet TicTacToe

O primeiro *applet* das demos J2SDK que demonstramos é o *applet TicTacToe*, que permite jogar o jogo-da-velha contra o computador. Para executar esse *applet*, abra uma janela de comando (*prompt* do MS-DOS no Windows/95/98/ME, *prompt de comando* no Windows NT/2000 ou uma *ferramenta de comando/ferramenta do shell* no UNIX) e mude para o diretório `demo` do J2SDK. Tanto no Windows como no UNIX, utilize o comando `cd` para mudar (*change*) de diretório. Por exemplo, o comando

```
cd c:\jdk1.3\demo
```

muda para o diretório `demo` no Windows e o comando

```
cd ~/jdk1.3/demo
```

muda para o diretório `demo` no UNIX.

O diretório `demo` contém quatro subdiretórios – `applets`, `jfc`, `jpda` e `sound` (você pode ver esses diretórios digitando na janela de comando o comando `dir` no Windows ou o comando `ls` no UNIX). O diretório `applets` contém muitos *applets* de demonstração. O diretório `jfc` (Java Foundation Classes) contém muitos exemplos dos novos recursos gráficos e de GUI de Java (alguns desses exemplos também são *applets*). O diretório `jpda` contém exemplos da Java Platform Debugging Architecture (além do escopo deste livro). O diretório `sound` contém exemplos da Java Sound API (abordada no Capítulo 18). Para as demonstrações nesta seção, mude para o diretório `applets` digitando o comando

```
cd applets
```

tanto no Windows como no UNIX.

Listar o conteúdo do diretório **applets** (com o comando **dir** no Windows ou **ls** no UNIX) mostrará que há muitos exemplos. A Fig. 3.1 mostra os subdiretórios e fornece uma breve descrição dos exemplos em cada subdiretório.

Mude para o subdiretório **TicTacToe**. Nesse diretório você encontrará o arquivo HTML **example1.html** que é utilizado para executar o *applet*. Digite o comando

```
appletviewer example1.html
```

e pressione a tecla *Enter*. Isso executa o **appletviewer**. O **appletviewer** carrega o arquivo de HTML especificado como seu *argumento de linha de comando* (**example1.html**), determina com base no arquivo qual *ap-*

Exemplo	Descrição
Animator	Realiza uma de quatro animações separadas.
ArcTest	Demonstra desenhos de arcos. Você pode interagir com o <i>applet</i> para alterar atributos do arco que é exibido.
BarChart	Desenha um gráfico de barras simples.
Blink	Exibe texto piscando em cores diferentes.
CardTest	Demonstra vários componentes GUI e uma variedade de maneiras em que os componentes GUI podem ser organizados na tela (o arranjo de componentes GUI também é conhecido como leiaute dos componentes GUI).
Clock	Desenha um relógio com ponteiros giratórios, a data e a hora atuais. O relógio é atualizado uma vez por segundo.
DitherTest	Demonstra desenhos com uma técnica gráfica conhecida como pontilhamento, que permite uma transformação gradual de uma cor para outra.
DrawTest	Permite ao usuário arrastar o mouse para desenhar linhas e pontos no <i>applet</i> em cores diferentes.
Fractal	Desenha um fractal. Os fractais em geral requerem cálculos complexos para determinar como são exibidos.
GraphicsTest	Desenha uma variedade de formas para ilustrar capacidades gráficas.
GraphLayout	Desenha um gráfico que consiste em muitos nós (representados como retângulos) ligados por linhas. Arraste um nó para ver os outros nós no gráfico se ajustarem na tela e para demonstrar interações gráficas complexas.
ImageMap	Demonstra uma imagem com <i>pontos ativos</i> . Posicionar o ponteiro do mouse sobre certas áreas da imagem destaca a área e uma mensagem é exibida no canto inferior esquerdo da janela appletviewer . Posicione sobre a boca da imagem para ouvir o <i>applet</i> dizer “olá”.
JumpingBox	Move um retângulo aleatoriamente pela tela. Tente pegá-lo clicando nele com o mouse!
MoleculeViewer	Apresenta uma visualização tridimensional de várias moléculas químicas diferentes. Arraste o mouse para ver a molécula de ângulos diferentes.
NervousText	Desenha um texto que se move por toda a tela.
SimpleGraph	Desenha uma curva complexa.
SortDemo	Compara três técnicas de classificação. A classificação (descrita no Capítulo 7) organiza as informações em ordem – como palavras em ordem alfabética. Quando você executa o <i>applet</i> , três janelas appletviewer aparecem. Clique em cada uma para iniciar a classificação. Note que todas as classificações operam em diferentes velocidades.
SpreadSheet	Demonstra uma planilha simples de linhas e colunas.
SymbolTest	Desenha caracteres do conjunto de caracteres Java.
TicTacToe	Permite ao usuário jogar o jogo-da-velha contra o computador.
WireFrame	Desenha uma forma tridimensional como um aramado. Arraste o mouse para ver a forma de ângulos diferentes.

Fig. 3.1 Os exemplos do diretório **applets**.

plet carregar (discutimos os detalhes de arquivos de HTML na Seção 3.4) e inicia a execução do *applet*. A Fig. 3.2 mostra várias capturas de tela do jogo-da-velha com esse *applet*.

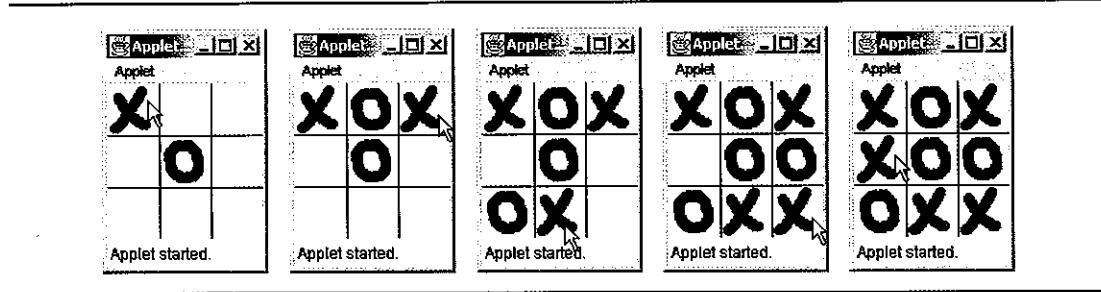


Fig. 3.2 Exemplo de execução do *applet* *tictactoe*.



Dica de teste e depuração 3.3

Se o comando **appletviewer** não funcionar e/ou o sistema indicar que o comando **appletviewer** não pode ser localizado, a variável de ambiente **PATH** pode não estar definida adequadamente em seu computador. Revise as orientações para a instalação do Java 2 Software Development Kit para assegurar que a variável de ambiente **PATH** esteja corretamente definida para seu sistema (em alguns computadores, você pode precisar reiniciar seu computador depois de definir a variável de ambiente **PATH**).

Você é o jogador **X**. Para interagir com o *applet*, aponte o mouse para o quadrado em que você quer colocar um **X** e clique (normalmente, o botão esquerdo do mouse). O *applet* emite um som (pressupondo que seu computador suporte reprodução de áudio) e coloca um **X** no quadrado, se o quadrado estiver aberto. Se o quadrado estiver ocupado, esse é um movimento inválido e o *applet* emite um som diferente indicando que você não pode fazer o movimento especificado. Depois de fazer um movimento válido, o *applet* responde fazendo seu próprio movimento (isso acontece muito rapidamente).

Para jogar novamente, execute outra vez o *applet* clicando no menu **Applet** do *appletviewer* e selecionando o item de menu **Reload** do menu (Fig. 3.3). Para terminar o *appletviewer*, clique no menu **Applet** do *appletviewer* e selecione o item de menu **Quit**.

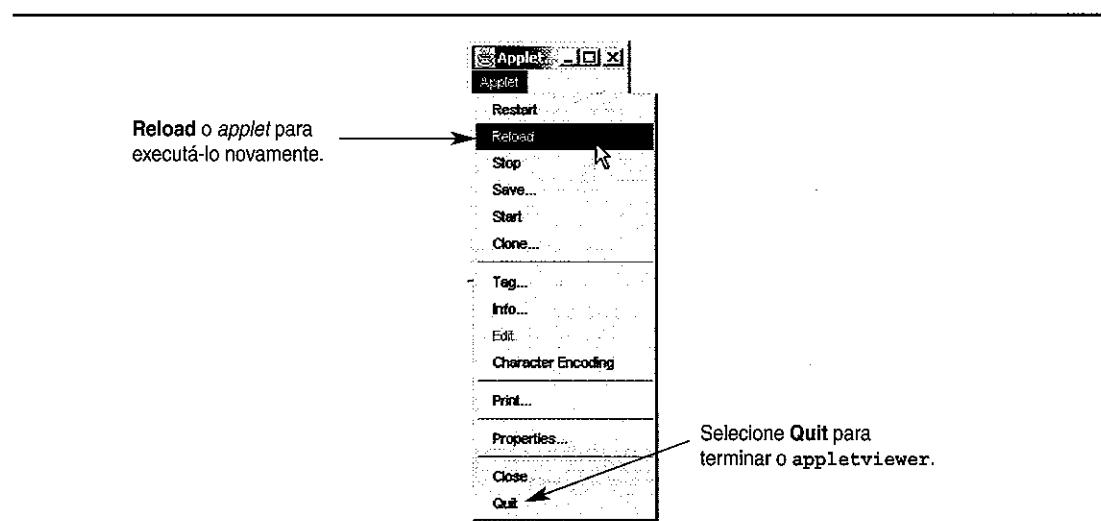


Fig. 3.3 Selecionando **Reload** do menu **Applet** do *appletviewer*.

3.2.2 O applet DrawTest

O próximo *applet* que demonstramos permite que você desenhe linhas e pontos em cores diferentes. Para desenhar, você simplesmente arrasta o mouse no *applet* pressionando um botão e mantendo-o pressionado enquanto arrasta o mouse. Para esse exemplo, altere os diretórios para o diretório **applets**, depois para o subdiretório **DrawTest**. Nesse diretório está o arquivo **example1.html** que é utilizado para executar o *applet*. Na janela de comando, digite o comando

```
appletviewer example1.html
```

e pressione a tecla *Enter*. Isso executa o **appletviewer**. O **appletviewer** carrega o arquivo de HTML especificado como seu argumento de linha de comando (**example1.html** novamente), determina com base no arquivo qual *applet* carregar e inicia a execução do *applet*. A Fig. 3.4 mostra uma captura de tela desse *applet* depois de desenhar algumas linhas e pontos.

A forma *default* para desenhar é uma linha e a cor *default* é o preto, de modo que você pode desenhar linhas pretas arrastando o mouse pelo *applet*. Para arrastar o mouse, pressione e mantenha pressionado o botão e move o mouse. Note que a linha segue o ponteiro do mouse por todo o *applet*. A linha não será permanente enquanto você não liberar o botão do mouse. Você pode então iniciar uma nova linha repetindo o processo.

Selecione uma cor clicando no círculo dentro de um dos retângulos coloridos na parte inferior do *applet*. Você pode selecionar vermelho, verde, azul, cor-de-rosa, alaranjado e preto. Os componentes GUI utilizados para apresentar essas opções são comumente conhecidos como *botões de rádio*. Se imaginar o rádio de um carro, apenas uma estação de rádio pode ser selecionada por vez. De maneira semelhante, somente uma cor de desenho pode ser selecionada por vez.

Tente alterar a forma a ser desenhada de **Lines** para **Points** clicando na seta para baixo à direita da palavra **Lines**, na parte inferior do *applet*. Abre-se uma lista do componente GUI que contém as duas escolhas – **Lines** e **Points**. Para selecionar **Points**, clique na palavra **Points** na lista. O componente GUI fecha a lista e a forma atual é **Points**. Esse componente GUI é comumente conhecido como *escolha*, *caixa de combinação* ou *lista escamoteável (drop-down)*.

Para iniciar um novo desenho, selecione **Reload** do menu **Applet** do **appletviewer**. Para terminar o *applet*, selecione **Quit** do menu **Applet** do **appletviewer**.

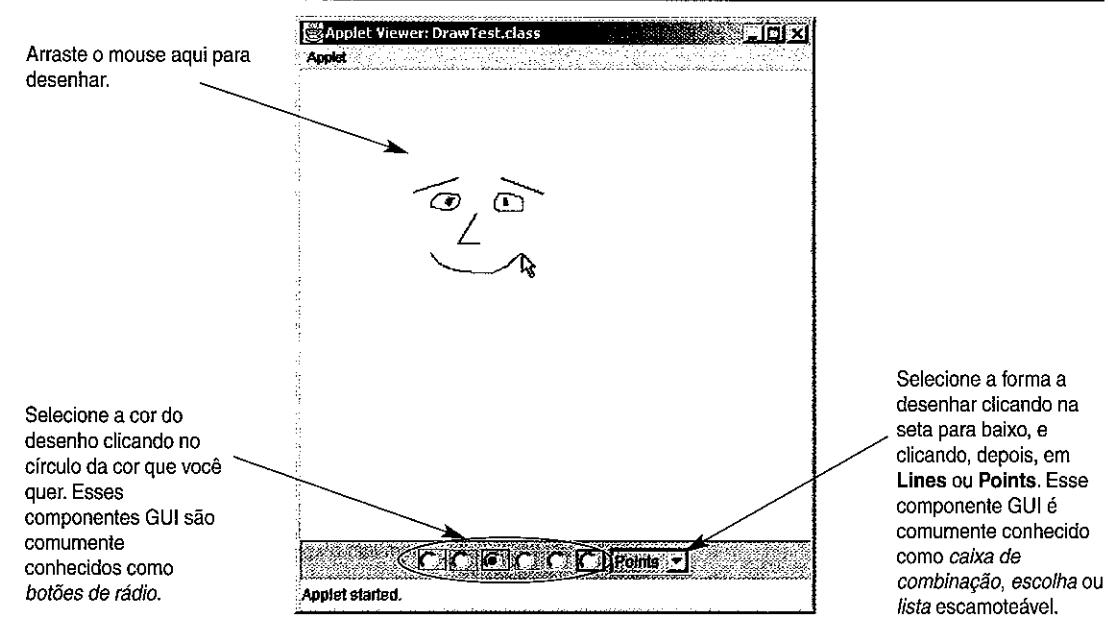


Fig. 3.4 Exemplo de execução do *applet* **DrawTest**.

3.2.3 O applet Java2D

O último *applet* que demonstramos antes de definir nossos próprios *applets* mostra muitas das novas e complexas capacidades de desenho bidimensional embutidas em Java 2 – conhecidas como *API de Java2D*. Para esse exemplo, mude para o diretório **jfc** no diretório **demo** do J2SDK, depois mude para o diretório **Java2D** (você pode mover-se na árvore de diretórios para cima em direção a **demo** utilizando o comando “**cd ..**”, tanto no Windows como no UNIX/Linux). Nesse diretório, há um arquivo HTML (**Java2Demo.html**) que é utilizado para executar o *applet*. Na janela de comando, digite o comando

```
appletviewer Java2Demo.html
```

e pressione a tecla *Enter*. Isso executa o **appletviewer**. O **appletviewer** carrega o arquivo de HTML especificado como seu argumento de linha de comando (**Java2Demo.html**), determina com base no arquivo qual *applet* carregar e inicia a execução do *applet*. Essa demo particular leva algum tempo para carregar pois é bem grande. A Fig. 3.5 mostra uma captura de tela de uma das muitas demonstrações das novas capacidades gráficas bidimensionais deste *applet* Java.

Na parte superior dessa demo, você vê guias que parecem pastas suspensas em um armário de arquivo. Essa demo fornece 11 guias diferentes com vários recursos diferentes em cada guia. Para alterar para uma parte diferente da demo, simplesmente clique em uma das guias. Além disso, tente alterar as opções no canto superior direito do *applet*. Algumas dessas opções afetam a velocidade com que o *applet* desenha as imagens gráficas. Por exemplo, clique na pequena caixa com uma marca de seleção nela (um componente GUI conhecido como *caixa de seleção*) à esquerda da palavra **Anti-Aliasing** para desativar a suavização ou *anti-aliasing* (uma técnica gráfica para produzir imagens gráficas menos irregulares na tela). Quando esse recurso está desativado (isto é, sua *caixa de seleção* está desmarcada), a velocidade da animação aumenta para as formas animadas na parte inferior da demo mostrada na Fig. 3.5. Isto acontece porque leva mais tempo para desenhar uma forma animada exibida com suavização do que uma forma animada sem suavização.

3.3 Um applet Java simples: desenhando um string

Agora, vamos introduzir alguns *applets* de nossa própria autoria. Lembre-se, estamos apenas começando – temos muitos mais tópicos a aprender antes de podermos escrever *applets* semelhantes a esses demonstrados na Seção 3.2. Entretanto, abordaremos muitas técnicas iguais neste livro.

Iniciamos analisando um *applet* simples que simula o programa da Fig. 2.1 exibindo o *string* “**Welcome to Java!**”. O *applet* e sua saída de tela são mostrados na Fig. 3.6. O documento de HTML para carregar o *applet* no **appletviewer** é mostrado e discutido na Fig. 3.7.

Esse programa ilustra vários recursos Java importantes. Analisemos cada linha do programa em detalhes. A linha 9 faz o “trabalho real” do programa, a saber, desenhar o *string* **Welcome to Java Programming!** na tela. Mas vamos analisar cada linha do programa em ordem. As linhas 1 e 2

```
// Fig. 3.6: Welcomeapplet.java
// Um primeiro applet em Java
```

começam com **//**, indicando que o restante de cada linha é um comentário. O comentário na linha 1 indica o número da figura e o nome do arquivo para o código-fonte do *applet*. O comentário na linha 2 simplesmente descreve o propósito do programa.

Como afirmado no Capítulo 2, Java contém muitos fragmentos predefinidos chamados de classes (ou tipos de dados) que são agrupados em pacotes na API Java. A linha 5

```
import java.awt.Graphics; // importa a classe Graphics
```

é uma instrução **import** que diz ao compilador para carregar a classe **Graphics** do pacote **java.awt** para usar neste *applet* Java. A classe **Graphics** permite a um *applet* Java desenhar gráficos como linhas, retângulos, elipses e *strings* de caracteres. Mais adiante no livro, você verá que a classe **Graphics** também permite que os aplicativos façam desenhos.

A linha 8

```
import javax.swing.JApplet; // importa a classe JApplet
```

é uma instrução **import** que diz ao compilador para carregar a classe **JApplet** do pacote **javax.swing**. Quando você cria um *applet* em Java, normalmente importa a classe **JApplet**. Você importa a classe **Graphics** para

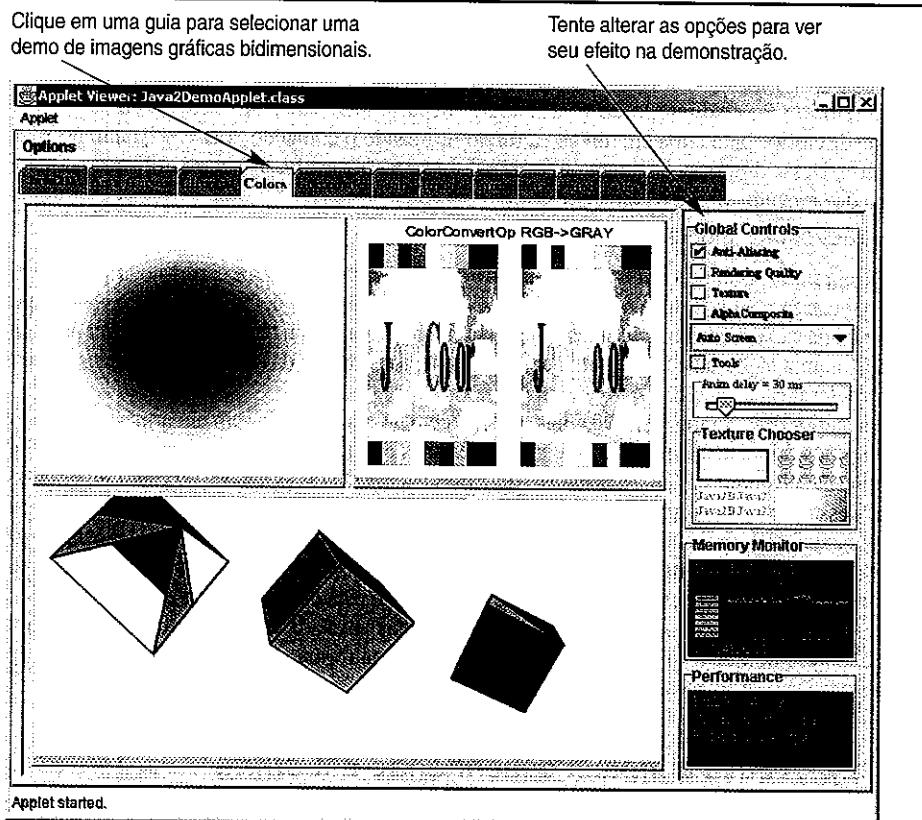


Fig. 3.5 Exemplo de execução do *applet Java2D*.

```

1 // Fig. 3.6: Welcomeapplet.java
2 // Um primeiro applet em Java
3
4 // Pacotes do núcleo de Java
5 import java.awt.Graphics;      // importa a classe Graphics
6
7 // Pacotes de extensão de Java
8 import javax.swing.JApplet;    // importa a classe JApplet
9
10 public class WelcomeApplet extends JApplet {
11
12     // desenha texto sobre o fundo do applet
13     public void paint( Graphics g )
14     {
15         // chama versão herdada do método paint
16         super.paint( g );
17
18         // desenha um String nas coordenadas x=25 e y=25
19         g.drawString( "Welcome to Java Programming!", 25, 25 );
20
21     } // fim do método paint

```

Fig. 3.6 Um primeiro *applet* em Java e a saída na tela do *applet* (parte 1 de 2).

```

22
23 } // fim da classe WelcomeApplet

```

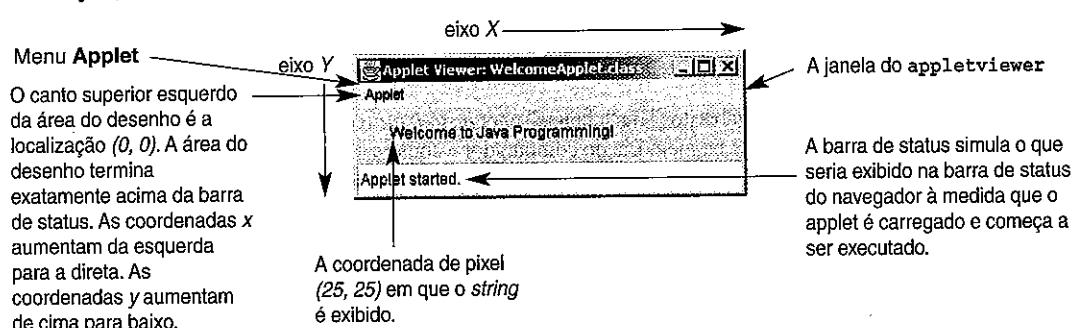


Fig. 3.6 Um primeiro *applet* em Java e a saída na tela do *applet* (parte 2 de 2).

que o programa possa desenhar imagens gráficas (como linhas, retângulos, ovais e *strings* de caracteres) em um *applet* Java (ou aplicativo mais tarde no livro). [Nota: há uma classe mais antiga chamada **Applet** do pacote `java.applet` que não é utilizada com os mais novos componentes da GUI Java do pacote `javax.swing`. Neste livro, utilizamos somente a classe **JApplet** com *applets*.]

Assim como os aplicativos, cada *applet* Java que você cria contém pelo menos uma definição de classe. Um recurso-chave das definições de classe que não foi mencionado no Capítulo 2 é que os programadores raramente criam definições de classes “a partir do zero”. De fato, ao criar uma definição de classe, você normalmente utiliza fragmentos de uma definição de classe existente. Java utiliza *herança* (apresentada na Seção 1.15 e discutida em detalhes no Capítulo 9) para criar novas classes a partir de definições de classe existentes. A linha 10

```
public class WelcomeApplet extends JApplet {
```

inicia uma definição de **class** para a classe **WelcomeApplet**. No fim da linha 10, a chave esquerda, **{**, inicia o corpo da definição da classe. A chave direita correspondente, **}**, na linha 23, termina a definição da classe. A palavra-chave **class** introduz a definição de classe. **WelcomeApplet** é o nome da classe. A palavra-chave **extends** indica que a classe **WelcomeApplet** herda pedaços existentes de uma outra classe. A classe da qual **WelcomeApplet** herda (**JApplet**) aparece à direita de **extends**. Nesse relacionamento de herança, **JApplet** é chamada de *superclasse* ou *classe básica* e **WelcomeApplet** é chamada de *subclasse* ou *classe derivada*. Utilizar herança aqui resulta em uma definição de classe **WelcomeApplet** que tem *atributos* (dados) e *comportamentos* (métodos) da classe **JApplet**, bem como os novos recursos que estamos adicionando em nossa definição de classe **WelcomeApplet** (mais especificamente, a capacidade de desenhar **Welcome to Java Programming!** no *applet*).

Um benefício fundamental de estender a classe **JApplet** é que outra pessoa já definiu “o que significa ser um *applet*”. O **appletviewer** e os navegadores da World Wide Web que suportam *applets* esperam que cada *applet* Java tenha certas capacidades (atributos e comportamentos). A classe **JApplet** já fornece todas essas capacidades – os programadores não precisam “reinventar a roda” e definir todas essas capacidades por conta própria. Na verdade, os contêineres de *applets* esperam que os *applets* tenham mais de 200 métodos diferentes. Em nossos programas feitos até aqui, definimos um método em cada programa. Se tivéssemos que definir mais de 200 métodos apenas para exibir **Welcome to Java Programming!**, nunca criariamos um *applet*, porque levaria tempo demais para definir um! Utilizar **extends** para herdar da classe **JApplet** permite que os programadores de *applets* criem novos *applets* rapidamente.

O mecanismo de herança é fácil de utilizar; o programador não precisa conhecer cada detalhe da classe **JApplet** ou de qualquer outra classe a partir da qual novas classes são herdadas. O programador só precisa saber que a classe **JApplet** define os recursos necessários para criar o *applet* mínimo. Entretanto, para fazer a melhor utilização de qualquer classe, o programador deve estudar todas as capacidades da classe que é estendida.



Boa prática de programação 3.1

Investigue os recursos de qualquer classe na documentação da Java API (java.sun.com/j2se/1.3/docs/api/index.html) cuidadosamente antes de criar uma subclasse por herança a partir dela. Isso ajuda a assegurar que o programador não vai “reinventar a roda”, redefinindo intencionalmente um recurso que já foi fornecido.

As classes são utilizadas como “gabaritos” ou “plantas” para *instanciar* (ou *criar*) objetos para uso em um programa. Um objeto (ou *instância*) reside na memória do computador e contém as informações utilizadas pelo programa. O termo objeto normalmente implica que atributos (dados) e comportamentos (métodos) estão associados ao objeto. Os métodos do objeto utilizam os atributos para fornecer serviços úteis ao *cliente do objeto* (isto é, o código em um programa que chama os métodos).

Quando um contêiner de *applets* (o **appletviewer** ou navegador em que o *applet* é executado) carrega nossa classe **WelcomeApplet**, o contêiner de *applets* cria um objeto (instância) da classe **WelcomeApplet** que implementa os atributos e os comportamentos do *applet*. [Nota: os termos *instância* e *objeto* são freqüentemente utilizados de maneira intercambiável.] Os contêineres de *applets* podem criar somente objetos de classes que sejam **public** e estendam **JApplet**. Portanto, os contêineres de *applets* exigem que a definição da classe comece com a palavra-chave **public** (linha 10). Caso contrário, o contêiner de *applets* não consegue carregar e executar o *applet*. A palavra-chave **public** e as palavras-chave relacionadas (como **private** e **protected**) são discutidas em detalhes no Capítulo 8. Por enquanto, pedimos que você simplesmente inicie todas as definições de classe com a palavra-chave **public** até a discussão de **public** no Capítulo 8.

Quando você salva uma classe **public** em um arquivo, o nome de arquivo deve ser o nome da classe seguido pela extensão de nome de arquivo **.java**. Para nosso *applet*, o nome de arquivo deve ser **WelcomeApplet.java**. Observe que a parte com o nome da classe do nome de arquivo deve ser digitada exatamente da mesma forma que o nome da classe, incluindo o uso idêntico de letras maiúsculas e minúsculas. Para reforçar, repetimos dois erros comuns de programação do Capítulo 2.



Erro comum de programação 3.1

Para uma classe **public** é um erro se o nome de arquivo não for idêntico ao nome da classe (mais a extensão **.java**) tanto na ortografia como no uso de letras maiúsculas e minúsculas. Portanto, também haverá um erro se um arquivo contiver duas ou mais classes **public**.



Erro comum de programação 3.2

É um erro não terminar um nome de arquivo com a extensão **.java** para um arquivo que contenha a definição de classe de um aplicativo. O compilador Java não será capaz de compilar a definição de classe.



Dica de teste e depuração 3.4

A mensagem de erro de compilador “Public class *ClassName* must be defined in a file called *ClassName.java*” indica ou que o nome de arquivo não corresponde exatamente ao nome da classe **public** no arquivo (incluindo todas as letras minúsculas e maiúsculas) ou que você digitou o nome da classe incorretamente ao compilar a classe (o nome deve ser digitado com as letras minúsculas e maiúsculas apropriadas).

A linha 13

```
public void paint( Graphics g )
```

inicia a definição do método **paint** do *applet* – um dos três métodos (comportamentos) que o contêiner de *applets* chama para um *applet* quando o contêiner começa a executar o *applet*. Na ordem, esses três métodos são **init** (discutido mais adiante neste capítulo), **start** (discutido no Capítulo 6) e **paint**. Sua classe de *applet* obtém uma versão “grátis” de cada um desses métodos da classe **JApplet** quando você especifica **extends JApplet** na primeira linha da definição de classe do seu *applet*. Se você não definir estes métodos em seu próprio *applet*, o contêiner de *applets* chama as versões herdadas de **JApplet**. As versões herdadas dos métodos **init** e **start** têm corpos vazios (i. e., seus corpos não contêm instruções, de modo que eles não executam uma tarefa) e a versão herdada do método **paint** não desenha nada no *applet*. [Nota: existem diversos outros métodos que um contêiner de *applets* chama durante a execução de um *applet*. Discutimos estes métodos mais adiante, no Capítulo 6.]

Para permitir que nosso *applet* desenhe, a classe **WelcomeApplet** sobrescreve (substitui ou redefine) a versão **default** de **paint** colocando instruções no corpo de **paint** que desenham uma mensagem na tela. Quando o

contêiner de *applets* diz ao *applet* para “desenhar a si mesmo” na tela, chamando o método **paint**, aparece sua mensagem **Welcome to Java Programming!** em vez de uma tela em branco.

As linhas 13 a 21 são a definição de **paint**. A tarefa do método **paint** é desenhar imagens gráficas (como linhas, ovais e *strings* de caracteres) na tela. A palavra-chave **void** indica que esse método não retorna nenhum resultado quando completa sua tarefa. O conjunto de parênteses depois de **paint** define a *lista de parâmetros* do método. A lista de parâmetros é onde os métodos recebem os dados necessários para realizar suas tarefas. Normalmente, esses dados são passados pelo programador para o método por uma *chamada de método* (também conhecida como *invocar um método ou enviar uma mensagem*). Por exemplo, no Capítulo 2, passamos dados para o método **showMessageDialog** de **JOptionPane**, como a mensagem a exibir ou o tipo de caixa de diálogo. Entretanto, ao escrever *applets*, o programador não chama o método **paint** explicitamente. Em vez disso, o contêiner de *applets* chama **paint** para dizer ao *applet* para desenhar e o contêiner de *applets* passa para o método **paint** as informações que **paint** requer para executar sua tarefa, neste caso um objeto da classe **Graphics** (chamado de **g**). É responsabilidade do contêiner de *applets* criar o objeto de **Graphics** ao qual **g** se refere. O método **paint** usa o objeto de **Graphics** para desenhar imagens gráficas no *applet*. A palavra-chave **public** no começo da linha 13 é necessária a fim de que o contêiner de *applets* possa chamar seu método **paint**. Por enquanto, todas as definições de método devem iniciar com a palavra-chave **public**. Apresentamos outras alternativas no Capítulo 8.

A chave esquerda, {, na linha 14, inicia o corpo do método **paint**. A chave direita correspondente, }, na linha 21, termina o corpo de **paint**.

A linha 16

```
super.paint( g );
```

chama a versão do método **paint** herdada da superclasse **JApplet**¹.

A linha 19

```
g.drawString( "Welcome to Java Programming!", 25, 25 );
```

instrui o computador a realizar uma ação (ou tarefa), a saber, desenhar os caracteres do *string* de caracteres **Welcome to Java Programming!** no *applet*. Essa instrução utiliza o método **drawString** definido pela classe **Graphics** (essa classe define todas as capacidades gráficas de desenho de um programa Java, inclusive desenhar *strings* de caracteres e formas como retângulos, elipses e linhas). A instrução chama o método **drawString** utilizando o objeto **g** de **Graphics** (na lista de parâmetros de **paint**) seguido pelo operador ponto (.), seguido pelo nome de método **drawString**. O nome do método é seguido por um conjunto de parênteses que contém a lista de argumentos de que **drawString** precisa para realizar sua tarefa.

O primeiro argumento para **drawString** é o **String** a desenhar no *applet*. Os últimos dois argumentos na lista – 25 e 25 – são as *coordenadas* (ou *posição*) x-y em que o canto inferior esquerdo do *string* deveria ser desenhado no *applet*. Os métodos de desenho da classe **Graphics** exigem coordenadas para especificar onde desenhar no *applet* (mais adiante no texto demonstramos como desenhar em aplicativos). A primeira coordenada é a *coordenada x* (o número de *pixels* a partir do canto esquerdo do *applet*) e a segunda coordenada é a *coordenada y* (que representa o número de *pixels* a partir do topo do *applet*). As coordenadas são medidas a partir do canto superior esquerdo do *applet* em *pixels* (imediatamente abaixo do menu **Applet** no exemplo de janela de saída da Fig. 3.6). O *pixel* (*picture element*) é a unidade de exibição para a tela do computador. Em uma tela colorida, o *pixel* aparece como um ponto colorido na tela. Muitos computadores pessoais têm 800 *pixels* de largura de tela e 600 *pixels* de altura de tela, para um total de 800 vezes 600 ou 480.000 *pixels* que podem ser exibidos. Muitos computadores de hoje em dia têm resoluções de tela mais altas, isto é, têm mais pixels de largura e altura da tela. O tamanho de um *applet* depende do tamanho e da resolução da tela. Para telas com o mesmo tamanho, o *applet* vai aparecer menor na tela com a mais alta resolução. A resolução baixa mais comum é 640 por 480.

Quando a linha 19 é executada, ela desenha a mensagem **Welcome to Java Programming!** na área de tela do *applet* nas coordenadas 25 e 25. Observe que as aspas que delimitam o *string* de caracteres não são exibidas na tela.

Como comentário colateral, por que você poderia querer cópias grátis dos métodos **init**, **start** e **paint** se eles não executam uma tarefa? A seqüência predefinida de chamada de métodos durante a inicialização feita pelo

¹ Por razões que se tornarão claras mais adiante no texto, esta instrução deve ser a primeira instrução no método **paint** de todos os *applets*. Embora os primeiros exemplos de *applets* funcionem sem esta instrução, omitir esta instrução provoca erros sutis em *applets* mais elaborados que combinem desenho com componentes GUI. Incluir esta instrução agora vai deixá-lo habituado a usá-la e vai economizar tempo e esforço à medida que você construir *applets* mais substanciais mais tarde.

appletviewer ou navegador para todos os *applets* é sempre **init**, **start** e **paint** – isto dá a um programador de *applets* a garantia de que estes métodos serão chamados sempre que um *applet* começa a ser executado. Nem todo *applet* precisa de todos estes três métodos. Entretanto, o **appletviewer** ou navegador espera que cada um destes métodos esteja definido, para que ele possa oferecer uma seqüência de inicialização consistente para um *applet*. [Nota: isto é semelhante ao fato de todos os aplicativos sempre iniciarem por **main**.] Herdar as versões *default* destes métodos garante o navegador que ele pode tratar cada *applet* de maneira uniforme, chamando **init**, **start** e **paint** quando começa a execução do *applet*. Além disso, o programador pode se concentrar apenas na definição dos métodos necessários para um *applet* particular.

3.3.1 Compilando e executando o WelcomeApplet

Assim como ocorre com as classes de aplicativos, você precisa compilar as classes de *applets* antes que elas possam ser executadas. Depois de definir a classe **WelcomeApplet** e salvá-la no arquivo **WelcomeApplet.java**, abra uma janela de comando, mude para o diretório no qual você salvou a definição da classe do *applet* e digite o comando

```
javac WelcomeApplet.java
```

para compilar a classe **WelcomeApplet**. Se não houver erros de sintaxe, os *bytecodes* resultantes são armazenados no arquivo **WelcomeApplet.class**.

Antes de executar o *applet*, você precisa criar um arquivo de *HTML (Hypertext Markup Language)* para carregar o *applet* no contêiner de *applets* (o **appletviewer** ou um navegador). Em geral, o arquivo de *HTML* termina com a extensão de nome de arquivo “*.html*” ou “*.htm*”. Os navegadores exibem o conteúdo dos documentos que contêm texto (também conhecidos como *arquivos de texto*). Para executar um *applet Java*, o arquivo de texto de *HTML* precisa indicar qual *applet* o contêiner de *applets* deve carregar e executar. A Fig. 3.7 mostra um arquivo de *HTML* simples – **WelcomeApplet.html** – que carrega o *applet* definido na Fig. 3.6 no contêiner de *applets*. [Nota: nesta primeira parte do livro, sempre demonstramos *applets* com o contêiner de *applets* **appletviewer**.]

Boa prática de programação 3.2

Sempre teste um *applet Java* no **appletviewer** e certifique-se de que ele esteja rodando corretamente antes de carregar o *applet* em um navegador da World Wide Web. Os navegadores freqüentemente salvam uma cópia do *applet* na memória até a sessão atual do navegador terminar (isto é, até que todas as janelas do navegador sejam fechadas). Portanto, se você alterar, recompilar e depois recarregar o *applet* no navegador, você pode não ver as alterações porque o navegador pode ainda estar executando a versão original do *applet*. Feche todas as janelas de seu navegador para remover a versão antiga do *applet* da memória. Abra uma nova janela de navegador e carregue o *applet* para ver suas alterações.

Observação de engenharia de software 3.1

Se seu navegador de World Wide Web não suporta Java 2, a maioria dos *applets* neste livro não executará no seu navegador. Isso ocorre porque a maioria dos *applets* neste livro utilizam recursos que são específicos para Java 2 ou que não são fornecidos por navegadores que suportam Java 1.1. A Seção 3.6.2 discute como usar o Plug-in Java para visualizar *applets* nos navegadores da Web que não suportam Java 2.

Muitos códigos (ou *marcas*) de *HTML* vêm em pares. Por exemplo, as linhas 1 e 4 da Fig. 3.7 indicam o início e o fim, respectivamente, das marcas de *HTML* no arquivo. Todas as marcas de *HTML* iniciam com um *sinal de menor que*, <, e terminam com um *sinal de maior que*, >. As linhas 2 e 3 são *tags* *HTML* especiais para *applets Java*. Elas instruem o contêiner de *applets* a carregar um *applet* específico e definem o tamanho da área de exibição do *applet* (sua *largura* e *altura* em pixels) no **appletviewer** (ou navegador). Normalmente, o *applet* e seu arquivo de *HTML* correspondente são armazenados no mesmo diretório no disco. Em geral, o navegador carrega um arquivo

```
1 <html>
2 <applet code= "WelcomeApplet.class" width = "300" height = "45">
3 </applet>
4 </html>
```

Fig. 3.7 **WelcomeApplet.html** carrega a classe **WelcomeApplet** da Fig. 3.6 no **appletviewer**.

de HTML de um computador (que não o seu próprio) conectado à Internet. Entretanto, os arquivos de HTML também podem residir em seu computador (como demonstramos na Seção 3.2). Quando o contêiner de *applets* encontra um arquivo de HTML que especifica um *applet* a executar, ele automaticamente carrega o arquivo (ou arquivos) **.class** do *applet* a partir do mesmo diretório no computador no qual o arquivo de HTML está armazenado.

A marca **<applet>** tem vários *atributos*. O primeiro atributo da marca **<applet>** na linha 2 (**code = "WelcomeApplet.class"**) indica que o arquivo **WelcomeApplet.class** contém a classe do *applet* compilada. Lembre-se: quando você compila seus programas Java, cada classe é compilada em um arquivo separado que tem o mesmo nome que a classe e termina com a extensão **.class**. O segundo e terceiro componentes da marca **<applet>** indicam a largura (**width**) e a altura (**height**) do *applet* em pixels. O canto superior esquerdo da área de exibição do *applet* está sempre na coordenada *x* 0 e na coordenada *y* 0. A largura desse *applet* é de 300 pixels e sua altura é de 45 pixels. Você pode querer (ou precisar) utilizar valores maiores de largura e de altura para definir uma área maior de desenho para seus *applets*. A marca **</applet>** (linha 3) termina com a marca **<applet>** que iniciou na linha 2. A marca **</html>** (linha 4) especifica o fim das marcas de HTML que iniciaram na linha 1 com **<html>**.



Observação de engenharia de software 3.2

Geralmente, cada applet deve ter menos de 800 pixels de largura e 640 pixels de altura (a maioria das telas de computador suporta essas dimensões como a largura e altura mínimas).



Erro comum de programação 3.3

Colocar caracteres adicionais como vírgulas (,) entre os atributos na marca **<applet>** pode fazer com que o **appletviewer** ou o navegador produzam uma mensagem de erro que indica uma **MissingResourceException** ao carregar o *applet*.



Erro comum de programação 3.4

Esquecer a marca **</applet>** final impede que o *applet* seja carregado no **appletviewer** ou no navegador adequadamente.



Dica de teste e depuração 3.5

Se você receber uma mensagem de erro **MissingResourceException** ao carregar um *applet* no **appletviewer** ou em um navegador, verifique se há erros de sintaxe na marca **<applet>** no arquivo de HTML cuidadosamente. Compare seu arquivo de HTML com o arquivo na Fig. 3.7 para confirmar a sintaxe adequada.

O **appletviewer** apenas entende as marcas de HTML **<applet>** e **</applet>**, por isso ele é às vezes mencionado como “navegador mínimo” (ele ignora todas as outras marcas de HTML). O **appletviewer** é um lugar ideal para testar um *applet* e se assegurar de que ele rode adequadamente. Uma vez que a execução do *applet* é verificada, você pode adicionar as marcas HTML do *applet* a um arquivo de HTML que será visualizado pelas pessoas que navegam na Internet.

Para executar o **WelcomeApplet** no **appletviewer**, abra uma janela de comando, mude para o diretório que contém o seu *applet* e o arquivo HTML e digite o comando

```
appletviewer WelcomeApplet.html
```

Observe que o **appletviewer** requer um arquivo de HTML para carregar um *applet*. Isso é diferente do interpretador **java** para aplicativos, que exige somente o nome da classe do aplicativo. Além disso, o comando precedente deve ser digitado a partir do diretório em que o arquivo de HTML e o arquivo **.class** do *applet* estão localizados.



Erro comum de programação 3.5

Executar o **appletviewer** com um nome de arquivo que não termina com **.html** ou **.htm** é um erro que impede o **appletviewer** de carregar seu *applet* para execução.



Dica de portabilidade 3.2

Teste seus *applets* em cada navegador utilizado pelas pessoas que o visualizarão. Isso ajudará a assegurar que elas experimentam a funcionalidade que você espera. [Nota: um dos objetivos do Plug-In Java (discutido mais adiante neste livro) é fornecer uma execução de *applet* consistente em muitos navegadores diferentes.]

3.4 Dois applets mais simples: desenhando strings e linhas

Vamos considerar outro *applet*. O *applet* pode desenhar *Welcome to Java Programming!* de várias maneiras. Por exemplo, ele pode usar duas instruções `drawString` no método `paint` para imprimir múltiplas linhas de texto, como na Fig. 3.8. O arquivo de HTML para carregar o *applet* em um contêiner de *applets* está na Fig. 3.9.

```

1 // Fig. 3.8: WelcomeApplet2.java
2 // Exibindo múltiplos strings em um applet.
3
4 // Pacotes do núcleo de Java
5 import java.awt.Graphics;      // importa a classe Graphics
6
7 // Pacotes de extensão de Java
8 import javax.swing.JApplet;    // importa a classe JApplet
9
10 public class WelcomeApplet2 extends JApplet {
11
12     // desenha texto no fundo do applet
13     public void paint( Graphics g )
14     {
15         // chama a versão do método paint herdada
16         super.paint( g );
17
18         // desenha dois strings em posições diferentes
19         g.drawString( "Welcome to", 25, 25 );
20         g.drawString( "Java Programming!", 25, 40 );
21
22     } // fim do método paint
23
24 } // fim da classe WelcomeApplet2

```

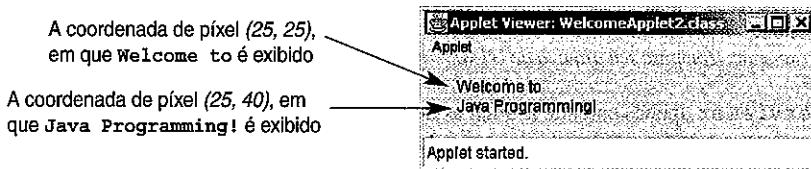


Fig. 3.8 Applet que exibe múltiplos strings.

```

1 <html>
2 <applet code = "WelcomeApplet2.class" width = "300" height = "60">
3 </applet>
4 </html>

```

Fig. 3.9 WelcomeApplet2.html carrega a classe WelcomeApplet2 da Fig. 3.8 no appletviewer.

Observe que cada chamada para o método `drawString` pode desenhar em qualquer posição de pixel no *applet*. A razão das duas linhas de saída aparecerem alinhadas à esquerda, como mostrado na Fig. 3.8, é que ambas usam a mesma coordenada *x* (25). Além disso, cada chamada para o método `drawString` usa coordenadas *y* diferentes (25 na linha 19 e 40 na linha 20), de modo que os *strings* aparecem em posições verticais diferentes no *applet*. Se invertêssemos as linhas 19 e 20 no programa, a janela de saída ainda apareceria como foi mostrado, porque as coordenadas de pixel especificadas em cada instrução `drawString` não dependem das coordenadas especifica-

das em todas as outras instruções `drawString` (e todas as outras operações de desenho). Ao se desenhar gráficos, as linhas de texto não são separadas por caracteres nova linha (como mostrado com o método `println` de `System.out` e o método `showMessageDialog` de `JOptionPane` no Capítulo 2). Na verdade, se você tentar dar saída de um `string` que contém um caractere de nova linha (`\n`), você simplesmente verá uma pequena caixa preta nessa posição no `string`.

Para tornar o desenho mais interessante, o *applet* da Fig. 3.10 desenha duas linhas e um `string`. O arquivo de HTML para carregar o *applet* no `appletviewer` é mostrado na Fig. 3.11.

As linhas 19 e 22 do método `paint`

```
g.drawLine( 15, 10, 210, 10 );
g.drawLine( 15, 30, 210, 30 );
```

utilizam o método `drawLine` da classe `Graphics` para indicar que o objeto `Graphics` a que `g` se refere deve desenhar as linhas. O método `drawLine` exige quatro argumentos que representam os dois pontos terminais da linha no *applet* – a coordenada *x* e a coordenada *y* da primeira extremidade da linha e a coordenada *x* e a coordenada *y* da segunda extremidade da linha. Todos os valores de coordenadas são especificados com relação à coordenada do canto superior esquerdo (*0, 0*) do *applet*. O método `drawLine` desenha uma linha reta entre os dois pontos extremos.

```

1 // Fig. 3.10: Welcomelines.java
2 // Exibindo texto e linhas
3
4 // Pacotes do núcleo de Java
5 import java.awt.Graphics;      // importa a classe Graphics
6
7 // Pacotes de extensão de Java
8 import javax.swing.JApplet;    // importa a classe JApplet
9
10 public class WelcomeLines extends JApplet {
11
12     // desenha linhas e um string no fundo do applet
13     public void paint( Graphics g )
14     {
15         // chama a versão do método paint herdada
16         super.paint( g );
17
18         // desenha linha horizontal de (15, 10) até (210, 10)
19         g.drawLine( 15, 10, 210, 10 );
20
21         // desenha linha horizontal de (15, 30) até (210, 30)
22         g.drawLine( 15, 30, 210, 30 );
23
24         // desenha String entre as linhas, na posição (25, 25)
25         g.drawString( "Welcome to Java Programming!", 25, 25 );
26
27     } // fim do método paint
28
29 } // fim da classe WelcomeLines
```



Fig. 3.10 Desenhando strings e linhas.

```

1 <html>
2 <applet code = "WelcomeLines.class" width = "300" height = "40" >
3 </applet>
4 </html>

```

Fig. 3.11 O arquivo WelcomeLines.html, que carrega a classe WelcomeLines da Fig. 3.10 no appletviewer.

3.5 Outro *applet* Java: adicionando números de ponto flutuante

Nosso próximo *applet* (Fig. 3.12) imita o aplicativo da Fig. 2.9 para adicionar dois inteiros. Entretanto, esse *applet* solicita que o usuário digite dois *números de ponto flutuante* (isto é, números com um ponto decimal como 7,33,

```

1 // Fig. 3.12: AdditionApplet.java
2 // Adicionando dois números de ponto flutuante
3
4 // Pacotes do núcleo de Java
5 import java.awt.Graphics; // importa a classe Graphics
6
7 // Pacotes de extensão de Java
8 import javax.swing.*; // importa o pacote javax.swing
9
10 public class AdditionApplet extends JApplet {
11     double sum; // soma dos valores inseridos pelo usuário
12
13     // inicializa o applet obtendo valores do usuário
14     public void init()
15     {
16         String firstNumber; // primeiro string inserido pelo usuário
17         String secondNumber; // segundo string inserido pelo usuário
18         double number1; // primeiro número a adicionar
19         double number2; // segundo número a adicionar
20
21         // obtém o primeiro número do usuário
22         firstNumber = JOptionPane.showInputDialog(
23             "Enter first floating-point value" );
24
25         // obtém o segundo número do usuário
26         secondNumber = JOptionPane.showInputDialog(
27             "Enter second floating-point value" );
28
29         // converte os números do tipo String para o tipo double
30         number1 = Double.parseDouble( firstNumber );
31         number2 = Double.parseDouble( secondNumber );
32
33         // adiciona números
34         sum = number1 + number2;
35     }
36
37     // desenha os resultados em um retângulo no fundo do applet
38     public void paint( Graphics g )
39     {
40         // chama a versão do método paint herdada
41         super.paint( g );
42
43         // desenha o retângulo começando em (15, 10) que tem 270
44         // pixels de largura e 20 pixels de altura
45         g.drawRect( 15, 10, 270, 20 );

```

Fig. 3.12 Um programa de adição “em ação” (parte 1 de 2).

```

46      // desenha os resultados como um string em (25, 25)
47      g.drawString( "The sum is " + sum, 25, 25 );
48
49  } // fim do método paint
50
51 } // fim da classe AdditionApplet

```

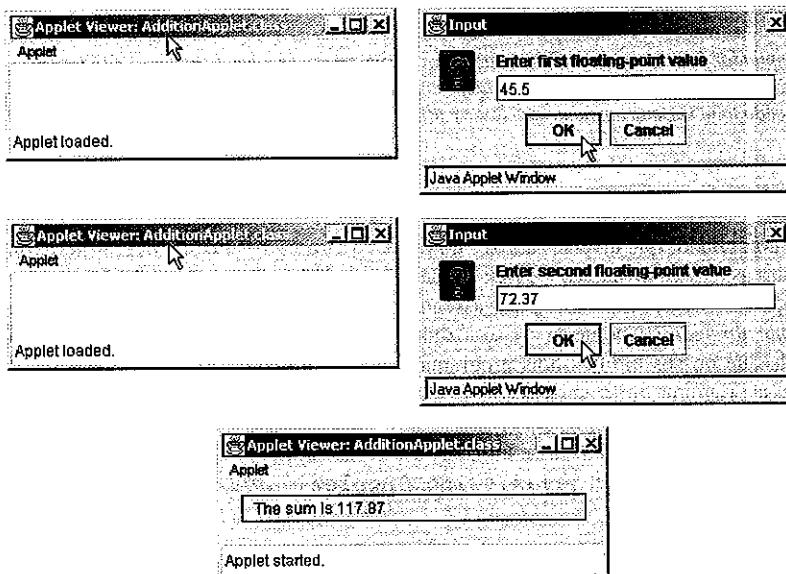


Fig. 3.12 Um programa de adição “em ação” (parte 2 de 2).

0,0975 e 1000,12345). Para armazenar números de ponto flutuante na memória, apresentamos o tipo de dados primitivos **double**, que representa números *de ponto flutuante com precisão dupla*. Há também tipo de dados primitivos **float** para armazenar números *de ponto flutuante de precisão simples*. O **double** requer mais memória para armazenar um valor de ponto flutuante, mas o armazena com aproximadamente duas vezes a precisão de um **float** (15 dígitos significativos para **double** versus sete dígitos significativos para **float**).

Mais uma vez, utilizamos **JOptionPane.showInputDialog** para solicitar entrada do usuário. O *applet* calcula a soma dos valores de entrada e exibe o resultado desenhando um *string* dentro de um retângulo no *applet*. O arquivo de HTML para carregar esse *applet* no **appletviewer** é mostrado na Fig. 3.13.

As linhas 1 e 2

```
// Fig. 3.12: AdditionApplet.java
// Adicionando dois números de ponto flutuante
```

são comentários de uma única linha que declaram o número da figura, o nome do arquivo e o propósito do programa.

A linha 5

```

1 <html>
2 <applet code = "AdditionApplet.class" width = "300" height = "40" >
3 </applet>
4 </html>

```

Fig. 3.13 *AdditionApplet.html* carrega a classe **AdditionApplet** da Fig. 3.12 no *appletviewer*.

```
import java.awt.Graphics; // importa a classe Graphics
```

importa a classe **Graphics** (pacote `java.awt`) para utilizar nesse *applet*. Na verdade, a instrução **import** na linha 5 não é necessária se sempre utilizarmos o nome completo da classe **Graphics** – `java.awt.Graphics` – o qual inclui o nome de pacote e o nome de classe completos. Por exemplo, a primeira linha do método **paint** pode ser definida como

```
public void paint( java.awt.Graphics g )
```



Observação de engenharia de software 3.3

O compilador Java não precisa de instruções **import** em um arquivo de código-fonte Java se o nome completo da classe – o nome completo do pacote e o nome completo da classe (por exemplo, `java.awt.Graphics`) – for especificado toda vez que um nome de classe é utilizado no código-fonte.

A linha 8

```
import javax.swing.*; // importa o pacote javax.swing
```

especifica para o compilador que diversas classes do pacote `javax.swing` são usadas neste *applet*. O asterisco (*) indica que todas as classes no pacote `javax.swing` (como `JApplet` e `JOptionPane`) devem estar disponíveis para o compilador, a fim de que este possa assegurar que estamos utilizando as classes corretamente. Isso permite utilizar o *nome abreviado* (o nome da classe sozinho) de qualquer classe do pacote `javax.swing` no programa. Lembre-se de que nossos últimos dois programas importaram somente a classe `JApplet` do pacote `javax.swing`. Nesse programa, utilizamos as classes `JApplet` e `JOptionPane` daquele pacote. Importar um pacote inteiro para um programa também é uma notação abreviada para que o programador não precise fornecer uma instrução **import** separada para cada classe utilizada desse pacote. Lembre-se de que você sempre pode utilizar o nome completo de cada classe, isto é, `javax.swing.JApplet` e `javax.swing.JOptionPane` em vez de instruções **import**.



Observação de engenharia de software 3.4

O compilador não carrega todas as classes de um pacote quando encontra uma instrução **import** que utiliza a notação * (por exemplo, `javax.swing.*`). O compilador carrega do pacote somente aquelas classes que o programa utiliza.



Observação de engenharia de software 3.5

Muitos pacotes têm subpacotes. Por exemplo, o pacote `java.awt` contém o subpacote `event` para o pacote `java.awt.event`. Quando o compilador encontra uma instrução **import** que utiliza a notação * (por exemplo, `java.awt.*`) para indicar que um programa utiliza múltiplas classes do pacote, o compilador não carrega aquelas classes do subpacote `event`. Portanto, você não pode definir um **import** de `java.*` para procurar classes de todos os pacotes do núcleo de Java.



Observação de engenharia de software 3.6

Ao utilizar instruções **import**, devem ser especificadas instruções **import** separadas para cada pacote utilizado em um programa.



Erro comum de programação 3.6

Assumir que uma instrução **import** para um pacote inteiro (por exemplo, `java.awt.*`) também importa classes de subpacotes nesse pacote (por exemplo, `java.awt.event.*`) resulta em erros de sintaxe para as classes dos subpacotes. Deve haver instruções de importação separadas para cada pacote a partir do qual as classes são utilizadas.

Lembre-se de que os *applets* herdam da classe `JApplet`, e portanto todos eles têm os métodos que um container de *applets* exige para executar o *applet*. A linha 10

```
public class AdditionApplet extends JApplet {
```

inicia a definição da classe `AdditionApplet` e indica que ela herda de `JApplet`.

Todas as definições de classe iniciam com uma chave esquerda de abertura (fim da linha 10), {, e terminam com uma chave direita de fechamento, } (linha 52).



Erro comum de programação 3.7

Se as chaves não ocorrem em pares correspondentes, o compilador indica um erro de sintaxe.

A linha 11

```
double sum; // soma dos valores inseridos pelo usuário
```

é uma declaração de variável de instância – cada instância (objeto) da classe contém uma cópia de cada variável de instância. Por exemplo, se há 10 instâncias desse applet sendo executadas, cada instância tem sua própria cópia de **sum**. Portanto, haveria 10 cópias separadas de **sum** (uma para cada applet). Os programadores declararam as variáveis de instância no corpo da definição de classe, mas fora do corpo de todas as definições de métodos da classe. A declaração precedente declara que **sum** é uma variável do tipo primitivo **double**.

Um benefício importante das variáveis de instância é que todos os métodos da classe podem usar as variáveis de instância. Até agora, declaramos todas as variáveis em um método **main** do aplicativo. As variáveis definidas no corpo de um método são conhecidas como *variáveis locais* e só podem ser utilizadas no corpo do método em que são definidas. Outra distinção entre variáveis de instância e variáveis locais é que as variáveis de instância têm valores *default* e variáveis locais não têm. O valor *default* da variável **sum** é 0.0 porque **sum** é uma variável de instância.



Boa prática de programação 3.3

Inicializar as variáveis de instância explicitamente em vez de contar com a inicialização automática melhora a legibilidade do programa.

O applet da Fig. 3.12 contém dois métodos – **init** (linhas 14 a 35) e **paint** (linhas 38 a 50). Quando um contêiner de applets carrega um applet, o contêiner cria uma instância da classe do applet e chama seu método **init**. O contêiner de applets chama o método **init** só uma vez durante a execução de um applet. O método **init** normalmente *inicializa* as variáveis de instância do applet (se precisam ser inicializadas com um outro valor que não seu valor *default*) e realiza tarefas que devem ocorrer somente uma vez quando o applet começa a ser executado. Como veremos em capítulos mais adiante, o método **init** do applet geralmente cria a interface gráfica com o usuário do applet.



Observação de engenharia de software 3.7

A ordem em que os métodos são definidos em uma definição de classe não tem efeito sobre quando esses métodos são chamados em tempo de execução. Entretanto, seguir convenções para a ordem na qual os métodos são definidos melhora a legibilidade e a facilidade de manutenção de programas.

A primeira linha do método **init** sempre aparece como

```
public void init()
```

indicando que **init** é um método **public** que não retorna informações (**void**) quando ele completa e não recebe argumentos (parênteses vazios depois de **init**) para realizar sua tarefa.

A chave esquerda (linha 15) marca o início do corpo do **init** e a direita correspondente (linha 35) marca o final de **init**. As linhas 16 e 17

```
String firstNumber; // primeiro string inserido pelo usuário
String secondNumber; // segundo string inserido pelo usuário
```

declaram as variáveis **String** locais **firstNumber** e **secondNumber**, nas quais o programa armazena os strings digitados pelo usuário.

As linhas 18 e 19

```
double number1; // primeiro número a adicionar
double number2; // segundo número a adicionar
```

declaram as variáveis locais **number1** e **number2**, do tipo de dados primitivo **double** – essas variáveis armazenam valores de ponto flutuante. Diferentemente de **sum**, **number1** e **number2** não são variáveis de instância, de modo que elas não são inicializadas com 0.0 (o valor *default* para variáveis de instância do tipo **double**).

Como um importante aspecto colateral, há, na realidade, dois tipos de variáveis em Java – *variáveis de tipo de dados primitivos* (normalmente chamadas de *variáveis*) e *variáveis de referência* (normalmente chamadas de *referências*).

rências). Os identificadores `firstNumber` e `secondNumber` são, na verdade, referências – os nomes que são utilizados para se *referir a objetos* no programa. Tais referências, na verdade, contêm o endereço de um objeto na memória do computador. Em nossos *applets* precedentes, o método `paint` realmente recebe uma referência chamada `g` que faz referência a um objeto `Graphics`. As instruções no método `paint` usam aquela referência para enviar mensagens para o objeto `Graphics`. Estas mensagens são chamadas para métodos (como `drawString`, `drawLine` e `drawRect`) que permitem que o programa desenhe. Por exemplo, a instrução

```
g.drawString( "Welcome to Java Programming!", 25, 25 );
```

envia a mensagem `drawString` para o objeto `Graphics` ao qual `g` faz referência. Como parte da mensagem, que é simplesmente uma chamada de método, fornecemos os dados que `drawString` exige para fazer sua tarefa. O objeto `Graphics`, então, desenha o `String` na localização especificada.

Os identificadores `number1`, `number2` e `sum` são nomes de *variáveis*. A variável é semelhante ao objeto. A principal diferença entre a variável e o objeto é que o objeto é definido por uma definição de classe que pode conter tanto dados (variáveis de instância) quanto métodos, ao passo que uma variável é definida por um *tipo de dados primitivo (ou predefinido)* (`char`, `byte`, `short`, `int`, `long`, `float`, `double` ou `boolean`) que pode conter dados. A variável pode armazenar exatamente um valor por vez, ao passo que um objeto pode conter muitos dados individuais. A distinção entre uma variável e uma referência é baseada no tipo de dados do identificador, que é especificado em uma declaração. Se o tipo de dados é um nome de classe, o identificador é uma referência para um objeto e essa referência pode ser utilizada para enviar mensagens (métodos de chamada) para esse objeto. Se o tipo de dados é um dos tipos de dados primitivos, o identificador é uma variável que pode ser utilizada para armazenar na memória ou recuperar da memória um único valor do tipo primitivo declarado.



Observação de engenharia de software 3.8

Uma dica para ajudá-lo a determinar se um identificador é uma variável ou uma referência é o tipo de dados da variável. Por convenção, todos os nomes de classe em Java começam com uma letra maiúscula. Portanto, se o tipo de dados inicia com uma letra maiúscula, normalmente você pode supor que o identificador é uma referência a um objeto do tipo declarado (por exemplo, `Graphics g` indica que `g` é uma referência a um objeto `Graphics`).

As linhas 21 a 23

```
// obtém o primeiro número do usuário
firstNumber = JOptionPane.showInputDialog(
    "Enter first floating-point value" );
```

lêem o primeiro número de ponto flutuante do usuário. O método `showInputDialog` de `JOptionPane` exibe um diálogo de entrada que solicita ao usuário para digitar um valor. O usuário digita um valor no campo de texto do diálogo de entrada e clica no botão **OK** para retornar o *string* que o usuário digitou para o programa. Se você digitar e não aparecer no campo de texto, posicione o ponteiro do mouse no campo de texto e clique para tornar o campo de texto ativo. A variável `firstNumber` é atribuído o resultado da chamada para a operação `JOptionPane.showInputDialog` com uma instrução de atribuição. A instrução é lida como “`firstNumber` obtém o valor de `JOptionPane.showInputDialog("Enter first floating-point value")`”.

Nas linhas 22 e 23, observe a sintaxe de chamada do método. Neste ponto, vimos duas maneiras diferentes de chamar métodos. Esta instrução usa a sintaxe de chamada de método `static` apresentada no Capítulo 2. Todos os métodos `static` são chamados com a sintaxe

```
NomeDeClasse.nomeDeMétodo( argumentos )
```

Também neste capítulo, chamamos métodos da classe `Graphics` com uma sintaxe semelhante que começou com uma referência para um objeto `Graphics`. Generalizando, esta sintaxe é

```
nomeDeReferência.nomeDeMétodo( argumentos )
```

Esta sintaxe é usada para a maioria das chamadas de métodos em Java. Na verdade, o contêiner de *applets* utiliza esta sintaxe para chamar os métodos `init`, `start` e `paint` para seus *applets*.

As linhas 25 a 27

```
// obtém o segundo número do usuário
secondNumber = JOptionPane.showInputDialog(
    "Enter second floating-point value" );
```

lêem o segundo valor de ponto flutuante do usuário exibindo um diálogo de entrada.

As linhas 30 e 31

```
number1 = Double.parseDouble( firstNumber );
number2 = Double.parseDouble( secondNumber );
```

convertem os dois *strings* digitados pelo usuário em valores *double* para uso em um cálculo. O método *Double.parseDouble* (um método de classe *Double* de *static*) converte seu argumento *String* em um valor de ponto flutuante *double*. A classe *Double* está no pacote *java.lang*. O valor de ponto flutuante retornado por *parseDouble* na linha 30 é atribuído à variável *number1*. O valor de ponto flutuante retornado por *parseDouble* na linha 31 é atribuído à variável *number2*.

Observação de engenharia de software 3.9



Cada tipo de dados primitivo (como *int* ou *double*) tem uma classe correspondente (como *Integer* ou *Double*) no pacote *java.lang*. Essas classes (comumente conhecidas como classes empacotadoras de tipo, type-wrappers) fornecem métodos para processamento de valores do tipo de dados primitivo (como converter um *String* em um valor do tipo de dados primitivo ou converter um valor do tipo de dados primitivo em um *String*). Os tipos de dados primitivos não têm métodos. Portanto, os métodos relacionados com um tipo de dados primitivo estão localizados na classe empacotadora de tipo correspondente (p. ex., o método *parseDouble* que converte um *String* em um valor *double* está localizado na classe *Double*). Veja a documentação on-line da API para obter os detalhes completos dos métodos nas classes empacotadoras de tipos.

A instrução de atribuição na linha 34

```
sum = number1 + number2;
```

calcula a soma dos valores armazenados nas variáveis *number1* e *number2* e atribui o resultado à variável *sum* utilizando o operador de atribuição *=*. A instrução é lida como “*sum* recebe o valor de *number1* + *number2*”. Observe que a variável de instância *sum* é utilizada no método *init* mesmo que *sum* não tenha sido definida no método *init*. Podemos usar *sum* em *init* (e em todos os outros métodos da classe), porque *sum* é uma variável de instância.

Nesse ponto, o método *init* do *applet* retorna e o contêiner de *applets* chama o método *start* do *applet*. Não definimos o método *start* nesse *applet*, de modo que o herdado da classe *JApplet* é chamado aqui. Normalmente, o método *start* é utilizado principalmente com um conceito avançado chamado *multithreading*. Veja o Capítulo 15 e o Capítulo 18 para conhecer os usos típicos de *start*.

Em seguida, o contêiner de *applets* chama o método *paint* do *applet*. Nesse exemplo, o método *paint* desenha um retângulo no qual o resultado da adição vai aparecer. A linha 45

```
g.drawRect( 15, 10, 270, 20 );
```

envia a mensagem de *drawRect* para o objeto *Graphics* ao qual *g* faz referência (chama o método *drawRect* do objeto *Graphics*). O método *drawRect* desenha um retângulo baseado em seus quatro argumentos. Os dois primeiros valores inteiros representam a *coordenada x superior esquerda* e a *coordenada y superior esquerda* nas quais o objeto *Graphics* começa a desenhar o retângulo. O terceiro e o quarto argumentos são inteiros não-negativos que representam a *largura* do retângulo em pixels e a *altura* do retângulo em pixels, respectivamente. Essa instrução particular desenha um retângulo que inicia na coordenada (15, 10) que tem 270 pixels de largura e 20 de altura.

Erro comum de programação 3.8



É um erro de lógica fornecer uma largura negativa ou uma altura negativa como argumento para o método *drawRect* de *Graphics*. O retângulo não será exibido e nenhum erro será indicado.

Erro comum de programação 3.9



É um erro de lógica fornecer dois pontos (isto é, pares de coordenadas *x* e *y*) como os argumentos para *drawRect* do método *Graphics*. O terceiro argumento deve ser a largura em pixels e o quarto argumento deve ser a altura em pixels do retângulo a desenhar.

Erro comum de programação 3.10



É normalmente um erro de lógica fornecer argumentos para o método *drawRect* de *Graphics* que façam com que o retângulo seja desenhado fora da área visualizável do applet (isto é, a largura e altura do applet como especificado no documento de HTML que faz referência ao applet). Aumente a largura e altura do applet no documen-

to de HTML ou passe os argumentos para o método `drawRect`, de forma que o retângulo seja desenhado dentro da área visível do applet.

A linha 48

```
g.drawString( "The sum is " + sum, 25, 25 );
```

envia a mensagem `drawString` para o objeto `Graphics` ao qual `g` faz referência (chama o método `drawString` do objeto `Graphics`). A expressão

```
"The sum is " + sum
```

da instrução precedente utiliza o operador de concatenação de `strings +` para concatenar o `string` "The sum is " e `sum` (convertida em um `string`) para criar o `string` que `drawString` exibe. Observe novamente que a instrução precedente utiliza a variável de instância `sum` muito embora `paint` não defina `sum` como variável local.

O benefício de definir `sum` como uma variável de instância é que fomos capazes de atribuir a `sum` um valor em `init` e utilizar o valor de `sum` no método `paint` mais tarde no programa. Todos os métodos de uma classe são capazes de utilizar as variáveis de instância na definição de classe.



Observação de engenharia de software 3.10

As únicas instruções que devem ser colocadas em um método `init` do applet são aquelas que estão diretamente relacionadas com a única inicialização de variáveis de instância de um applet. Os resultados do applet devem ser exibidos a partir de outros métodos da classe de applet. Os resultados que envolvem desenho devem ser exibidos a partir do método `paint` do applet.



Observação de engenharia de software 3.11

As únicas instruções que devem ser colocadas em um método `paint` do applet são aquelas que estão diretamente relacionadas com o desenho (isto é, chamam métodos da classe `Graphics`) e com a lógica do desenho. Geralmente, as caixas de diálogo não devem ser exibidas a partir de um método `paint` do applet.

3.6 Visualizando *applets* em um navegador da Web

Demonstramos diversos *applets* neste capítulo com o contêiner de *applets* `appletviewer`. Como mencionamos, os *applets* também podem ser executados em navegadores da Web habilitados para Java. Infelizmente, existem muitas versões diferentes de navegadores que estão usadas no mundo inteiro. Algumas suportam somente Java 1.0 e muitas suportam Java 1.1; entretanto, poucas suportam a plataforma Java 2. Além disso, mesmo os navegadores que suportam Java 1.1 fazem isto de maneira inconsistente. Na Seção 3.6.1, demonstramos um *applet* que está sendo executado no Netscape Navigator 6, que suporta Java 2. Na Seção 3.6.2, demonstramos como usar o Java Plug-in para executar *applets* Java 2 em outros navegadores da Web, como o Microsoft Internet Explorer ou versões mais antigas do Netscape Navigator.



Dica de portabilidade 3.3

Nem todos os navegadores da Web suportam Java. Aqueles que o fazem freqüentemente suportam versões diferentes e nem sempre são consistentes em todas as plataformas.

3.6.1 Visualizando *applets* no Netscape Navigator 6

Quando você instala o Netscape Navigator 6, um dos componentes do navegador na instalação *default* é Java 2. Uma vez instalado, você pode simplesmente carregar o arquivo HTML de um *applet* no navegador para executar o *applet*. Você pode baixar e instalar o Netscape 6 a partir de

www.netscape.com

clicando no botão **Download** no topo da página da Web.

Após instalar o navegador, abra o programa. No Windows, o Netscape 6 geralmente coloca um ícone em sua área de trabalho durante o processo de instalação. No menu **File**, clique em **Open File ...** para selecionar um documento de HTML do disco rígido de seu computador local. No diálogo **Open File**, vá até a localização do arquivo de HTML da Fig. 3.11. Selecione o nome de arquivo `WelcomeLines.html`, clicando nele, depois clique no

botão **Open** para abrir o arquivo no navegador. Em alguns instantes, você deve ver o *applet* da Fig. 3.10 aparecer na janela do navegador, como mostrado na Fig. 3.14.

3.6.2 Visualizando *applets* em outros navegadores usando o Java Plug-in

Se você gostaria de usar os recursos da plataforma Java 2 em um *applet* e executar aquele *applet* em um navegador que não suporta Java 2, a Sun oferece o Java Plug-in para contornar o suporte a Java de um navegador e usar uma versão completa do *Java 2 Runtime Environment (J2RE* – ambiente de execução Java 2) que está instalado no computador local do usuário. Se o J2RE ainda não existe na máquina cliente, ele pode ser baixado e instalado dinamicamente.

Dica de desempenho 3.1



Devido ao tamanho do Java Plug-in, é difícil e ineficiente baixar o Plug-in para usuários que têm conexões com a Internet mais lentas. Por este motivo, o Plug-in é ideal para intranets corporativas, em que os usuários estão conectados a uma rede de alta velocidade. Uma vez que o Plug-in tenha sido baixado, ele não precisa ser baixado novamente.

Você precisa indicar, no arquivo HTML que contém o *applet*, que o navegador deve usar o Java Plug-in para executar o *applet*. Para tanto, você precisa converter as marcas `<applet>` e `</applet>` em marcas que carreguem o Java Plug-in e executem o *applet*. A Sun fornece um utilitário de conversão chamado de *Java Plug-in 1.3 HTML Converter*² que executa a conversão para você. Informações completas sobre como baixar e usar o Java Plug-in e o HTML Converter estão disponíveis no site da Web

java.sun.com/products/plugin/

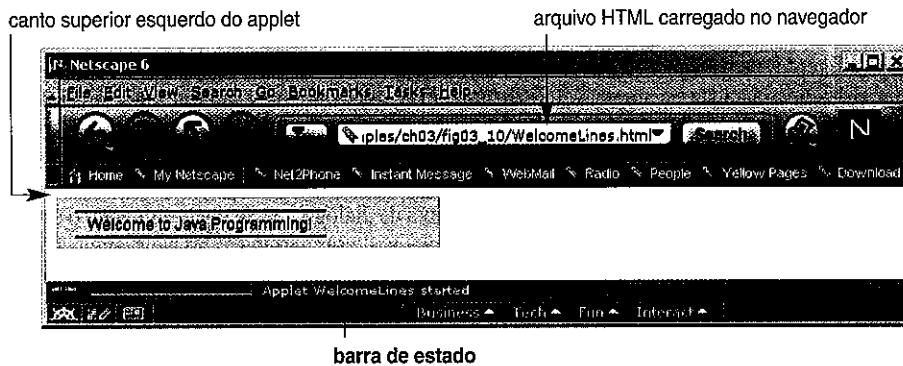


Fig. 3.14 Applet da Fig. 3.10 sendo executado no Netscape Navigator 6.

Depois que você baixou e instalou o Java Plug-in HTML Converter, você pode executá-lo através do arquivo de comandos **HTMLConverter.bat** no Windows ou do script de shell **HTMLConverter.sh** no Linux/UNIX. Estes arquivos estão localizados no subdiretório **classes** do diretório **converter**. A Fig. 3.15 mostra a janela do **Java Plug-in HTML Converter**.

Para executar a conversão, você precisa selecionar o diretório que contém os arquivos HTML a converter. Você pode digitar o nome do diretório no campo de texto abaixo de **All Files in Folder**, ou pode selecionar o

² No Java 2 Software Development Kit versão 1.3.1, uma versão de linha de comando do Java Plug-in HTML Converter é uma das ferramentas do J2SDK. Para usar a versão de linha de comando, abra uma janela de comando e mude de diretório para a posição que contém o arquivo HTML a converter. Neste diretório, digite **HTMLConverter nomeDeArquivo**, onde *nomeDeArquivo* é o arquivo HTML a converter. Visite o endereço java.sun.com/products/plugin/1.3/docs/htmlconv.html para obter mais detalhes sobre o HTML Converter de linha de comando.

O Java Plug-in HTML Converter permite converter todos os arquivos HTML que contêm applets em um diretório. Clique no botão **Browse...** para selecionar o diretório que contém os arquivos a converter.

Além disso, você pode especificar o diretório no qual os arquivos HTML originais são salvos.

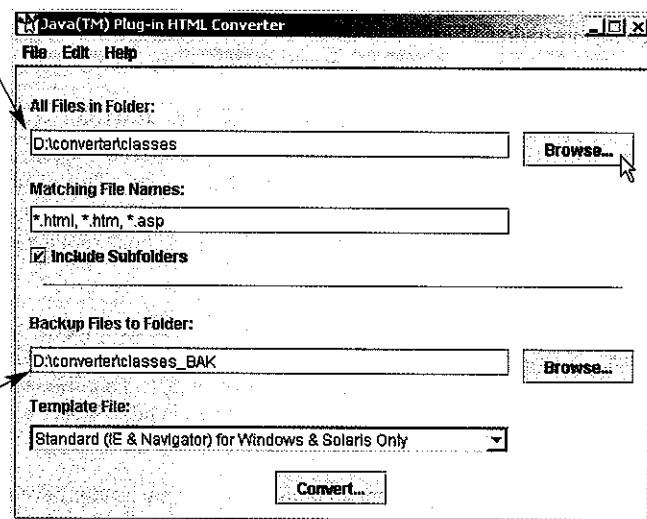


Fig. 3.15 Janela do Java Plug-in HTML Converter.

diretório clicando no botão **Browse...** à direita do campo de texto. Clicamos do botão **Browse...** para exibir o diálogo **Open** na Fig. 3.16.

Depois de selecionar o diretório que contém os arquivos a converter, a janela do **Java Plug-in HTML Converter** fica com a aparência mostrada na Fig. 3.17. O conversor oferece diversos gabaritos de conversão para suportar combinações diferentes de navegadores. O gabarito *default* suporta o Netscape Navigator e o Microsoft Internet Explorer. A Fig. 3.17 mostra a lista escamoteável **Template File**, que contém os gabaritos de conversão predefinida. Selecionei o gabarito *default*, que permite ao Netscape Navigator e ao Microsoft Internet Explorer usar o *plug-in* para executar um *applet*.

Depois de selecionar o arquivo de gabarito apropriado, clique no botão **Convert...** na parte inferior da janela do **Java Plug-in HTML Converter**. A Fig. 3.18 mostra a caixa de diálogo que aparece, contendo o estado e os resultados da conversão. A esta altura, o arquivo HTML do *applet* pode ser carregado no Netscape Navigator ou no Microsoft Internet Explorer para executar o *applet*. Se o Java 2 Runtime Environment ainda não existe no computador do usuário, o arquivo HTML convertido contém informações que permitem ao navegador interagir com os usuários para determinar se eles gostariam de baixar o *plug-in*.

A caixa de diálogo **Open** permite selecionar o diretório que contém os arquivos HTML a converter.

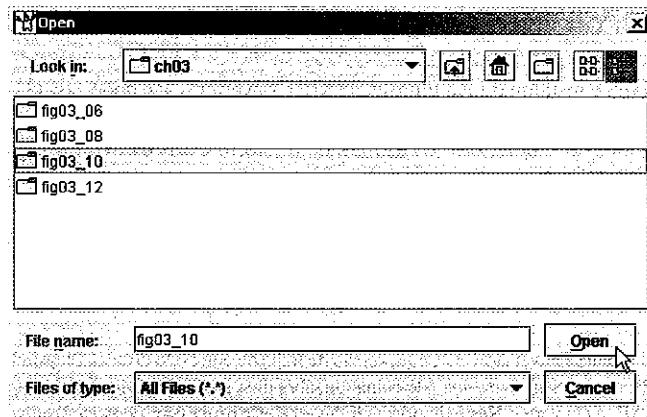


Fig. 3.16 Selecionando o diretório que contém os arquivos HTML a converter.

Neste capítulo e no Capítulo 2 apresentamos muitos recursos importantes de Java, incluindo aplicativos, *applets*, exibição de dados na tela, leitura de dados do teclado, execução de cálculos e tomada de decisões. No Capítulo 4, baseamo-nos nestes recursos à medida que apresentamos a *programação estruturada*. Aqui, você vai se familiarizar mais com as técnicas de recuo, também chamada de indentação. Também estudamos como especificar e variar a ordem na qual um programa executa instruções – esta ordem é chamada de *fluxo de controle*.

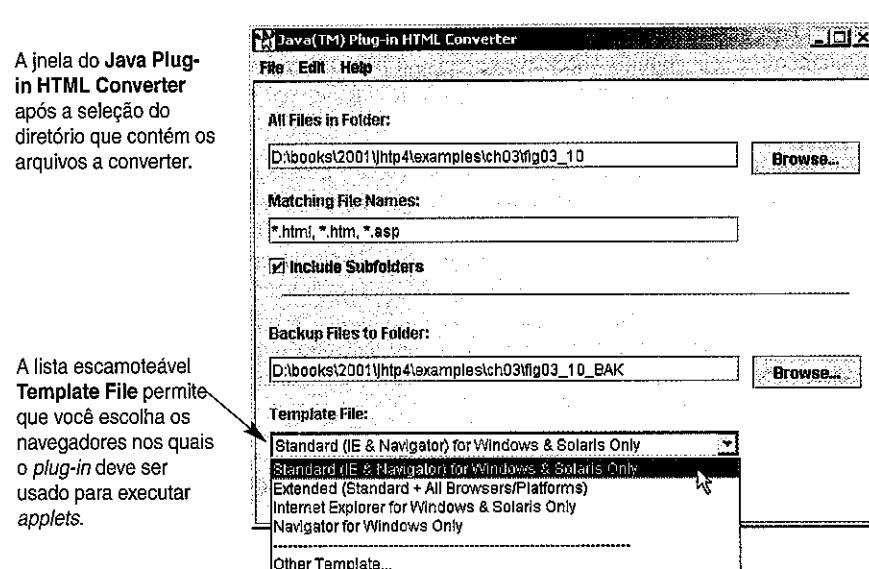


Fig. 3.17 Selezionando o gabarito usado para converter os arquivos HTML.

O diálogo de confirmação mostrando que o conversor encontrou e converteu um applet em um arquivo HTML no diretório especificado.

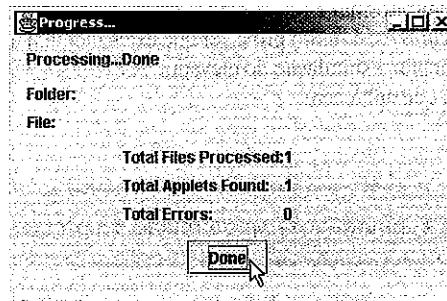


Fig. 3.18 Diálogo de confirmação após a conversão ter terminado.

3.7 Recursos para applets Java na Internet e na World Wide Web

Se você tem acesso à Internet e à World Wide Web, há um grande número de recursos para os *applets* Java disponíveis. O melhor lugar para começar é na fonte – o site de Java da Sun Microsystems, Inc. java.sun.com. No canto superior esquerdo da página da Web há um *hyperlink* **Applets** que o leva para a página da Web

java.sun.com/applets

Essa página contém uma variedade de recursos de *applets* Java, incluindo *applets* gratuitos que você pode utilizar em seu próprio site da World Wide Web, os *applets* de demonstração do J2SDK e uma variedade de outros *applets* (muitos dos quais podem ser baixados e utilizados em seu próprio computador). Há também uma seção intitulada "Applets at Work", na qual você pode ler sobre utilizações de *applets* no setor.

No site da Web da Sun Microsystems, visite *Java Developer Connection*

developer.java.sun.com/developer

Esse site gratuito inclui suporte técnico, fóruns de discussão, cursos de treinamento *on-line*, artigos técnicos, recursos, anúncios de novos recursos Java, acesso de primeira mão a novas tecnologias Java e links para outros importantes sites da Web de Java. Embora o site seja gratuito, você deve se registrar para utilizá-lo.

Outro site da Web útil é JARS – originalmente chamado *Java Applet Rating Service*. O site JARS

www.jars.com

autodenomina-se o "#1 Java Review Service" (o serviço de revisão de Java nº1). Esse site era originalmente um grande repositório Java para *applets*. Seu benefício era que avaliava cada *applet* registrado no site como entre os 1%, os 5% e os 25% mais baixados, de modo que você podia ver os melhores *applets* na Web. No início dos tempos do desenvolvimento da linguagem Java, ter seu *applet* avaliado aqui era uma excelente maneira de demonstrar suas capacidades de programação em Java. JARS é agora um recurso abrangente para programadores de Java.

Os recursos listados nesta seção fornecem hyperlinks para muitos outros sites da Web relacionados a Java. Se você tem acesso à Internet, passe algum tempo navegando por esses sites, executando *applets* e lendo o código-fonte dos *applets* quando disponível. Isso o ajudará a expandir rapidamente seu conhecimento de Java. O Apêndice B contém muitos outros recursos para Java baseados na Web.

3.8 (Estudo de caso opcional) Pensando em objetos: identificando as classes em uma definição de problema

Começamos agora a substancial tarefa de projetar o modelo de simulação do elevador, que representa o funcionamento do sistema de elevador. Projetaremos a interação com o usuário e a exibição deste modelo na Seção 12.16 e na Seção 22.9, respectivamente.

Identificando as classes em um sistema

A primeira etapa em nosso processo de OOD é identificar as classes em nosso modelo. Em algum momento vamos descrever estas classes de uma maneira formal e implementá-las em Java. Primeiro, revisamos a definição do problema e localizamos todos os substantivos; é muito provável que estes incluam a maioria das classes (ou instâncias de classes) necessárias para implementar a simulação do elevador. A Fig. 3.19 é uma lista destes substantivos (e frases com substantivos) na ordem em que aparecem.

Substantivos (e frases com substantivos) na definição do problema

empresa	sistema de elevador	interface gráfica com o usuário (GUI)
edifício de escritórios	poço do elevador	carro do elevador
elevador	painel	pessoa
aplicativo simulador em software	modelo	andar (primeiro andar; segundo andar)
passageiro	campainha dentro do elevador	botão GUI do primeiro andar (First Floor)
porta do andar	luz naquele andar	botão GUI do segundo andar (Second Floor)
usuário de nosso aplicativo	energia	áudio
botão do andar	capacidade	música de elevador
botão do elevador		

Fig. 3.19 Lista de substantivos na definição do problema.

Escolhemos somente os substantivos que desempenham tarefas importantes em nosso modelo. Por esta razão, omitimos diversos substantivos (o próximo parágrafo explica porque cada um é omitido):

- empresa
- edifício de escritórios
- painel
- interface gráfica com o usuário (GUI)
- usuário de nosso aplicativo
- energia
- capacidade
- botões GUI **First Floor** e **Second Floor**
- áudio
- música de elevador

Não precisamos modelar “empresa” como uma classe porque a empresa não faz parte da simulação; a empresa simplesmente quer que nós modelemos o elevador. Não modelamos o “edifício de escritórios”, ou o local onde o elevador está fisicamente situado, porque o edifício não afeta como nossa simulação de elevador opera. As frases “painel”, “áudio” e “música de elevador” fazem parte da apresentação do modelo, mas não pertencem ao modelo em si. Usamos estas frases quando construímos a apresentação na Seção 22.9 e no Apêndice I. As frases “interface gráfica com o usuário”, “usuário de nosso aplicativo” e “botões GUI **First Floor** e **Second Floor**” dizem respeito a como o usuário controla o modelo, mas elas não representam o modelo. Usamos estas frases quando construímos a interface com o usuário na seção 12.16. “Capacidade” é uma propriedade do elevador e do andar – não uma entidade autônoma separada. Finalmente, embora estejamos economizando energia com a política de não movimentar o elevador até que seja pedido, não modelamos “energia”.

Determinamos as classes para nosso sistema agrupando os substantivos restantes em categorias. Descartamos “sistema de elevador” por enquanto – concentrarmo-nos em projetar somente o modelo do sistema e desconsiderarmos como este modelo se relaciona com o sistema como um todo. (Discutimos o sistema como um todo na Seção 13.17.) Usando esta lógica, descartamos “simulação”, porque a simulação é o sistema em nosso estudo de caso. Finalmente, juntamos “elevador” e “carro do elevador” em “elevador”, porque a definição do problema usa as duas palavras de forma intercambiável. Cada substantivo remanescente da Fig. 3.19 se refere a uma ou mais das seguintes categorias:

- modelo
- poço do elevador
- elevador
- pessoa
- andar (primeiro andar, segundo andar)
- porta do elevador
- porta do andar
- botão do elevador
- botão do andar
- campainha
- luz

Estas categorias são provavelmente as classes que precisaremos implementar para nosso sistema. Observe que criamos uma categoria para os botões dos andares e uma categoria para o botão do elevador. Os dois tipos de botões

desempenham tarefas diferentes em nossa simulação – os botões nos andares chamam o elevador e o botão no elevador informa ao elevador para se mover para o outro andar.

Podemos agora modelar as classes em nosso sistema com base nas categorias que criamos. Por convenção, vamos usar letras maiúsculas para os nomes de classes no processo de projeto (como faremos quando escrevermos o programa Java que implementa nosso projeto). Se o nome de uma classe contiver mais de uma palavra, juntamos as palavras e começamos cada palavra com letra maiúscula (por exemplo, `NomeComVariasPalavras`). Usando esta convenção, criamos as classes `ElevatorModel`³, `ElevatorShaft`, `Elevator`, `Person`, `Floor`, `ElevatorDoor`, `FloorDoor`, `ElevatorButton`, `FloorButton`, `Bell` e `Light`^{*}. Construímos nosso sistema usando todas estas classes como blocos de construção. Antes de começarmos a construir o sistema, no entanto, precisamos entender melhor como as classes se relacionam entre si.

Diagramas de classes

A UML possibilita modelar, através de *diagramas de classes*, as classes no sistema de elevador e seus relacionamentos. Os diagramas de classes modelam a estrutura do sistema, fornecendo as classes, ou “blocos de construção”, do sistema. A Fig. 3.20 representa a classe `Elevator` usando a UML. Em um diagrama de classes, cada classe é modelada como um retângulo. Dividimos, então, este retângulo em três partes. A parte superior contém o nome da classe. A parte do meio contém os *atributos* da classe. (Discutimos atributos na Seções 4.14 e 5.11 de “Pensando em objetos”.) A parte inferior contém as *operações* da classe (discutidas na Seção 6.17 de “Pensando em objetos”).

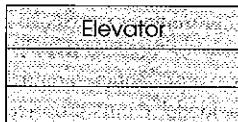


Fig. 3.20 Representando uma classe na UML.

As classes se relacionam uma com as outras através de *associações*. A Fig. 3.21 mostra como nossas classes `ElevatorShaft`, `Elevator` e `FloorButton` se relacionam umas com as outras. Observe que os retângulos neste diagrama não estão subdivididos em três seções. A UML permite a supressão dos atributos e operações de classes desta maneira para criar diagramas mais legíveis. Diz-se que um diagrama como este é um *diagrama elidido* (i.e., algumas informações, como o conteúdo dos compartimentos do meio e inferior), não estão modeladas. Colocamos informações nestes compartimentos na Seção 4.14 e na Seção 6.17, respectivamente.

Neste diagrama de classes, uma linha cheia que conecta classes representa uma *associação*. Uma associação é um relacionamento entre classes. Os números junto às linhas expressam valores de *multiplicidade*. Os valores de multiplicidade indicam quantos objetos de uma classe participam da associação. Pelo diagrama, vemos que dois objetos da classe `FloorButton` participam da associação com um objeto da classe `ElevatorShaft`, porque os dois botões de andar estão localizados no `ElevatorShaft`. Portanto, a classe `FloorButton` têm um relacionamento *dois para um* com a classe `ElevatorShaft`; também podemos dizer que a classe `ElevatorShaft` tem um relacionamento *um para dois* com a classe `FloorButton`. Também vemos que a classe `ElevatorShaft` tem um relacionamento *um para um* com a classe `Elevator` e vice-versa. Usando a UML, podemos modelar muitos tipos de multiplicidade. A Fig. 3.22 mostra os tipos de multiplicidade e como representá-los.

³ Quando nos referimos ao “modelo do elevador”, referimo-nos implicitamente a todas as classes que compõem o modelo que descreve a operação de nosso sistema de elevador – em outras palavras, em nossa simulação, diversas classes compõem o modelo. Veremos na Seção 13.17 que nosso sistema exige uma única classe para representar o modelo – criamos a classe `ElevatorModel` para agir como a “representante” do modelo, porque, como veremos na Fig. 3.23, `ElevatorModel` é a classe que agrupa todas as outras classes que compõem o modelo.

* N. de R. Modelo do elevador, poço do elevador, elevador, pessoa, andar, porta do elevador, porta do andar, botão do elevador, botão do andar, campainha e luz, respectivamente.

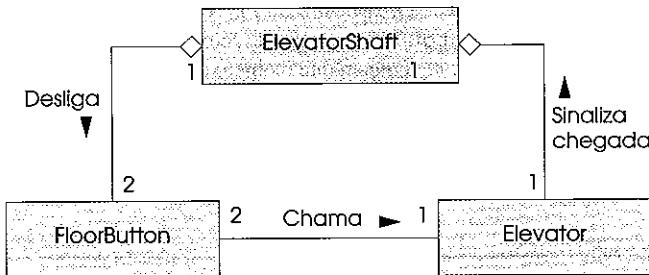


Fig. 3.21 Diagrama de classes mostrando as associações entre classes.

Símbolo	Significado
0	Nenhum.
1	Um.
<i>m</i>	Um valor inteiro.
0..1	Zero ou um.
<i>m, n</i>	<i>m ou n</i>
<i>m..n</i>	No mínimo <i>m</i> , mas não mais do que <i>n</i> .
*	Qualquer inteiro não-negativo.
0..*	Zero ou mais.
1..*	Um ou mais.

Fig. 3.22 Tipos de multiplicidade.

Uma associação pode ter um nome. Por exemplo, a palavra **Chama** acima da linha que conecta as classes **FloorButton** e **Elevator** indica o nome daquela associação – a seta mostra a direção da associação. Esta parte do diagrama é lida assim: “Um objeto da classe **FloorButton** chama um objeto da classe **Elevator**”. Observe que as associações são direcionais, sendo a direção indicada pela seta junto ao nome da associação – de modo que seria inadequado, por exemplo, ler a associação precedente como “o objeto da classe **Elevator** chama um objeto da classe **FloorButton**”. Além disso, a palavra **Desliga** indica que “o objeto da classe **Elevator** desliga dois objetos da classe **FloorButton**”. Finalmente, a frase **Sinaliza chegada** indica que “o objeto da classe **Elevator** sinaliza a chegada do objeto da classe **ElevatorShaft**”.

O losango colocado nas linhas de associação da classe **ElevatorShaft** indica que a classe **ElevatorShaft** tem um relacionamento de *agregação* com as classes **FloorButton** e **Elevator**. A agregação implica em um relacionamento todo/parte. A classe que tem o símbolo de agregação (o losango vazio) em sua extremidade de uma linha de associação é o todo (neste caso, **ElevatorShaft**) e a classe na outra extremidade da linha de associação é a parte (neste caso, as classes **FloorButton** e **Elevator**). Neste exemplo, o poço do elevador “tem um” elevador e dois botões de andar. O relacionamento “tem um” define a agregação (veremos na Seção 9.23 que o relacionamento “é um” define a herança).

A Fig. 3.23 mostra o diagrama de classes completo para o modelo do elevador. Modelamos todas as classes que criamos, bem como as associações entre estas classes. [Nota: no Capítulo 9, expandimos nosso diagrama de classes usando o conceito orientado a objetos de *herança*].

A classe **ElevatorModel** está representada próximo ao topo do diagrama e agrupa um objeto da classe **ElevatorShaft** e dois objetos da classe **Floor**. A classe **ElevatorShaft** é uma agregação de um objeto da clas-

se **Elevator** e dois objetos de cada uma das classes **Light**, **FloorDoor** e **FloorButton**. (Observe os relacionamentos dois para um entre cada uma dessas classes e **ElevatorShaft**). A classe **Elevator** é uma agregação das classes **ElevatorDoor**, **ElevatorButton** e **Bell**. A classe **Person** tem associações tanto com **FloorButton** como com **ElevatorButton** (e outras classes, como veremos em breve). O nome de associação **Pressiona** e as setas de *direção do nome* indicam que o objeto da classe **Person** pressiona estes botões. O objeto da classe **Person** também anda no objeto da classe **Elevator** e caminha através do objeto da classe **Floor**. O nome **Chama** indica que um objeto da classe **FloorButton** chama o objeto da classe **Elevator**. O nome **Sinaliza para movimentar** indica que o objeto da classe **ElevatorButton** sinaliza ao objeto da classe **Elevator** para se movimentar até o outro andar. O diagrama também indica muitas outras associações.

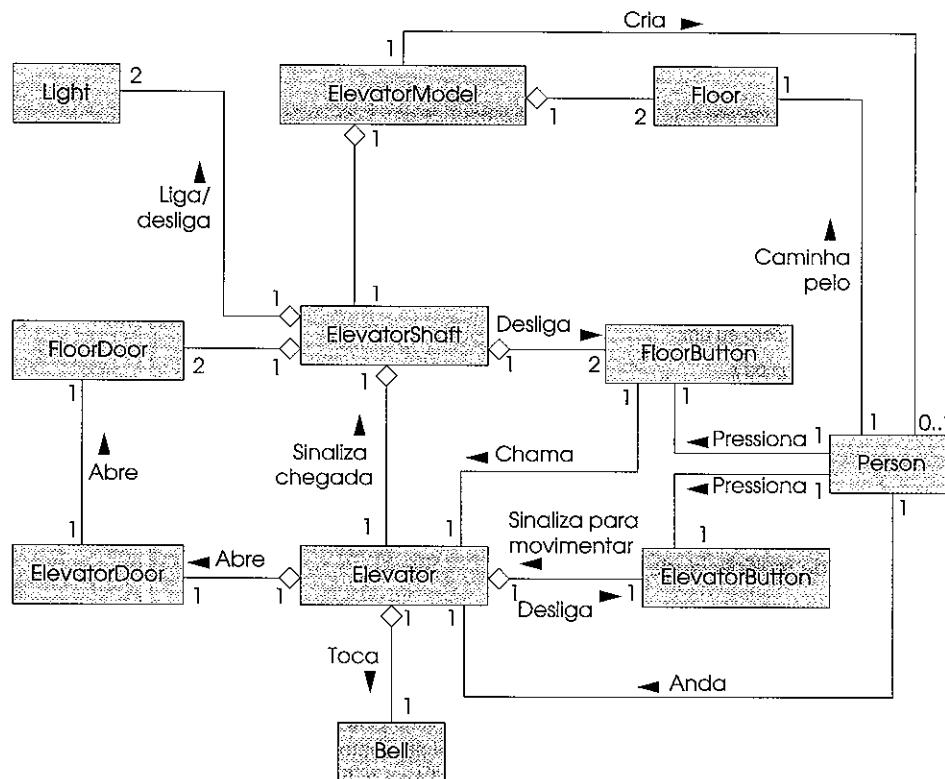


Fig. 3.23 Diagrama de classes para o modelo do elevador.

Diagramas de objetos

A UML também define *diagramas de objetos*, que são similares aos diagramas de classes no sentido de que os dois modelam a estrutura do sistema. Entretanto, diagramas de objetos modelam objetos (instâncias de classes) *em um momento específico* na execução do programa. Os diagramas de objetos apresentam uma fotografia da estrutura enquanto o sistema está em execução, fornecendo informações sobre quais objetos estão participando do sistema em um determinado instante no tempo. Os diagramas de objetos representam relacionamentos entre objetos como linhas cheias – estes relacionamentos chamam-se *vínculos*.

A Fig. 3.24 modela uma fotografia do sistema quando ninguém está no edifício (i.e., nenhum objeto da classe **Person** existe no sistema neste instante). Os objetos normalmente são escritos na forma nomeDoObjeto:NomeDaClasse – nomeDoObjeto se refere ao nome do objeto e NomeDaClasse se refere à classe à qual aque-

le o objeto pertence. Todos os nomes em um diagrama de objetos são sublinhados. A UML permite omitir o nome do objeto para os objetos no diagrama dos quais existe somente um objeto daquela classe (p. ex., um objeto da classe **Bell** na parte inferior do diagrama). Em sistemas grandes, diagramas de objetos podem conter muitos objetos. Isso pode dar origem a diagramas congestionados, difíceis de ler. Se o nome de um objeto em particular é desconhecido ou se não é necessário incluir o nome (i.e., preocupamo-nos somente com o tipo do objeto), podemos desconsiderar o nome do objeto e mostrar somente o dois-pontos e o nome da classe.

Agora já identificamos as classes para nosso sistema (embora possamos descobrir outras nas fases posteriores do processo de projeto). Na Seção 4.14 de “Pensando em objetos”, determinamos os atributos para cada uma destas classes e, na Seção 5.11 de “Pensando em objetos”, usamos estes atributos para examinar como o sistema muda ao longo do tempo e para apresentar seus aspectos comportamentais. À medida que expandirmos nossos conhecimentos, descobriremos novas informações que nos possibilitarão descrever nossas classes de maneira mais completa. Como o mundo real é inherentemente orientado a objetos, ser-lhe-á bastante natural prosseguir neste projeto, embora você possa ter começado há pouco tempo seu estudo de orientação a objetos.

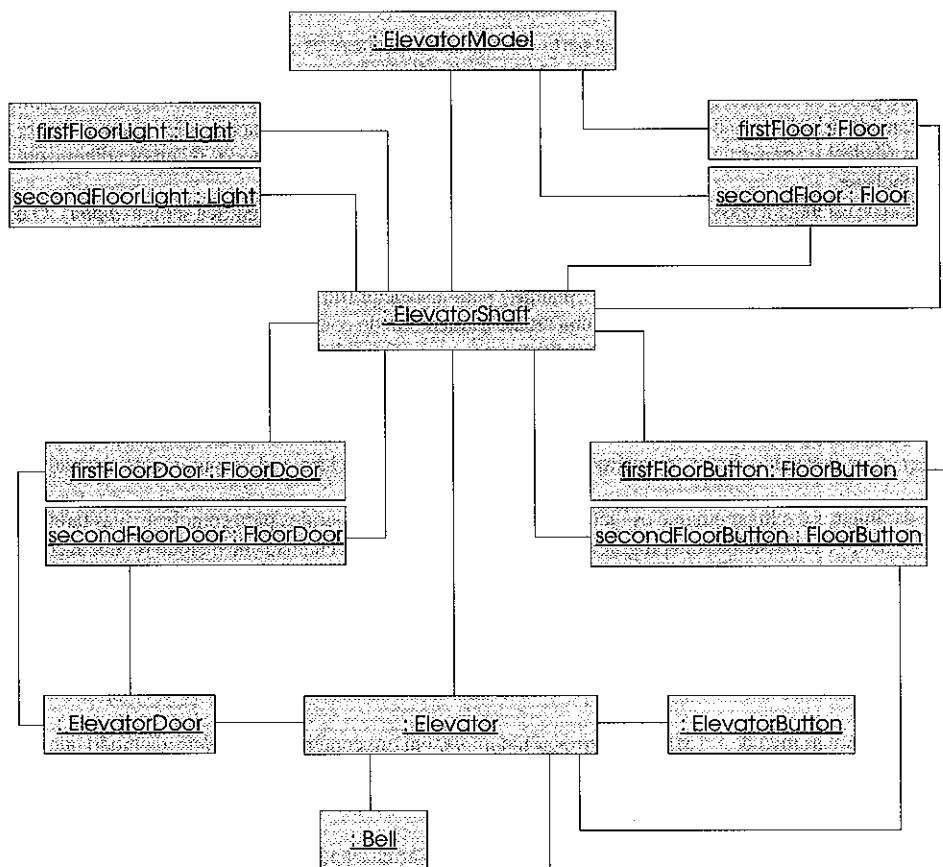


Fig. 3.24 Diagrama de objetos de um prédio vazio em nosso modelo de elevador.

Perguntas

1. Por que poderia ser mais complicado implementar um edifício de três andares (ou mais alto)?
2. É comum em grandes edifícios ter muitos elevadores. Veremos no Capítulo 9 que, uma vez que tenhamos criado um objeto elevador, é fácil criar tantos quanto queiramos. Quais são os problemas ou oportunidades que você prevê ao ter vários elevadores? Cada um deles pode apanhar e deixar os passageiros em todos os andares de um grande edifício?
3. Para simplificar, demos ao nosso elevador e a cada andar a capacidade de um passageiro. Quais são os problemas e as oportunidades que você prevê ao aumentar estas capacidades?

Resumo

- *Applets* são programas Java que podem ser embutidos em documentos *Hypertext Markup Language* (HTML – i.e., páginas da Web). Quando um navegador carrega uma página da Web que contém um *applet*, o *applet* é baixado para o navegador e começa a ser executado.
- No **appletviewer**, você pode executar um *applet* novamente clicando no menu **Applet** do **appletviewer** e selecionando a opção **Reload** do menu. Para terminar um *applet*, clique no menu **Applet** do **appletviewer** e selecione a opção **Quit**.
- A classe **Graphics** está localizada no pacote **java.awt**. Importe a classe **Graphics** de modo que o programa possa desenhar imagens gráficas.
- A classe **JApplet** está localizada no pacote **javax.swing**. Quando você cria um *applet* em Java, você normalmente importa a classe **JApplet**.
- Java utiliza herança para criar novas classes a partir de definições de classe existentes. A palavra-chave **extends**, seguida por um nome de classe, indica a classe da qual uma nova classe herda.
- No relacionamento de herança, a classe que se segue a **extends** é chamada de superclasse ou classe básica, e a nova classe é chamada de subclasse ou classe derivada. O uso de herança resulta em uma nova definição de classe que tem os atributos (dados) e comportamentos (métodos) da superclasse, assim como os novos recursos adicionados na definição de subclasse.
- Um benefício de estender a classe **JApplet** é que outra pessoa já definiu “o que significa ser um *applet*”. O **appletviewer** e os navegadores da World Wide Web que suportam *applets* esperam que cada *applet* Java tenha certas capacidades (atributos e comportamentos), e a classe **JApplet** já fornece todas essas capacidades.
- As classes são utilizadas como “modelos” ou “plantas” para instanciar (ou criar) objetos na memória para utilização em um programa. Um objeto (ou instância) é uma região na memória do computador em que as informações são armazenadas para utilização pelo programa. O termo objeto normalmente implica que atributos (dados) e comportamentos (métodos) estão associados com o objeto e que esses comportamentos realizam operações sobre os atributos do objeto.
- O método **paint** é um dos três métodos (comportamentos) que um contêiner de *applets* chama quando qualquer *applet* começa a ser executado. Esses três métodos são **init**, **start** e **paint** e são seguramente chamados nessa ordem.
- A lista de parâmetros é o lugar em que os métodos recebem os dados requeridos para completar suas tarefas. Normalmente, esses dados são passados pelo programador para o método por uma chamada de método (também conhecida como invocar um método). No caso do método **paint**, o contêiner de *applets* chama o método e passa a ele o argumento **Graphics**.
- O método **drawString** da classe **Graphics** desenha um *string* na localização especificada no *applet*. O primeiro argumento para **drawString** é o *String* a desenhar. Os dois últimos argumentos na lista são as coordenadas (ou posição) em que o *string* deve ser desenhado. As coordenadas são medidas em relação ao canto superior esquerdo do *applet* em pixels.
- Você deve criar um arquivo de HTML para carregar um *applet* em um contêiner de *applets*, de modo que o contêiner de *applets* possa executar o *applet*.
- Muitos códigos de HTML (conhecidos como *marcas*) vêm em pares. As marcas de HTML iniciam com um sinal de menor que < e terminam com um sinal de maior que >.
- Normalmente, o *applet* e seu arquivo HTML correspondente são armazenados no mesmo diretório no disco.
- O primeiro componente da marca <**applet**> indica o arquivo que contém a classe compilada do *applet*. O segundo e o terceiro componentes da marca <**applet**> indicam a largura (**width**) e a altura (**height**) do *applet* em pixels. Geralmente, cada *applet* deve ter menos de 800 pixels de largura e 640 pixels de altura.

- O **appletviewer** só entende as marcas de HTML `<applet>` e `</applet>`; por isso, ele é às vezes conhecido como “navegador mínimo”. Ele ignora todas as outras marcas de HTML.
- O método **drawLine** da classe **Graphics** desenha linhas. O método exige quatro argumentos que representam as duas extremidades da linha no *applet* – a coordenada *x* e coordenada *y* da primeira extremidade da linha e a coordenada *x* e a coordenada *y* da segunda extremidade da linha. Todos os valores de coordenadas são especificados com relação à coordenada do canto esquerdo superior (0, 0) do *applet*.
- O tipo de dados primitivos **double** armazena os números de ponto flutuante com precisão dupla. O tipo de dados primitivos **float** armazena números de ponto flutuante com precisão simples. Um **double** exige mais memória para armazenar um valor de ponto flutuante, mas o armazena com aproximadamente duas vezes a precisão de um **float** (15 dígitos significativos para **double** versus sete dígitos significativos para **float**).
- As instruções **import** não são necessárias se você sempre utilizar o nome completo de uma classe, incluindo o nome completo do pacote e o nome completo da classe (p. ex., `java.awt.Graphics`).
- A notação de asterisco (*) depois de um nome de pacote em um **import** indica que todas as classes no pacote devem estar disponíveis para o compilador a fim de que o compilador possa assegurar que as classes sejam utilizadas corretamente. Isso permite utilizar o nome abreviado (o nome de classe sozinho) de qualquer classe do pacote no programa.
- Cada instância (objeto) de uma classe contém uma cópia de cada variável de instância daquela classe. As variáveis de instância são declaradas no corpo da definição de classe, mas não no corpo de qualquer método daquela definição de classe. Um benefício importante de variáveis de instância é que seus identificadores podem ser utilizados em todos os métodos da classe.
- As variáveis definidas no corpo de um método são conhecidas como variáveis locais e só podem ser utilizadas no corpo do método em que estão definidas.
- Sempre é atribuído um valor-padrão às variáveis de instância, mas não às variáveis locais.
- O método **init** normalmente inicializa as variáveis de instância do *applet* (se elas precisam ser inicializadas com um valor diferente do seu valor *default*) e executa quaisquer tarefas que devam ocorrer somente uma vez, quando a execução do *applet* inicia.
- Há na realidade dois tipos de variáveis em Java – variáveis de tipos de dados primitivos e referências.
- As referências referem-se a objetos em um programa. As referências, na verdade, contêm o endereço de um objeto na memória do computador. Utiliza-se uma referência para enviar mensagens (isto é, chamar métodos) para o objeto na memória. Como parte da mensagem (chamada do método), fornecemos os dados (argumentos) que o método exige para fazer sua tarefa.
- Uma variável é semelhante a um objeto. A diferença básica entre uma variável e um objeto é que o objeto é definido por uma definição de classe que pode conter dados (variáveis de instância) e métodos, ao passo que a variável é definida por um tipo de dados primitivo (ou predefinido) (um **char**, **byte**, **short**, **int**, **long**, **float**, **double** ou **boolean**), que pode conter apenas dados.
- A variável pode armazenar exatamente um valor por vez, ao passo que um objeto pode conter muitos membros individuais de dados.
- Se o tipo de dado usado para declarar uma variável for um nome de classe, o identificador é uma referência a um objeto e essa referência pode ser utilizada para enviar mensagens (chamar métodos) para esse objeto. Se o tipo de dados usado para declarar uma variável é um dos tipos de dados primitivos, o identificador é uma variável que pode ser utilizada para armazenar na memória ou recuperar da memória um único valor do tipo primitivo declarado.
- O método **Double.parseDouble** (um método **static** da classe **Double**) converte seu argumento de **String** em um valor de ponto flutuante **double**. A classe **Double** faz parte do pacote `java.lang`.
- O método **drawRect** desenha um retângulo com base em seus quatro argumentos. Os dois primeiros valores inteiros representam a coordenada *x* superior esquerda e a coordenada *y* superior esquerda em que o objeto **Graphics** inicia o desenho do retângulo. O terceiro e o quarto argumentos são inteiros não-negativos que representam a largura do retângulo em pixels e a altura do retângulo em pixels, respectivamente.
- Para usar os recursos de Java 2 em um *applet*, a Sun fornece o Java Plug-in para contornar o suporte a Java de um navegador e usar uma versão completa do Java 2 Runtime Environment (J2RE) que é instalada no computador local do usuário.
- Para especificar que um *applet* deve usar o Java Plug-in em vez do suporte a Java do navegador, use o HTML Converter para converter as marcas `<applet>` e `</applet>` do *applet* no arquivo HTML, a fim de indicar que o contêiner de *applets* deve usar o Plug-in para executar o *applet*. A Sun fornece o Java Plug-in 1.3 HTML Converter para fazer a conversão para você.

Terminologia

<i>altura (height) de um applet</i>	<i>método drawRect da classe Graphics</i>
<i>applet</i>	<i>método drawString da classe Graphics</i>
<i>appletviewer</i>	<i>método init da classe JApplet</i>
<i>argumento de linha de comando</i>	<i>método paint da classe JApplet</i>
<i>arquivo de texto</i>	<i>método start da classe JApplet</i>
<i>chamada de método</i>	<i>Microsoft Internet Explorer</i>
<i>classe derivada</i>	<i>navegador</i>
<i>classe Graphics</i>	<i>Netscape Communicator</i>
<i>classe JApplet</i>	<i>número de ponto flutuante</i>
<i>código-fonte</i>	<i>número de ponto flutuante com precisão dupla</i>
<i>contêiner de applets</i>	<i>número de ponto flutuante com precisão simples</i>
<i>coordenada</i>	<i>objeto</i>
<i>criar um objeto</i>	<i>ocultamento de informações</i>
<i>erro de lógica</i>	<i>pacote java.awt</i>
<i>HTML Converter</i>	<i>pacote javax.swing</i>
<i>Hypertext Markup Language (HTML)</i>	<i>palavra-chave extends</i>
<i>instanciar um objeto</i>	<i>pixel</i>
<i>instrução de importação</i>	<i>referências</i>
<i>interface</i>	<i>subclasse</i>
<i>invocar um método</i>	<i>superclasse</i>
<i>item de menu Quit</i>	<i>tipo de dados predefinido</i>
<i>item de menu Reload</i>	<i>tipo de dados primitivo</i>
<i>Java 2 Runtime Environment (J2RE)</i>	<i>tipo de dados primitivo double</i>
<i>Java Plug-in</i>	<i>tipo primitivo boolean</i>
<i>largura (width) de um applet</i>	<i>tipo primitivo byte</i>
<i>lista de parâmetros</i>	<i>tipo primitivo char</i>
<i>marca <applet></i>	<i>tipo primitivo float</i>
<i>marca de HTML</i>	<i>tipo primitivo int</i>
<i>mensagem</i>	<i>tipo primitivo long</i>
<i>menu Applet</i>	<i>tipo primitivo short</i>
<i>método Double.parseDouble</i>	<i>variável de instância</i>
<i>método drawLine da classe Graphics</i>	<i>variável local</i>
	<i>World Wide Web</i>

Exercícios de auto-revisão

- 3.1** Preencha as lacunas em cada uma das frases seguintes:
- A classe _____ fornece métodos para desenhar.
 - Os *applets* Java começam a execução com uma série de três chamadas de método: _____, _____ e _____.
 - Os métodos _____ e _____ exibem linhas e retângulos.
 - A palavra-chave _____ indica que uma nova classe é uma subclasse de uma classe existente.
 - Cada *applet* Java deve estender uma classe _____ e uma classe _____.
 - Os oito tipos de dados primitivos de Java são: _____, _____, _____, _____, _____, _____, _____ e _____.
- 3.2** Determine se cada uma das frases seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- Para desenhar um retângulo, o método **drawRect** exige quatro argumentos que especificam dois pontos no *applet* para desenhar um retângulo.
 - O método **drawLine** exige quatro argumentos que especificam dois pontos no *applet* para desenhar uma linha.
 - O tipo **Double** é um tipo de dados primitivo.
 - O tipo de dados **int** é utilizado para declarar um número de ponto flutuante.
 - O método **Double.parseDouble** converte um **String** em um valor **double** primitivo.
- 3.3** Escreva instruções Java para realizar cada uma das instruções seguintes:
- Exibir um diálogo pedindo para o usuário inserir um número de ponto flutuante.

- b) Converter um **String** em um número de ponto flutuante e armazenar o valor convertido na variável **age** do tipo **double**. Pressuponha que o **String** seja armazenado em **stringValue**.
- c) Desenhar a mensagem "**This is a Java program**" em uma linha em um **applet** (pressuponha que você esteja definindo essa instrução no método **paint** do **applet**) na posição (10, 10).
- d) Desenhar a mensagem "**This is a Java program**" em duas linhas em um **applet** (pressuponha que essas instruções sejam definidas no método **paint** do **applet**) iniciando na posição (10, 10) e onde termina a primeira linha com **Java**. Faça as duas linhas iniciarem na mesma coordenada **x**.

Respostas aos exercícios de auto-revisão

- 3.1** a) **Graphics**. b) **init**, **start** e **paint**. c) **drawLine** e **drawRect**. d) **extends**. e) **JApplet**, **Applet**. f) **byte**, **short**, **int**, **long**, **float**, **double**, **char** e **boolean**.
- 3.2** a) Falsa. O método **drawRect** exige quatro argumentos – dois que especificam o canto superior esquerdo do retângulo e dois que especificam a largura e a altura do retângulo. b) Verdadeira. c) Falsa. O tipo **Double** é uma classe no pacote **java.lang**. Lembre-se de que os nomes que começam com uma letra maiúscula são normalmente nomes de classe. d) Falsa. O tipo de dados **double** ou o tipo de dados **float** podem ser utilizados para declarar um número de ponto flutuante. O tipo de dados **int** é utilizado para declarar inteiros. e) Verdadeira.
- 3.3** a) `stringValue = JOptionPane.showInputDialog(`
`"Enter a floating-point number");`
 b) `age = Double.parseDouble(stringValue);`
 c) `g.drawString("This is a Java program", 10, 10);`
 d) `g.drawString("This is a Java", 10, 10);`
`g.drawString("program", 10, 25);`

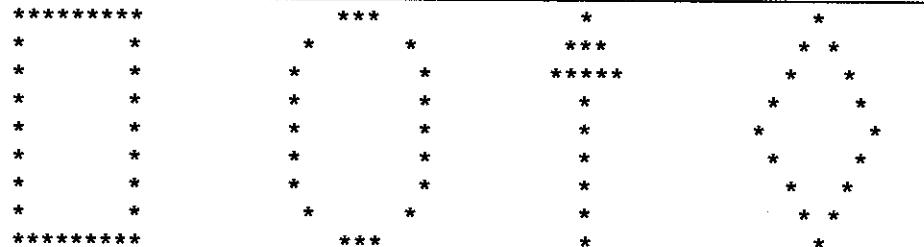
Exercícios

- 3.4** Preencha as lacunas em cada uma das frases seguintes:
- a) O tipo de dados _____ declara uma variável de ponto flutuante de precisão simples.
 - b) Se a classe **Double** fornece o método **parseDouble** para converter um **String** em um **double** e a classe **Integer** fornece o método **parseInt** para converter um **String** em um **int**, então a classe **Float** provavelmente fornece o método _____ para converter um **String** em um **float**.
 - c) O tipo de dados _____ declara uma variável de ponto flutuante com precisão dupla.
 - d) O _____ ou um navegador pode ser utilizado para executar um **applet** Java.
 - e) Para carregar um **applet** em um navegador você deve primeiro definir um arquivo _____.
 - f) As marcas de HTML _____ e _____ especificam que um **applet** deve ser carregado em um container de **applets** e executado.
- 3.5** Determine se cada uma das frases seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- a) Todos os navegadores suportam Java 2.
 - b) Ao se utilizar um **import** na forma **javax.swing.***, todas as classes no pacote são importadas.
 - c) Você não precisa de instruções de importação se o nome completo do pacote e o nome completo da classe forem especificados toda vez que você fizer referência a uma classe em um programa.
- 3.6** Escreva um **applet** que pede ao usuário para digitar dois números de ponto flutuante, que lê os dois números do usuário e desenha a soma, o produto (multiplicação), a diferença e o quociente (divisão) dos dois números. Utilize as técnicas mostradas na Fig. 3.12.
- 3.7** Escreva um **applet** que pede ao usuário para digitar dois números de ponto flutuante, lê os números do usuário e exibe o maior número seguido pelas palavras "**is larger**" como um **string** no **applet**. Se os números forem iguais, deve imprimir a mensagem "**These numbers are equal**". Utilize as técnicas mostradas na Fig. 3.12.
- 3.8** Escreva um **applet** que lê três números de ponto flutuante do usuário e exibe a soma, a média, o produto, o maior e o menor desses números como **strings** no **applet**. Utilize as técnicas mostradas na Fig. 3.12.
- 3.9** Escreva um **applet** que obtém uma entrada do usuário definindo o raio de um círculo como um número de ponto flutuante e desenha o diâmetro, a circunferência e a área do círculo. Utilize o valor 3,14159 para π . Utilize as técnicas mostradas na Fig. 3.12. [Nota: você também pode utilizar a constante predefinida **Math.PI** para o valor de π . Essa cons-

tante é mais precisa que o valor 3,14159. A classe **Math** é definida no pacote **java.lang**, assim você não precisa importá-la.] Utilize as seguintes fórmulas (r é o raio):

$$\begin{aligned} \text{diâmetro} &= 2r \\ \text{circunferência} &= 2\pi r \\ \text{área} &= \pi r^2 \end{aligned}$$

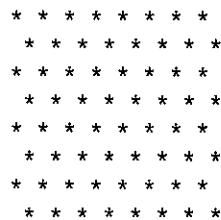
- 3.10** Escreva um *applet* que desenha uma caixa, uma oval, uma seta e um losango utilizando asteriscos (*), como segue:



- 3.11** Escreva um *applet* que lê cinco inteiros e determina e imprime os inteiros maiores e menores no grupo. Utilize apenas as técnicas de programação que você aprendeu neste capítulo e no Capítulo 2. Desenhe os resultados no *applet*.

- 3.12** Escreva um *applet* que lê dois números de ponto flutuante e determina e imprime se o primeiro é um múltiplo do segundo. (Dica: utilize o operador de módulo.) Utilize apenas as técnicas de programação que você aprendeu neste capítulo e no Capítulo 2. Desenhe os resultados no *applet*.

- 3.13** Escreva um *applet* que desenha um padrão de tabuleiro de damas, como segue:



- 3.14** Escreva um *applet* que desenha uma variedade de retângulos de tamanhos diferentes e em posições diferentes.

- 3.15** Escreva um *applet* que permite digitar os quatro argumentos exigidos pelo método **drawRect**, depois desenhe um retângulo utilizando os quatro valores de entrada.

- 3.16** A classe **Graphics** contém um método **drawOval** que obtém exatamente os mesmos quatro argumentos que o método **drawRect**. Entretanto, os argumentos para o método **drawOval** especificam o “retângulo delimitador” para a oval. Os lados do retângulo delimitador são os limites da oval. Escreva um *applet* Java que desenha uma oval e um retângulo com os mesmos quatro argumentos. Você verá que a oval tangencia o retângulo no centro de cada lado.

- 3.17** Modifique a solução do Exercício 3.16 para dar saída a uma variedade de ovais de formas e tamanhos diferentes.

- 3.18** Escreva um *applet* que permite digitar os quatro argumentos exigidos pelo método **drawOval**, depois desenhe uma oval utilizando os quatro valores de entrada.

3.19 O que o seguinte código imprime?

```
g.drawString( "*", 25, 25 );
g.drawString( "****", 25, 55 );
g.drawString( "*****", 25, 85 );
g.drawString( "*****", 25, 70 );
g.drawString( "***", 25, 40 );
```

3.20 Utilizando apenas as técnicas de programação dos Capítulos 2 e 3, escreva um *applet* que calcula os quadrados e cubos dos números de 0 a 10 e desenha os valores resultantes no formato de tabela como segue:

número	quadrado	cubo
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

[Nota: esse programa não exige nenhuma entrada do usuário.]

4

Estruturas de controle: parte 1

Objetivos

- Entender técnicas básicas de solução de problemas.
- Ser capaz de desenvolver algoritmos pelo processo de refinamento passo a passo de cima para baixo.
- Ser capaz de utilizar as estruturas de seleção `if` e `if/else` para escolher entre ações alternativas.
- Ser capaz de utilizar a estrutura de repetição `while` para executar repetidamente instruções em um programa.
- Entender repetição controlada por contador e repetição controlada por sentinela.
- Ser capaz de utilizar operadores de incremento, decremento e atribuição.

Vamos todos dar um passo para a frente.

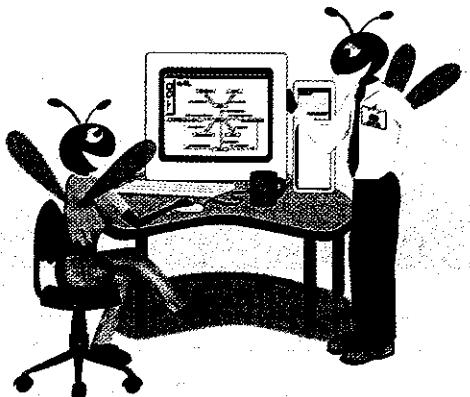
Lewis Carroll

A roda já deu uma volta completa.

William Shakespeare, *Rei Lear*

Quantas maçãs não caíram na cabeça de Newton antes de ele ter percebido a dica!

Robert Frost, comentário



Sumário do capítulo	
4.1	Introdução
4.2	Algoritmos
4.3	Programando
4.4	Estruturas de controle
4.5	A estrutura de seleção if
4.6	A estrutura de seleção if-else
4.7	A estrutura de repetição while
4.8	Formulando algoritmos: estrutura de caso (seleção condicional de contagem)
4.9	Formulando algoritmos: estrutura de loop (controle de cima para baixo: estrutura de caso) (seleção condicional por semântica)
4.10	Formulando algoritmos: estrutura de loop (controle aninhadas)
4.11	Operadores de atribuição
4.12	Operadores de incremento e decremento
4.13	Tipos de dados mutáveis
4.14	(Estudo de caso optional) Personalizar objetos, identificando os diferentes classes

4.1 Introdução

Antes de escrever um programa para resolver um problema, é essencial ter um entendimento completo do problema e uma abordagem cuidadosamente planejada para resolvê-lo. Ao escrever um programa, é igualmente essencial entender os tipos de blocos de construção que estão disponíveis e empregar princípios comprovados de construção de programa. Neste capítulo e no Capítulo 5, discutimos essas questões em nossa apresentação da teoria e dos princípios da programação estruturada. As técnicas a serem aprendidas aqui são aplicáveis à maioria das linguagens de alto nível, incluindo Java. Quando estudarmos a programação baseada em objetos mais profundamente no Capítulo 8, veremos que as estruturas de controle são úteis na construção e na manipulação de objetos.

4.2 Algoritmos

Qualquer problema de computação pode ser resolvido executando-se uma série de ações em uma ordem específica. O *procedimento* para resolver um problema em termos:

1. das ações a serem executadas e
2. da ordem em que essas ações devem ser executadas

chama-se de *algoritmo*. O exemplo a seguir demonstra que é importante especificar corretamente a ordem em que as ações serão executadas.

Considere o “algoritmo levante e brilhe”, utilizado por um executivo novato para sair da cama e ir trabalhar: (1) levantar da cama, (2) tirar o pijama, (3) tomar banho, (4) vestir-se, (5) tomar o café da manhã, (6) pegar carona até o trabalho.

Essa rotina leva o executivo a trabalhar bem preparado para tomar decisões críticas. Suponha, entretanto, que os mesmos passos sejam realizados em uma ordem um pouquinho diferente: (1) levantar da cama, (2) tirar o pijama, (3) vestir-se, (4) tomar banho, (5) tomar o café da manhã, (6) pegar carona até o trabalho.

Nesse caso, nosso executivo aparece no trabalho molhado. Especificar a ordem em que as instruções devem ser executadas em um programa de computador chama-se *controle de programa*. Neste capítulo e no Capítulo 5, investigamos os recursos para controle de programa em Java.

4.3 Pseudocódigo

O *pseudocódigo* é uma linguagem artificial informal que ajuda os programadores a desenvolver algoritmos. O pseudocódigo que apresentamos aqui é particularmente útil para desenvolver algoritmos que serão convertidos em partes estruturadas de programas Java. O pseudocódigo é similar à língua cotidiana; é conveniente e amigável ao usuário, embora não seja uma linguagem real de programação de computador.

Os programas em pseudocódigo não são realmente executados em computadores. Mais exatamente, eles ajudam o programador a “estudar” um programa antes de tentar escrevê-lo em uma linguagem de programação como Java. Neste capítulo, fornecemos vários exemplos de programas em pseudocódigo.



Observação de engenharia de software 4.1

O pseudocódigo é freqüentemente utilizado para “estudar” um programa durante o processo de projeto do programa. Assim, o programa em pseudocódigo é convertido para Java.

O estilo de pseudocódigo que apresentamos consiste puramente em caracteres, portanto os programadores podem digitar convenientemente programas em pseudocódigo utilizando um programa editor de texto. O computador pode produzir uma cópia impressa de um programa em pseudocódigo conforme necessário. O programa em pseudocódigo cuidadosamente preparado pode ser facilmente convertido em um programa Java correspondente. Em muitos casos, esta conversão é feita simplesmente substituindo-se as instruções em pseudocódigo por seus equivalentes Java.

Normalmente, o pseudocódigo descreve apenas as instruções executáveis – as ações que são realizadas quando o programa é convertido de pseudocódigo para Java e é executado. As declarações não são instruções executáveis. Por exemplo, a declaração

```
int i;
```

diz ao compilador o tipo da variável `i` e instrui o compilador a reservar espaço na memória para a variável. Essa declaração não faz com que qualquer ação – como entrada, saída ou cálculo – ocorra quando o programa é executado. Alguns programadores preferem listar variáveis e mencionar o propósito de cada uma no começo de um programa em pseudocódigo.

4.4 Estruturas de controle

Normalmente, as instruções em um programa são executadas uma após a outra na ordem em que são escritas. Isso é chamado de *execução seqüencial*. Várias instruções de Java que discutiremos a seguir permitem que o programador especifique que a próxima instrução a ser executada pode ser outra que não a próxima na seqüência. Isso se chama *transferência de controle*.

Durante a década de 1960, tornou-se claro que a utilização indiscriminada de transferências de controle era a raiz de muitas dificuldades experimentadas por grupos de desenvolvimento de *software*. O dedo acusador foi apontado para a *instrução goto* (usada em diversas linguagens de programação, inclusive C e Basic) que permite especificar uma transferência de controle para uma ampla variedade de possíveis destinos em um programa. A noção da chamada *programação estruturada* tornou-se quase sinônimo de “*eliminação do goto*”. Java não tem uma instrução `goto`; entretanto, `goto` é uma palavra reservada e não deve ser usada em um programa Java.

A pesquisa de Bohm e Jacopini¹ tinha demonstrado que os programas poderiam ser escritos sem nenhuma instrução `goto`. O desafio da era para os programadores foi mudar seus estilos para “*programação sem goto*”. Mas

¹ 1. Bohm, C., e G. Jacopini, “Flow Diagrams, Turing Machines, and Languages With Only Two Formation Rules.” *Communications of the ACM*, vol. 9, No. 5, May 1966, pp. 336-371.

foi só na década de 1970 que os programadores começaram a tratar programação estruturada seriamente. Os resultados foram impressionantes quando os grupos de desenvolvimento de *software* informaram ter reduzido o tempo de desenvolvimento, que a entrega de sistemas ocorria com mais freqüência dentro do prazo e que a conclusão dos projetos de *software* ocorria com mais freqüência dentro do orçamento. A chave para esses sucessos é que programas estruturados são mais claros, mais fáceis de depurar e de modificar e menos propensos a conter falhas.

O trabalho de Bohm e Jacopini demonstrou que todos os programas poderiam ser escritos em termos de somente três *estruturas de controle*, a saber a *estrutura de seqüência*, a *estrutura de seleção* e a *estrutura de repetição*. A estrutura de seqüência faz parte de Java. A menos que instruído de outro modo, o computador executa instruções Java uma após a outra na ordem em que estão escritas. O segmento de *fluxograma* da Fig. 4.1 ilustra uma estrutura de seqüência típica em que dois cálculos são realizados na ordem indicada.

O fluxograma é uma representação gráfica de um algoritmo ou de uma parte de um algoritmo. Os fluxogramas são desenhados com certos símbolos de uso especial como retângulos, losangos, elipses e pequenos círculos; esses símbolos são conectados por setas denominadas *linhas de fluxo*, que indicam a ordem em que as ações do algoritmo são executadas.

Como o pseudocódigo, os fluxogramas freqüentemente são úteis para desenvolver e representar algoritmos, embora o pseudocódigo seja preferido por muitos programadores. Os fluxogramas mostram claramente como as estruturas de controle operam; e é somente para isso que são usados neste texto. O leitor deve comparar cuidadosamente as representações em pseudocódigo e em fluxograma de cada estrutura de controle.

Considere o segmento de fluxograma para a estrutura de seqüência na Fig. 4.1. Utilizamos o *retângulo*, também chamado de *símbolo de ação*, para indicar qualquer tipo de ação, incluindo um cálculo ou uma operação de entrada/saída. As linhas de fluxo na figura indicam a ordem em que as ações devem ser executadas – primeiro, **grade** será adicionado a **total**, depois **1** será adicionado a **counter**. Java permite ter quantas ações quisermos em uma estrutura de seqüência. Como veremos em seguida, podemos colocar várias ações em seqüência em qualquer lugar onde uma única ação possa ser colocada.

Ao se desenhar um fluxograma que representa um algoritmo *completo*, a *elipse* que contém a palavra “*Ínicio*” é o primeiro símbolo utilizado no fluxograma; a elipse que contém a palavra “*Fim*” indica onde o algoritmo termina. Ao se desenhar somente uma parte de um algoritmo, como na Fig. 4.1, as elipses são omitidas em favor da utilização de *pequenos círculos*, também chamados de *símbolos de conexão*.

Talvez o símbolo de fluxograma mais importante seja o *losango*, também chamado de *símbolo de decisão*, que indica que uma decisão deve ser tomada. Discutiremos o símbolo de decisão na próxima seção.

Java fornece três tipos de estruturas de seleção; discutimos cada um desses tipos neste capítulo e no Capítulo 5. A estrutura de seleção **if** executa (seleciona) uma ação se uma condição for verdadeira ou pula a ação se for falsa. A estrutura de seleção **if/else** executa uma ação se uma condição for verdadeira e executa uma ação diferente se for falsa. A estrutura de seleção **switch** (Capítulo 5) executa apenas uma de muitas ações diferentes, dependendo do valor de uma expressão.

A estrutura **if** é chamada de *estrutura de seleção única*, porque ela seleciona ou ignora uma única ação (ou, como veremos a seguir, um único grupo de ações). A estrutura **if/else** chama-se *estrutura de seleção dupla* porque seleciona entre duas ações (ou grupos de ações) diferentes. A estrutura **switch** chama-se *estrutura de seleção múltipla* uma vez que seleciona entre muitas ações (ou grupos de ações) diferentes.

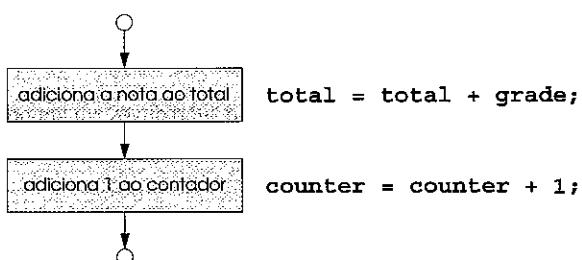


Fig. 4.1 Fluxograma da estrutura de seqüência de Java.

Java fornece três tipos de estruturas de repetição, a saber, **while**, **do/while** e **for** (**do/while** e **for** são abordadas no Capítulo 5). Cada uma das palavras **if**, **else**, **switch**, **while**, **do** e **for** são *palavras-chave* de Java. Essas palavras são reservadas pela linguagem para implementar vários recursos, como as estruturas de controle de Java. As palavras-chave não podem ser utilizadas como identificadores, como nomes variáveis. Uma lista completa de palavras-chave Java é mostrada na Fig. 4.2.



Erro comum de programação 4.1

Utilizar uma palavra-chave como identificador é um erro de sintaxe.

Bem, isso é tudo. Java tem apenas sete estruturas de controle: a estrutura de seqüência, três tipos de estruturas de seleção e três tipos de estruturas de repetição. Cada programa é formado combinando-se tantos tipos de estruturas de controle quanto forem apropriados para o algoritmo que o programa implementa. Assim como ocorre com a estrutura de seqüência da Fig. 4.1, veremos que cada estrutura de controle pode ser representada por um fluxograma com dois símbolos de conexão, um no ponto de entrada para a estrutura de controle e um no ponto de saída.

As estruturas de controle de entrada única/saída única facilitam a construção dos programas – as estruturas de controle são ligadas umas às outras conectando o ponto de saída de uma estrutura de controle com o ponto de entrada da próxima. Este procedimento é semelhante à maneira como uma criança empilha os blocos de construção; portanto, chamamo-lo de *empilhamento de estruturas de controle*. Aprenderemos que há apenas uma outra maneira através da qual as estruturas de controle podem ser conectadas – o *aninhamento das estruturas de controle*. Portanto, algoritmos em programas Java são construídos a partir de apenas sete tipos de estruturas de controle diferentes, combinadas apenas de duas maneiras.

Palavras-chave de Java

abstract	boolean	break	byte	case
catch	char	class	continue	default
do	double	else	extends	false
final	finally	float	for	if
implements	import	instanceof	int	interface
long	native	new	null	package
private	protected	public	return	short
static	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		
<i>Palavras-chave que são reservadas mas não utilizadas por Java</i>				
const	goto			

Fig. 4.2 Palavras-chave de Java.

4.5 A estrutura de seleção if

Utiliza-se a estrutura de seleção para escolher entre cursos de ação alternativos em um programa. Por exemplo, suponha que a nota de aprovação em um exame seja 60 (em 100). Então, a instrução em pseudocódigo

*Se (If) a nota do aluno for maior do que ou igual a 60
Imprimir "Aprovado"*

determina se a condição “nota do aluno for maior do que ou igual a 60” é verdadeira ou falsa. Se a condição for verdadeira, então “Aprovado” é impresso e a próxima instrução do pseudocódigo é “executada” (lembre-se de que o pseudocódigo não é uma linguagem de programação real). Se a condição for falsa, a instrução de impressão é igno-

rada e a próxima instrução do pseudocódigo na seqüência é executada. Observe que a segunda linha dessa estrutura de seleção está recuada. Esse recuo é opcional, mas é veementemente recomendado uma vez que enfatiza a estrutura inerente de programas estruturados. O compilador Java ignora caracteres de espaçamento como espaços em branco, tabulações e novas linhas, utilizados para recuo e espaçamento vertical. Os programadores inserem esses caracteres de espaçamento para melhorar a clareza do programa.



Boa prática de programação 4.1

Aplicar consistentemente convenções de recuo de maneira racional a todos os seus programas melhora a legibilidade do programa. Sugerimos uma tabulação de tamanho fixo de aproximadamente 1/4 de polegada (0,63 cm) ou três espaços por recuo.

A instrução em pseudocódigo *Se (If)* precedente pode ser escrita em Java como

```
if ( studentGrade >= 60 )
    System.out.println( "Aprovado" );
```

Observe que o código Java corresponde aproximadamente ao pseudocódigo. Esse atributo é uma propriedade do pseudocódigo que o torna uma ferramenta útil de desenvolvimento de programas. A instrução no corpo da estrutura **if** gera como saída o *string* de caracteres "**Aprovado**" na janela de comando.

O fluxograma da Fig. 4.3 ilustra a estrutura **if** de seleção única. Esse fluxograma contém o que talvez seja o símbolo mais importante de um fluxograma – o *losango*, também chamado de *símbolo de decisão*, que indica que uma decisão deve ser tomada. O símbolo de decisão contém uma expressão, como uma condição, que pode ser **true** (verdadeira) ou **false** (falsa). O símbolo de decisão tem duas linhas de fluxo que saem dele. Um indica a direção a ser tomada quando a expressão no símbolo for verdadeira; o outro indica a direção a ser tomada quando a expressão for falsa. Uma decisão pode ser tomada com base em qualquer expressão que seja avaliada como um valor do tipo **boolean** de Java (isto é, qualquer expressão que dê como resultado **true** ou **false**).

Observe que a estrutura **if** também é uma estrutura de entrada única/saída única. Logo aprenderemos que os fluxogramas para as estruturas de controle restantes também contêm (além de símbolos de conexão e linhas de fluxo) apenas símbolos de ação para indicar as ações a serem realizadas e símbolos de decisão para indicar decisões a serem tomadas. Essa característica indica o *modelo de programação de ação/decisão* que temos enfatizado ao longo deste capítulo.

Podemos conceber sete contêineres, cada um contendo somente estruturas de controle de um dos sete tipos. Essas estruturas de controle estão vazias. Nada está escrito nos retângulos nem nos losangos. A tarefa do programador é, portanto, montar um programa empregando quantas estruturas de controle de cada tipo forem necessárias para o algoritmo, combinando essas estruturas de controle de apenas duas maneiras possíveis (empilhando ou aninhando), e depois preencher as ações e decisões de uma maneira que seja apropriada para o algoritmo. Neste capítulo discutiremos as várias maneiras em que se podem escrever as ações e decisões.

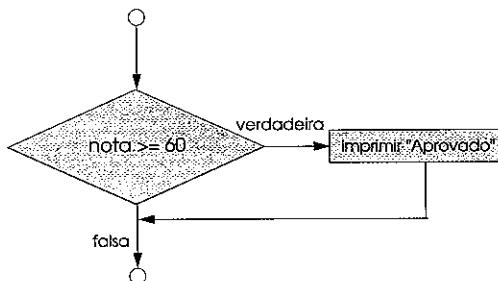


Fig. 4.3 Fluxograma da estrutura de seleção única **if**.

4.6 A estrutura de seleção `if/else`

A estrutura de seleção `if` executa uma ação indicada só quando a condição especificada é avaliada como `true`; caso contrário, a ação é pulada. A estrutura de seleção `if/else` permite especificar que, quando a condição for verdadeira, deve ser executada uma ação diferente da executada quando a condição for falsa. Por exemplo, a instrução em pseudocódigo

```
Se (If) a nota do aluno for maior do que ou igual a 60
    Imprimir "Aprovado"
senão (else)
    Imprimir "Reprovado"
```

imprime *Aprovado* se a nota do aluno for maior do que ou igual a 60 e imprime *Reprovado* se a nota do aluno for menor do que 60. Nos dois casos, depois que ocorre a impressão, a próxima instrução em pseudocódigo na seqüência é “executada”. Observe que o corpo do `else` também é recuado.



Boa prática de programação 4.2

Recue ambas as instruções do corpo de uma estrutura `if/else`.

A convenção de recuo que você escolhe deve ser cuidadosamente aplicada em todos os seus programas. É difícil ler programas que não utilizam convenções uniformes de espaçamento.

A estrutura `if/else` do pseudocódigo anterior pode ser escrita em Java como

```
if ( studentGrade >= 60 )
    System.out.println( "Aprovado" );
else
    System.out.println( "Reprovado" );
```

O fluxograma da Fig. 4.4 ilustra bem o fluxo de controle na estrutura `if/else`. Mais uma vez, observe que (além de pequenos círculos e setas) os únicos símbolos no fluxograma são retângulos (para ações) e um losango (para uma decisão). Continuamos a enfatizar esse modelo de computação de ação/decisão. Imagine novamente um container fundo contendo tantas estruturas de seleção duplas vazias quantas forem necessárias para construir um algoritmo Java. O trabalho do programador é montar as estruturas de seleção (empilhando e aninhando) com outras estruturas de controle exigidas pelo algoritmo e preencher os retângulos vazios e os losangos vazios com ações e decisões apropriadas para o algoritmo que está sendo implementado.

O operador condicional (`? :`) relaciona-se à estrutura `if/else`. O `? :` é o único operador ternário de Java – ele tem três operandos. Os operandos, junto com o `? :`, formam uma expressão condicional. O primeiro operando é uma expressão do tipo `boolean`, o segundo é o valor para a expressão condicional se a condição for avaliada como `true` e o terceiro é o valor para a expressão condicional se a condição for avaliada como `false`. Por exemplo, a instrução de saída

```
System.out.println( studentGrade >= 60 ? "Aprovado" : "Reprovado" );
```

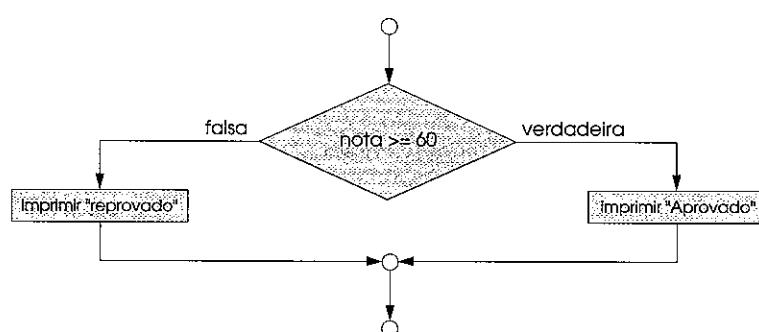


Fig. 4.4 Fluxograma da estrutura de seleção dupla `if/else`.

contém uma expressão condicional que é avaliada como o *string* "Aprovado" se a condição `studentGrade >= 60` for verdadeira, e é avaliada como o *string* "Reprovado" se a condição for falsa. Portanto, essa instrução com o operador condicional realiza essencialmente a mesma função que a instrução `if/else` mostrada anteriormente. A precedência do operador condicional é baixa, por isso a expressão condicional inteira normalmente é colocada entre parênteses. Veremos que os operadores condicionais podem ser utilizados em algumas situações em que as instruções `if/else` não podem.



Boa prática de programação 4.3

Em geral, as expressões condicionais são mais difíceis de ler do que as estruturas if/else. Tais expressões devem ser usadas com prudência quando elas ajudam a melhorar a legibilidade de um programa.

As estruturas aninhadas `if/else` testam múltiplos casos colocando estruturas `if/else` dentro de outras estruturas `if/else`. Por exemplo, a instrução em pseudocódigo a seguir imprimirá **A** para as notas maiores do que ou iguais a 90, **B** para as notas no intervalo de 80 a 89, **C** para as notas no intervalo de 70 a 79, **D** para as notas no intervalo de 60 a 69 e **F** para todas as outras notas:

Se a nota do aluno for maior do que ou igual a 90

Imprimir "A"

senão

Se a nota do aluno for maior do que ou igual a 80

Imprimir "B"

senão

Se a nota do aluno for maior do que ou igual a 70

Imprimir "C"

senão

Se a nota do aluno for maior do que ou igual a 60

Imprimir "D"

senão

Imprimir "F"

Esse pseudocódigo pode ser escrito em Java como

```
if ( studentGrade >= 90 )
    System.out.println( "A" );
else
    if ( studentGrade >= 80 )
        System.out.println( "B" );
    else
        if ( studentGrade >= 70 )
            System.out.println( "C" );
        else
            if ( studentGrade >= 60 )
                System.out.println( "D" );
            else
                System.out.println( "F" );
```

Se `studentGrade` for maior do que ou igual a 90, as primeiras quatro condições serão verdadeiras, mas somente a instrução `System.out.println` depois do primeiro teste será executada. Depois que aquela `System.out.println` particular for executada, a parte `else` da instrução `if/else` "externa" é pulada.



Boa prática de programação 4.4

Se existirem vários níveis de recuo, cada nível deve ser recuado pela mesma quantidade adicional de espaço.

A maioria dos programadores Java prefere escrever a estrutura `if` anterior como

```
if ( grade >= 90 )
    System.out.println( "A" );
```

```

else if ( grade >= 80 )
    System.out.println( "B" );
else if ( grade >= 70 )
    System.out.println( "C" );
else if ( grade >= 60 )
    System.out.println( "D" );
else
    System.out.println( "F" );

```

Ambas as formas são equivalentes. A última forma é popular porque evita o grande recuo do código para a direta. Esse recuo grande freqüentemente deixa pouco espaço em uma linha, forçando quebras de linha e diminuindo a legibilidade do programa.

É importante notar que o compilador Java sempre associa um `else` com o `if` anterior a ele, a menos que instruído a fazer de outro modo pela colocação de chaves (`{}`). Isso é conhecido como o *problema do else oscilante*. Por exemplo,

```

if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
    else
        System.out.println( "x is <= 5" );

```

parece indicar que, se `x` for maior do que 5, a estrutura `if` no seu corpo determina se `y` também é maior do que 5. Se for assim, o string "`x and y are > 5`" é enviado para a saída. Caso contrário, *parece* que, se `x` não for maior do que 5, a parte `else` da estrutura `if/else` envia para a saída o string "`x is <= 5`".

Cuidado! A estrutura aninhada `if` precedente não é executada como parece. O compilador na realidade interpreta a estrutura precedente como

```

if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
    else
        System.out.println( "x is <= 5" );

```

na qual o corpo da primeira estrutura `if` é uma estrutura `if/else`. Essa estrutura testa se `x` é maior que 5. Se for, a execução continua testando se `y` também é maior que 5. Se a segunda condição for verdadeira, o string adequado – "`x and y are > 5`" – é exibido. Entretanto, se a segunda condição for falsa, o string "`x is <= 5`" é exibido, embora saibamos que `x` é maior que 5.

Para forçar a estrutura aninhada `if` anterior a ser executada como originalmente pretendido, a estrutura deve ser escrita como segue:

```

if ( x > 5 ) {
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
}
else
    System.out.println( "x is <= 5" );

```

As chaves (`{}`) indicam ao compilador que a segunda estrutura `if` está no corpo da primeira estrutura `if` e que o `else` corresponde à primeira estrutura `if`. Nos Exercícios 4.21 e 4.22 você investigará ainda mais o problema do *else oscilante*.

A estrutura de seleção `if` normalmente espera encontrar apenas uma instrução no seu corpo. Para incluir várias instruções no corpo de um `if`, inclua as instruções entre chaves (`{ e }`). O conjunto de instruções contidas dentro de um par de chaves chama-se *bloco*.



Observação de engenharia de software 4.2

O bloco pode ser colocado em qualquer lugar de um programa em que se possa colocar uma instrução simples.

O exemplo a seguir inclui um bloco na parte `else` de uma estrutura `if/else`.

```

if (grade >= 60)
    System.out.println( "Aprovado" );

```

```

else {
    System.out.println( "Reprovado" );
    System.out.println( "Você deve repetir este curso." );
}

```

Nesse caso, se `grade` for menor que 60, o programa executa ambas as instruções no corpo do `else` e imprime
`Reprovado.`
`Você deve repetir este curso.`

Observe as chaves que cercam as duas instruções na cláusula `else`. Essas chaves são importantes. Sem as chaves, a instrução

```
System.out.println( "Você deve repetir este curso." );
```

estaria fora do corpo da parte `else` da estrutura `if` e seria executada independentemente de a nota ser ou não menor que 60.



Erro comum de programação 4.2

Esquecer uma ou ambas as chaves que delimitam um bloco pode levar a erros de sintaxe ou erros de lógica.

Os erros de sintaxe (como ocorre quando uma chave em uma instrução composta é omitida em um programa) são capturados pelo compilador. O *erro de lógica* (como ocorre quando ambas as chaves em uma instrução composta são omitidas no programa) produz seu efeito durante a execução. O *erro fatal de lógica* faz com que um programa apresente problemas e seja finalizado. O *erro não-fatal de lógica* permite que um programa continue sendo executado, mas o programa produz resultados incorretos.



Observação de engenharia de software 4.3

Assim como um bloco pode ser colocado em qualquer lugar onde uma instrução simples pode ser colocada, também é possível não ter absolutamente nenhuma instrução (isto é, uma instrução vazia em tais lugares). A instrução vazia é representada colocando-se um ponto-e-vírgula (;) onde normalmente haveria uma instrução.



Erro comum de programação 4.3

Colocar um ponto-e-vírgula depois da condição em uma estrutura if leva a um erro de lógica em estruturas if de uma única seleção e a um erro de sintaxe em estruturas if de seleção dupla (se o corpo da parte if contiver uma instrução não-vazia).



Boa prática de programação 4.5

Alguns programadores preferem digitar as chaves iniciais e finais de blocos antes de digitar as instruções individuais dentro das chaves. Essa prática ajuda a evitar a omissão de uma ou ambas as chaves.

Nesta seção, apresentamos a noção de bloco. O bloco pode conter declarações (como o corpo de `main`, por exemplo). As declarações em um bloco são comumente colocadas no início do bloco antes da ocorrência de qualquer instrução de ação, mas as declarações também podem ser intercaladas com instruções de ação.

4.7 A estrutura de repetição while

A estrutura de repetição permite especificar que uma ação deve ser repetida enquanto alguma condição permanecer verdadeira. A instrução em pseudocódigo

Enquanto (While) houver mais itens em minha lista de compras

Comprar o próximo item e riscá-lo de minha lista

descreve a repetição que ocorre durante um passeio de compras. A condição “enquanto houver mais itens em minha lista de compras” pode ser verdadeira ou falsa. Se ela for verdadeira, a ação “Comprar o próximo item e riscá-lo de minha lista” é executada. Essa ação será executada repetidamente enquanto a condição permanecer verdadeira. A(s) instrução(ões) contida(s) na estrutura de repetição *while* constitui(em) o corpo da estrutura *while*. O corpo da estrutura *while* pode ser uma instrução única ou um bloco. Em algum momento, a condição se tornará falsa (quando o úl-

timo item na lista de compras for comprado e riscado da lista). Neste ponto, a repetição termina e a primeira instrução em pseudocódigo depois da estrutura de repetição é executada.



Erro comum de programação 4.4

É um erro de lógica não colocar no corpo de uma estrutura `while` uma ação que em algum momento faz com que a condição em um `while` se torne falsa. Normalmente, uma estrutura de repetição assim nunca terminará – um erro chamado “laço (loop) infinito”.



Erro comum de programação 4.5

Escrever a palavra-chave `while` com uma letra maiúscula `W`, como em `While`, é um erro de sintaxe. (Lembre-se de que Java é uma linguagem que faz distinção entre maiúsculas e minúsculas). Todas as palavras-chave reservadas de Java, como `while`, `if` e `else`, contêm somente letras minúsculas.

Como exemplo de uma estrutura `while`, considere um segmento de programa projetado para encontrar a primeira potência de 2 maior que 1000. Suponha que a variável `int product` tenha sido inicializada com 2. Quando a estrutura `while` a seguir terminar a execução, `product` conterá o resultado:

```
int product = 2;

while ( product <= 1000 )
    product = 2 * product;
```

O fluxograma da Fig. 4.5 ilustra o fluxo de controle da estrutura de repetição `while` precedente. Mais uma vez, observe que, além de pequenos círculos e setas, o fluxograma contém somente um retângulo e um losango.

Imagine, novamente, um contêiner profundo com estruturas `while` vazias que podem ser empilhadas e aninhadas com outras estruturas de controle para formar uma implementação estruturada do fluxo de controle de um algoritmo. Os retângulos e losangos vazios são então preenchidos com ações e decisões apropriadas. O fluxograma mostra claramente a repetição. A linha de fluxo emergente do retângulo volta à decisão, que é testada toda vez pelo laço até que a condição finalmente se torne falsa. Nesse ponto, a estrutura `while` encerra e o controle passa para a próxima instrução no programa.

Quando o fluxo de controle entra na estrutura `while`, `product` é 2. A variável `product` é repetidamente multiplicada por 2, assumindo os valores 4, 8, 16, 32, 64, 128, 256, 512 e 1024 sucessivamente. Quando `product` se torna 1024, a condição `product <= 1000` na estrutura `while` torna-se `false`. Esse resultado termina a repetição com 1024 como valor final de `product`. A execução continua com a próxima instrução depois do `while`. [Nota: se uma condição de estrutura `while` for inicialmente `false`, a(s) instrução(ões) do corpo nunca será(ão) executada(s).]

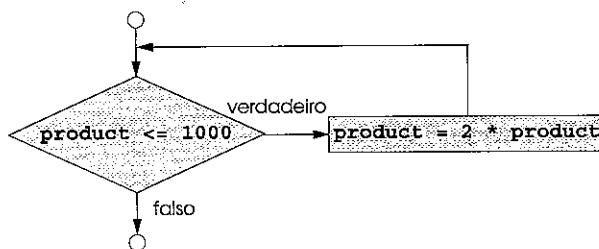
4.8 Formulando algoritmos: estudo de caso 1 (repetição controlada por contador)

Para ilustrar como os algoritmos são desenvolvidos, resolvemos diversas variações de um problema de cálculo da média da turma. Considere a seguinte definição de problema:

Uma turma de dez alunos se submeteu a um teste. As notas (inteiros no intervalo 0 a 100) para esse teste estão disponíveis. Determine a média da turma no teste.

A média da turma é igual à soma das notas dividida pelo número de alunos. O algoritmo para resolver esse problema em um computador deve ler cada uma das notas, realizar o cálculo da média e imprimir o resultado.

Vamos utilizar um pseudocódigo para listar as ações a ser executadas e especificar a ordem em que essas ações devem ser executadas. Utilizamos *repetição controlada por contador* para ler as notas uma por vez. Essa técnica utiliza uma variável chamada `contador` para controlar o número de vezes que um conjunto de instruções será executado. Neste exemplo, a repetição termina quando o contador excede 10. Nesta seção, apresentamos um algoritmo em pseudocódigo (Fig. 4.6) e o programa correspondente (Fig. 4.7) para resolver este problema usando repetição controlada por contador. Na próxima seção, mostramos como os algoritmos em pseudocódigo são desenvolvidos. A repetição controlada por contador é freqüentemente chamada de *repetição definida*, uma vez que o número de repetições é conhecido antes de o laço começar a ser executado.

**Fig. 4.5** Fluxograma da estrutura de repetição `while`.

Observe as referências no algoritmo a um total e a um contador. O *total* é uma variável utilizada para acumular a soma de uma série de valores. O *contador* é uma variável utilizada para contar – neste caso, para contar o número de notas lidas. As variáveis utilizadas para armazenar totais normalmente devem ser inicializadas com zero antes de serem utilizadas em um programa; caso contrário, a soma incluiria o valor anterior armazenado na posição de memória do total.

*Ajustar o total para zero
Ajustar o contador de nota para um*

Enquanto (While) o contador de nota for menor do que ou igual a 10

*Ler próxima nota
Adicionar a nota ao total
Adicionar um ao contador de notas*

Atribuir à média da turma o valor total dividido por 10

Imprimir a média da turma

Fig. 4.6 Algoritmo em pseudocódigo que utiliza repetição controlada por contador para resolver o problema de média da turma.

```

1 // Fig. 4.7: Average1.java
2 // Programa de média da turma com repetição controlada por contador
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 public class Average1 {
8
9     // O método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        int total,           // soma das notas digitadas pelo usuário
13            gradeCounter, // número de notas lidas
14            gradeValue,   // valor da nota
15            average;      // média de todas as notas
16            String grade; // nota digitada pelo usuário
17
18        // Fase de Inicialização
19        total = 0;         // limpa total
  
```

Fig. 4.7 Programa de média da turma com repetição controlada por contador (parte 1 de 3).

```

20     gradeCounter = 1;    // prepara para executar o laço
21
22     // Fase de Processamento
23     while ( gradeCounter <= 10 ) {    // executa o laço 10 vezes
24
25         // solicita entrada e lê a nota digitada pelo usuário
26         grade = JOptionPane.showInputDialog(
27             "Enter integer grade: " );
28
29         // converte a nota de String para inteiro
30         gradeValue = Integer.parseInt( grade );
31
32         // adiciona gradeValue ao total
33         total = total + gradeValue;
34
35         // adiciona 1 ao gradeCounter
36         gradeCounter = gradeCounter + 1;
37
38     } // fim da estrutura while
39
40     // Fase de Conclusão
41     average = total / 10;    // executa divisão inteira
42
43     // exibe a média das notas do teste
44     JOptionPane.showMessageDialog( null,
45         "Class average is " + average, "Class Average",
46         JOptionPane.INFORMATION_MESSAGE );
47
48     System.exit( 0 );        // termina o programa
49
50 } // fim do método main
51
52 } // fim da classe Average1

```

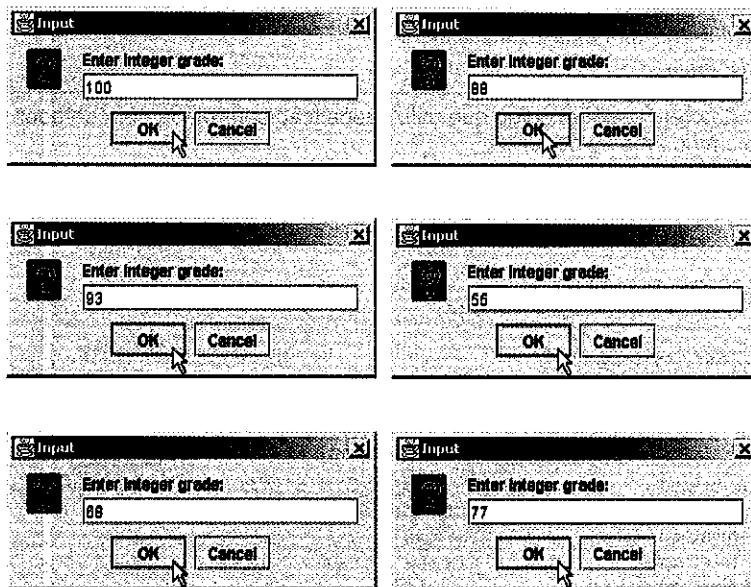


Fig. 4.7 Programa de média da turma com repetição controlada por contador (parte 2 de 3).

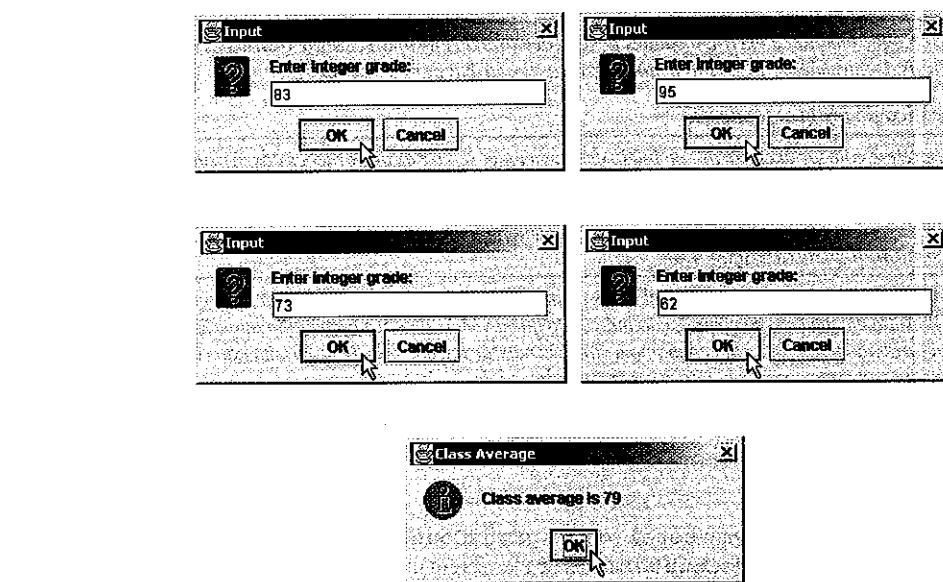


Fig. 4.7 Programa de média da turma com repetição controlada por contador (parte 3 de 3).



Boa prática de programação 4.6

Inicialize contadores e totais.

A linha 5,

```
import javax.swing.JOptionPane;
```

importa a classe `JOptionPane` para permitir que o programa leia os dados do teclado e gere como saída os dados para a tela utilizando o diálogo de entrada e o diálogo de mensagem mostrados no Capítulo 2.

A linha 7 inicia a definição da classe `Average1` do aplicativo. Lembre-se de que uma definição de classe de aplicativo deve conter um método `main` (linhas 10 a 49) para que o aplicativo seja executado.

As linhas 12 a 16,

```
int total,           // soma das notas digitadas pelo usuário
    gradeCounter,   // número de notas lidas
    gradeValue,     // valor da nota
    average;        // média de todas as notas
String grade;       // nota digitada pelo usuário
```

declaram que as variáveis `total`, `gradeCounter`, `gradeValue` e `average` são do tipo `int` e que a variável `grade` é do tipo `String`. A variável `grade` armazena o `String` que o usuário digita no diálogo de entrada. A variável `gradeValue` armazena o valor inteiro de `grade` depois que o programa o converte de um `String` para um `int`.

Repare que as declarações precedentes aparecem no corpo do método `main`. Lembre-se de que variáveis declaradas no corpo de uma definição de método são *variáveis locais* e só podem ser utilizadas a partir da linha de sua declaração no método até a chave direita de fechamento (`}`) da definição do método. A declaração de variável local deve aparecer antes de a variável ser utilizada nesse método. A variável local declarada em um método de uma classe não pode ser acessada diretamente por outros métodos de uma classe.

**Boa prática de programação 4.7**

Sempre coloque uma linha em branco antes de uma declaração que aparece entre instruções executáveis. Este formato faz as declarações se destacarem no programa e contribui para a sua clareza.

**Boa prática de programação 4.8**

Se você prefere colocar declarações no início de um método, separe as declarações das instruções executáveis nesse método com uma linha em branco, para destacar onde terminam as declarações e começam as instruções executáveis.

**Erro comum de programação 4.6**

Tentar usar o valor de uma variável local antes de inicializar a variável (normalmente com uma instrução de atribuição) resulta em um erro de compilação indicando que a variável pode não ter sido inicializada. O valor de uma variável local não pode ser usado enquanto a variável não for inicializada. O programa não vai ser compilado adequadamente enquanto a variável não receber um valor inicial.

As linhas 19 e 20,

```
total = 0;           // limpa o total
gradeCounter = 1;    // prepara para executar o laço
```

são instruções de atribuição que inicializam **total** com 0 e **gradeCounter** com 1. Observe que estas instruções inicializam as variáveis **total** e **gradeCounter** antes de elas serem utilizadas em cálculos.

A linha 23,

```
while ( gradeCounter <= 10 ) { // executa o laço 10 vezes
```

indica que a estrutura **while** deve continuar executando o laço (também dito *iterando*) enquanto o valor de **gradeCounter** for menor do que ou igual a 10.

As linhas 26 e 27,

```
grade = JOptionPane.showInputDialog(
    "Enter integer grade: " );
```

correspondem à instrução em pseudocódigo “Leia a próxima nota”. A instrução exibe um diálogo de entrada com o prompt “Enter integer grade:” na tela.

Depois que o usuário digita um valor para **grade**, o programa o converte de um **String** para um **int** na linha 30,

```
gradeValue = Integer.parseInt( grade );
```

Lembre-se de que a classe **Integer** é do pacote **java.lang**, que o compilador importa em todos os programas Java. O pseudocódigo para o problema de média da turma não reflete a instrução precedente. A instrução em pseudocódigo “Leia a próxima nota” exige que o programador implemente o processo de obter o valor fornecido pelo usuário e convertê-lo para um tipo que possa ser usado no cálculo da média. À medida que você aprender a programar, você descobrirá que você precisa de menos instruções em pseudocódigo para ajudá-lo a implementar um programa.

A seguir, o programa atualiza **total** com o novo **gradeValue** digitado pelo usuário. A linha 33,

```
total = total + gradeValue;
```

adiciona **gradeValue** ao valor anterior de **total** e atribui o resultado a **total**.

A linha 36,

```
gradeCounter = gradeCounter + 1;
```

adiciona 1 a **gradeCounter** para indicar que o programa processou uma nota e está pronto para ler a próxima nota fornecida pelo usuário. Incrementar **gradeCounter** é necessário para que a condição na estrutura **while** em algum momento se torne **false** e termine o laço.

A linha 41,

```
average = total / 10; // executa divisão inteira
```

atribui o resultado do cálculo da média à variável **average**. As linhas 44 a 46,

```
JOptionPane.showMessageDialog(
    null, "Class average is " + average, "Class Average",
    JOptionPane.INFORMATION_MESSAGE );
```

exibem um diálogo de mensagem informativa contendo o string "Class average is " seguido pelo valor da variável `average`. O string "Class Average" (o terceiro argumento) é o título do diálogo de mensagem.

A linha 48,

```
System.exit( 0 ); // termina o programa
```

termina o aplicativo.

Depois de compilar a definição de classe com `javac`, execute o aplicativo a partir da janela de comando com o comando

```
java Average1
```

Esse comando executa o interpretador Java e lhe diz que o método `main` para esse aplicativo está definido na classe `Average1`.

Observe que o cálculo da média no programa produziu um resultado inteiro. Na verdade, a soma dos valores das notas de pontuação neste exemplo é 794, que, quando dividido por 10, deve resultar em 79,4 (isto é, um número com fração decimal). Veremos como lidar com tais números (chamados *números de ponto flutuante*) na próxima seção.

4.9 Formulando algoritmos com refinamento passo a passo de cima para baixo: estudo de caso 2 (repetição controlada por sentinelas)

Vamos generalizar o problema da média da turma. Considere o seguinte problema:

Desenvolver um programa de média da turma que processará um número arbitrário de notas toda vez que o programa for executado.

No primeiro exemplo de média da turma, o número de notas (10) era conhecido com antecedência. Nesse exemplo, não se dá nenhuma indicação de quantas notas o usuário irá digitar. O programa deve processar um número arbitrário de notas. Como o programa pode determinar quando parar a leitura de notas? Como ele vai saber quando calcular e imprimir a média da turma?

Uma maneira de resolver esse problema é utilizar um valor especial chamado de *valor de sentinelas* (também chamado de *valor de sinalização*, *valor fictício* ou *valor de flag*) para indicar final de entrada de dados. O usuário vai digitando até todas as notas válidas terem sido fornecidas. O usuário então digita o valor da sentinelas para indicar que a última nota foi fornecida. A repetição controlada por sentinelas é freqüentemente chamada de *repetição indefinida*, uma vez que o número de repetições não é conhecido antes de o laço começar a ser executado.

Claramente, o valor de sentinelas deve ser escolhido de modo que não possa ser confundido com um valor de entrada aceitável. Uma vez que as notas em um teste são normalmente inteiros não-negativos, -1 é um valor de sentinelas aceitável para esse problema. Portanto, a execução do programa de média da turma poderia processar um conjunto de entradas como 95, 96, 75, 74, 89 e -1. Neste caso, o programa calcularia e imprimiria a média da turma para as notas 95, 96, 75, 74 e 89 (-1 é o valor de sentinelas, então ele não deve entrar no cálculo da média).



Erro comum de programação 4.7

Escolher um valor de sentinelas que também é um valor de dados válido resulta em um erro de lógica e pode impedir que o laço controlado por sentinelas seja adequadamente finalizado.

Abordamos o programa de média da turma com uma técnica chamada de *refinamento passo a passo de cima para baixo*, método essencial para o desenvolvimento de algoritmos bem estruturados. Iniciamos com uma representação em pseudocódigo do topo:

Determinar a média da turma para o teste

O topo é uma instrução única que resume a função total do programa. Como tal, o topo é, com efeito, uma representação completa de um programa. Infelizmente, o topo raramente fornece uma quantidade suficiente de detalhes pa-

ra escrever o algoritmo em Java. Portanto, iniciamos agora o processo de refinamento. Dividimos o topo em uma série de tarefas menores e listamos estas tarefas na ordem em que precisam ser executadas. Esse procedimento resulta no primeiro refinamento a seguir:

Iniciarizar as variáveis

Ler, somar e contar as notas do teste

Calcular e imprimir a média da turma

Este pseudocódigo usa somente a estrutura de seqüência – os passos listados ocorrem na ordem, um após o outro.



Observação de engenharia de software 4.4

Cada refinamento, bem como o próprio topo, é uma especificação completa do algoritmo; só o nível de detalhe varia.

Para prosseguir para o próximo nível de refinamento (isto é, o *segundo refinamento*), definimos variáveis específicas. Precisamos de uma soma total das notas, uma contagem de quantas notas foram processadas, uma variável para receber o valor de cada nota quando lida e uma variável para armazenar a média calculada. A instrução em pseudocódigo

Iniciarizar as variáveis

pode ser refinada como segue:

Iniciarizar total com zero

Iniciarizar contador com zero

Repare que apenas as variáveis *total* e *contador* são inicializadas antes de serem utilizadas; as variáveis *média* e *nota* (para média calculada e o valor fornecido pelo usuário, respectivamente) não precisam ser inicializadas uma vez que seus valores são substituídos à medida que são calculados ou lidos.

A instrução em pseudocódigo

Ler, somar e contar as notas do teste

exige uma estrutura de repetição (isto é, um laço) que leia sucessivamente cada nota. Não sabemos quantas notas o usuário vai fornecer, de modo que o programa utilizará repetição controlada por sentinelas. O usuário digita uma nota válida por vez. Depois de a última nota válida ter sido digitada, o usuário digita o valor da sentinelas. O programa testa o valor da sentinelas depois que cada nota for lida e termina o laço quando o usuário digita o valor da sentinelas. O segundo refinamento da instrução em pseudocódigo precedente é, então,

Ler a primeira nota (pode ser a sentinelas)

Enquanto (While) o usuário não inserir a sentinelas

Adicionar essa nota à soma total

Adicionar um ao contador de notas

Ler a próxima nota (pode ser a sentinelas)

Repare que, em pseudocódigo, não utilizamos chaves em torno do pseudocódigo que forma o corpo da estrutura *while*. Simplesmente recuamos o pseudocódigo sob o *while*, para mostrar que ele pertence ao *while*. Novamente, o pseudocódigo é apenas um auxílio informal ao desenvolvimento de programas.

A instrução em pseudocódigo

Calcular e imprimir a média da turma

pode ser refinada como segue:

Se (If) o contador não for igual a zero

Calcular a média como o total dividido pelo contador

Imprimir a média

senão (else)

Imprimir "Nenhuma nota foi lida"

Observe que estamos testando a possibilidade de divisão por zero – um *erro de lógica* que, se não detectado, faria com que o programa produzisse uma saída inválida. O segundo refinamento completo do algoritmo em pseudocódigo para o problema da média da turma é mostrado na Fig. 4.8.



Dica de teste e depuração 4.1

Ao realizar divisão por uma expressão cujo valor poderia ser zero, teste explicitamente esse caso e trate-o apropriadamente em seu programa (imprimindo uma mensagem de erro, por exemplo) em vez de permitir que a divisão por zero ocorra.



Boa prática de programação 4.9

Inclua linhas completamente em branco em programas em pseudocódigo para torná-los mais legíveis. As linhas em branco separam as estruturas de controle do pseudocódigo, bem como as fases dos programas.

Iniciarizar total com zero

Iniciarizar contador com zero

Ler a primeira nota (pode ser a sentinela)

Enquanto (While) o usuário não inserir a sentinela

Adicionar essa nota à soma total

Adicionar um ao contador de notas

Ler a próxima nota (pode ser a sentinela)

Se (If) o contador não for igual a zero

Calcular a média como o total dividido pelo contador

Imprimir a média

senão (else)

Imprimir “Nenhuma nota foi lida”

Fig. 4.8 Algoritmo em pseudocódigo que utiliza repetição controlada por sentinela para resolver o problema da média da turma.



Observação de engenharia de software 4.5

Muitos algoritmos podem ser logicamente divididos em três fases: uma fase de inicialização, que inicializa as variáveis do programa, uma fase de processamento, que obtém os valores dos dados e ajusta as variáveis do programa de acordo com eles, e uma fase de conclusão, que calcula e exibe os resultados.

O algoritmo em pseudocódigo da Fig. 4.8 resolve o problema mais geral da média da turma. Esse algoritmo foi desenvolvido depois de apenas dois níveis de refinamento. Às vezes são necessários mais níveis.



Observação de engenharia de software 4.6

O programador termina o processo de refinamento passo a passo de cima para baixo quando o algoritmo em pseudocódigo é especificado em detalhes suficientes para que o programador seja capaz de converter o pseudocódigo em um applet ou um aplicativo Java. Normalmente, implementar o applet Java ou aplicativo então é simples e direto.

O aplicativo Java e uma execução de exemplo são mostrados na Fig. 4.9. Embora cada nota seja um inteiro, o cálculo da média tende a produzir um número com ponto e fração decimal (isto é, um número real). O tipo `int` não pode representar números reais (isto é, números com fração decimal), de modo que esse programa utiliza o tipo de dados `double` para tratar números de ponto flutuante. O programa apresenta um operador especial chamado de *operador de coerção* para tratar a conversão de tipo de que precisaremos para o cálculo da média. Esses recursos serão explicados em detalhes na discussão do aplicativo.

Neste exemplo, vemos que as estruturas de controle podem ser empilhadas umas sobre as outras (em seqüência), assim como uma criança empilha os blocos de construção. A estrutura **while** (linhas 33 a 37) é seguida por uma estrutura **if/else** (linhas 52 a 63) em seqüência. Grande parte do código neste programa é idêntica ao código da Fig. 4.7; assim, concentramo-nos neste exemplo nos novos recursos e questões.

```

1 // Fig. 4.9: Average2.java
2 // Programa de média da turma com repetição controlada por sentinelas
3
4 // Pacotes do núcleo de Java
5 import java.text.DecimalFormat;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JOptionPane;
9
10 public class Average2 {
11
12     // método main inicia a execução de aplicativo Java
13     public static void main( String args[] )
14     {
15         int gradeCounter,      // número de notas lidas
16             gradeValue,        // valor da nota
17             total;            // soma das notas
18         double average;       // média de todas as notas
19         String input;        // nota digitada pelo usuário
20
21         // Fase de Inicialização
22         total = 0;           // limpa o total
23         gradeCounter = 0;    // prepara para executar o laço
24
25         // Fase de Processamento
26         // solicita entrada e lê a nota digitada pelo usuário
27         input = JOptionPane.showInputDialog(
28             "Enter Integer Grade, -1 to Quit:" );
29
30         // converte a nota de String para inteiro
31         gradeValue = Integer.parseInt( input );
32
33         while ( gradeValue != -1 ) {
34
35             // adiciona gradeValue ao total
36             total = total + gradeValue;
37
38             // adiciona 1 a gradeCounter
39             gradeCounter = gradeCounter + 1;
40
41             // solicita entrada e lê a nota digitada pelo usuário
42             input = JOptionPane.showInputDialog(
43                 "Enter Integer Grade, -1 to Quit:" );
44
45             // converte a nota de String para inteiro
46             gradeValue = Integer.parseInt( input );
47         }
48
49         // Fase de Conclusão
50         DecimalFormat twoDigits = new DecimalFormat( "0.00" );
51

```

Fig. 4.9 Programa de média da turma com repetição controlada por sentinelas (parte 1 de 2).

```

52     if ( gradeCounter != 0 ) {
53         average = (double) total / gradeCounter;
54
55         // exibe a média das notas do teste
56         JOptionPane.showMessageDialog( null,
57             "Class average is " + twoDigits.format( average ),
58             "Class Average", JOptionPane.INFORMATION_MESSAGE );
59     }
60     else
61         JOptionPane.showMessageDialog( null,
62             "No grades were entered", "Class Average",
63             JOptionPane.INFORMATION_MESSAGE );
64
65     System.exit( 0 ); // termina o programa
66
67 } // fim do método main
68
69 } // fim da classe Average2

```

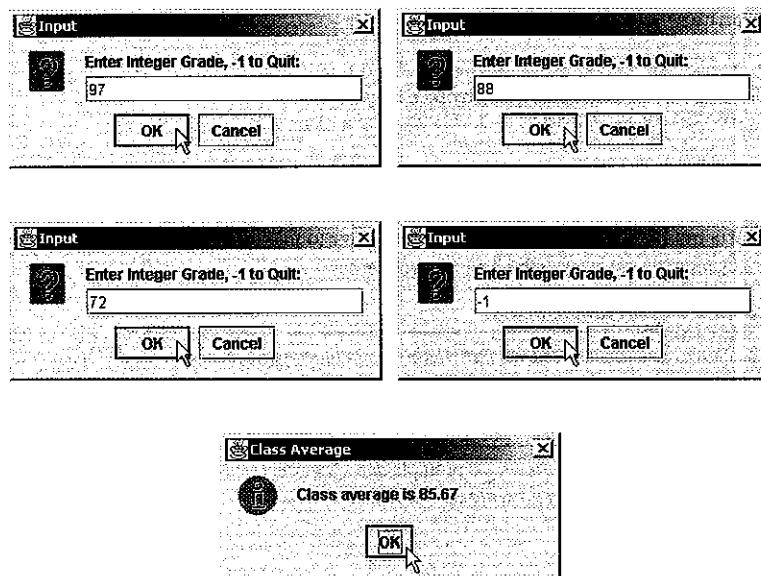


Fig. 4.9 Programa de média da turma com repetição controlada por sentinelas (parte 2 de 2).

A linha 18 declara a variável **double average**. Esta alteração permite armazenar o resultado do cálculo de média da turma como um número em ponto flutuante. A linha 23 inicializa **gradeCounter** com 0, porque nenhuma nota foi ainda lida. Lembre-se de que esse programa utiliza repetição controlada por sentinelas. Para manter um registro exato do número de notas que foram lidas, a variável **gradeCounter** é incrementada apenas quando um valor de nota válido é lido.

Repare na diferença na lógica do programa de repetição controlada por sentinelas comparada com a repetição controlada por contador na Fig. 4.7. Na repetição controlada por contador, cada iteração (laço) da estrutura **while** lê um valor digitado pelo usuário, pelo número especificado de iterações. Na repetição controlada por sentinelas, o programa lê e converte um valor (linhas 27 a 31) antes de alcançar a estrutura **while**. Esse valor determina se o fluxo de controle do programa deve entrar no corpo da estrutura **while**. Se a condição da estrutura **while for false**, o usuário digitou a sentinelas, de modo que o corpo da estrutura **while** não é executado (i. e., nenhuma nota foi digitada). Se, por outro lado, a condição for **true**, é iniciada a execução do corpo e o laço adiciona o valor digitado ao total.

do pelo usuário ao **total**. Depois de o valor ter sido processado, as linhas 42 a 46 no corpo do laço lêem o próximo valor digitado pelo usuário antes de o controle do programa atingir o fim do corpo da estrutura **while**. Quando o controle do programa chega na chave direita de fechamento {} do corpo na linha 47, a execução continua com o próximo teste de condição da estrutura **while** (linha 33). A condição utiliza o novo valor há pouco digitado pelo usuário para determinar se o corpo da estrutura **while** deve ser executado novamente. Repare que o próximo valor é sempre digitado pelo usuário imediatamente antes de o programa testar a condição da estrutura **while**. Essa estrutura permite determinar se o valor há pouco digitado pelo usuário é o valor de sentinela *antes* que o programa processe aquele valor (isto é, adicione-o ao **total**). Se o valor digitado é o valor da sentinela, a estrutura **while** termina e o programa não adiciona o valor ao **total**.

Observe o bloco no laço **while** na Fig. 4.9. Sem as chaves, as últimas quatro instruções no corpo do laço cairiam fora do laço, fazendo o computador interpretar esse código incorretamente conforme segue:

```
while ( gradeValue != -1 )

    // adiciona gradeValue ao total
    total = total + gradeValue;

    // adiciona 1 a gradeCounter
    gradeCounter = gradeCounter + 1;

    // solicita entrada e lê a nota digitada pelo usuário
    input = JOptionPane.showInputDialog(
        "Enter Integer Grade, -1 to Quit:" );

    // converte a nota de String para inteiro
    gradeValue = Integer.parseInt( input );
```

Esse código provocaria um laço infinito no programa se o usuário não digitasse sentinela **-1** como o valor de entrada nas linhas 27 e 28 (antes da estrutura **while**) no programa.



Erro comum de programação 4.8

Omitir as chaves que são necessárias para delinear um bloco pode levar a erros de lógica como laços infinitos. Para evitar este problema, alguns programadores colocam o corpo de todas as estruturas de controle entre chaves.



Boa prática de programação 4.10

Em um laço controlado por sentinela, os prompts que solicitam a entrada de dados devem lembrar explicitamente ao usuário qual é o valor que representa a sentinela.

A linha 50,

```
DecimalFormat twoDigits = new DecimalFormat( "0.00" );
```

declara **twoDigits** como uma referência para um objeto da classe **DecimalFormat** (pacote **java.text**). Objetos **DecimalFormat** formatam números. Neste exemplo, queremos gerar como saída a média da turma com dois dígitos à direita da casa decimal (isto é, arredondado para o centésimo mais próximo). A linha precedente cria um objeto **DecimalFormat** que é inicializado com o string **"0.00"**. Cada 0 é um *indicador de formato* que especifica uma posição de dígito obrigatória no número de ponto flutuante formatado. Esse formato particular indica que cada número formatado com **twoDigits** terá pelo menos um dígito à esquerda da casa decimal e exatamente dois dígitos à direita do ponto decimal. Se o número não atender aos requisitos de formatação, 0s são inseridos no número formatado nas posições obrigatórias. O operador **new** cria um objeto enquanto o programa está sendo executado, obtendo memória suficiente para armazenar um objeto do tipo especificado à direita de **new**. O processo de criar novos objetos é também conhecido como *criar uma instância* ou *instanciar um objeto*. O operador **new** é conhecido como *operador de alocação de memória dinâmica*. O valor entre parênteses depois do tipo em um operador **new** é utilizado para *inicializar* (i. e., atribuir um valor a) o novo objeto. O resultado da operação **new** é atribuído à referência **twoDigits** usando o *operador de atribuição =*. A instrução é lida como “**twoDigits** recebe o valor de **new DecimalFormat("0.00")**”.



Observação de engenharia de software 4.7

Normalmente, os objetos são criados com o operador `new`. Uma exceção a essa regra é um literal de string que está entre aspas, como `"hello"`. Os literais de string são tratados como objetos da classe `String` e são instanciados automaticamente.

As médias nem sempre são avaliadas como valores inteiros. Freqüentemente, a média é um valor como 3,333 ou 2,7 que contém uma parte fracionária. Esses valores são conhecidos como *números em ponto flutuante* e são representados pelo tipo de dados `double`. A variável `average` é declarada como do tipo `double` para capturar o resultado fracionário de nosso cálculo. Entretanto, o resultado do cálculo `total / gradeCounter` é um inteiro, uma vez que `total` e `gradeCounter` são variáveis inteiros. Dividir dois inteiros resulta em uma *divisão inteira*, na qual qualquer parte fracionária do cálculo é perdida (isto é, *truncada*). A parte fracionária do cálculo é perdida antes que o resultado possa ser atribuído a `average`, porque o cálculo é executado antes que a atribuição aconteça.

Para executar um cálculo em ponto flutuante com valores inteiros, devemos criar valores temporários que sejam números de ponto flutuante para o cálculo. Java fornece o operador *unário de coerção* para realizar esta tarefa. A linha 53

```
average = (double) total / gradeCounter;
```

utiliza o operador de coerção (`double`) para criar uma cópia em ponto flutuante temporária de seu operando – `total`. O uso de um operador de coerção dessa maneira chama-se *conversão explícita*. O valor armazenado em `total` ainda é um inteiro. O cálculo agora consiste em um valor de ponto flutuante (a versão `double` temporária de `total`) dividido pelo inteiro `gradeCounter`. Java sabe como avaliar apenas as expressões aritméticas em que os tipos de dados dos operandos são idênticos. Para assegurar que os operandos sejam do mesmo tipo, Java realiza uma operação chamada *promoção* (ou *conversão implícita*) sobre operandos selecionados. Por exemplo, em uma expressão que contém os tipos de dados `int` e `double`, os valores dos operandos `int` são *promovidos* a valores `double` para uso na expressão. Nesse exemplo, Java promove `gradeCounter` para o tipo `double` e então o programa executa o cálculo e atribui o resultado da divisão em ponto flutuante a `average`. Mais adiante neste capítulo discutimos todos os tipos de dados padrão e sua ordem de promoção.



Erro comum de programação 4.9

O operador de coerção pode ser usado para converter entre tipos numéricos primitivos e entre tipos de classes relacionados (como discutimos no Capítulo 9). Fazer a coerção de uma variável para o tipo errado pode provocar erros de compilação ou erros durante a execução.

Os operadores de coerção estão disponíveis para qualquer tipo de dados. O operador de coerção é formado colocando-se parênteses em volta do nome de um tipo de dados. O operador é um *operador unário* (isto é, um operador que recebe somente um operando). No Capítulo 2, estudamos os operadores aritméticos binários. Java também suporta versões unárias dos operadores mais (+) e menos (-), de modo que o programador pode escrever expressões como `-7` ou `+5`. Os operadores de coerção são associados da direita para a esquerda e têm a mesma precedência que outros operadores unários, como `+ unário` e `- unário`. Essa precedência é um nível mais alto que aquela dos *operadores multiplicativos* `*`, `/` e `%` e um nível mais baixo que aquela dos parênteses (veja a tabela de precedência de operadores no Apêndice C). Indicamos o operador de coerção com a notação `(tipo)` em nossas tabelas de precedência, para indicar que qualquer nome de tipo pode ser utilizado para formar um operador de coerção.



Erro comum de programação 4.10

Utilizar números de ponto flutuante de uma maneira que supõe que eles sejam representados precisamente pode levar a resultados incorretos. Números reais são representados apenas aproximadamente por computadores.



Erro comum de programação 4.11

Supor que a divisão inteira arredonda (em vez de truncar) pode levar a resultados incorretos.



Boa prática de programação 4.11

Não compare valores de ponto flutuante em termos de igualdade ou desigualdade. Em vez disso, teste se o valor absoluto da diferença entre dois valores em ponto flutuante é menor que um valor pequeno especificado.

Apesar do fato de os números de ponto flutuante não serem sempre 100% precisos, eles possuem inúmeras aplicações. Por exemplo, quando falamos de uma temperatura “normal” do corpo de 36,9°C, não precisamos ter a precisão de um grande número de dígitos. Quando vemos a temperatura em um termômetro e a lemos como 36,9°C, na realidade ela pode ser 36,899473210643°C. A questão principal aqui é que chamar esse número simplesmente de 36,9°C é suficiente para a maioria das aplicações.

Outra maneira pela qual são gerados números de ponto flutuante é através da divisão. Quando dividimos 10 por 3, o resultado é 3,333333..., com a seqüência de 3s se repetindo infinitamente. O computador aloca apenas uma quantidade fixa de espaço para armazenar tal valor; portanto, evidentemente o valor em ponto flutuante armazenado pode ser apenas uma aproximação.

4.10 Formulando algoritmos com refinamento passo a passo de cima para baixo: estudo de caso 3 (estruturas de controle aninhadas)

Vamos trabalhar em outro problema completo. Mais uma vez formularemos o algoritmo utilizando pseudocódigo e refinamento passo a passo de cima para baixo e escreveremos um programa Java correspondente. Considere a seguinte definição do problema:

Uma faculdade oferece um curso que prepara os candidatos a obter uma licença estadual para os corretores de imóveis. No último ano, grande parte dos alunos que concluiu esse curso prestou o exame. Naturalmente, a faculdade quer saber qual o desempenho de seus alunos no exame. Você foi contratado para escrever um programa que resumisse os resultados. Você recebeu uma lista desses 10 alunos. Ao lado de cada nome, está escrito 1 se o aluno passou no exame e 2 se o aluno foi reprovado.

Seu programa deve analisar os resultados do exame desta forma:

1. *Ler cada resultado do teste (isto é, um 1 ou um 2). Exibir a mensagem “Digitar resultado” na tela toda vez que o programa solicitar o resultado de outro teste.*
2. *Contar o número de resultados da cada tipo.*
3. *Exibir um resumo dos resultados do teste para indicar o número de alunos que foram aprovados e o número de alunos reprovados.*
4. *Se mais de oito alunos foram aprovados no exame, imprimir a mensagem “Aumentar a taxa de matrícula”.*

Depois de ler a definição do problema cuidadosamente, fazemos as seguintes observações sobre o problema:

1. O programa deve processar resultados do teste para 10 alunos. Um laço controlado por contador será utilizado.
2. Cada resultado do teste é um número – 1 ou 2. Toda vez que o programa ler um resultado de teste, deve determinar se o número é 1 ou 2. Em nosso algoritmo, testamos se o número é 1. Se o número não for 1, assumimos que é 2. (Um exercício no final do capítulo considera as consequências dessa suposição.)
3. Utilizam-se dois contadores para monitorar os resultados do exame – um para contar o número de alunos que foram aprovados no exame e outro para contar o número de alunos que foram reprovados no exame.
4. Depois que o programa processou todos os resultados, ele deve decidir se mais de oito alunos foram aprovados no exame.

Vamos prosseguir com o refinamento passo a passo de cima para baixo. Iniciamos com uma representação do topo do pseudocódigo:

Analisar os resultados do exame e decidir se a taxa de matrícula deve ser aumentada

Mais uma vez, é importante enfatizar que a parte superior é uma representação completa do programa, mas possivelmente vários refinamentos serão necessários antes que o pseudocódigo possa ser naturalmente transformado em um programa Java. Nossa primeiro refinamento é

Iniciar variáveis

Ler as 10 notas de exame e contar aprovações e reparações

Imprimir um resumo dos resultados de exame e decidir se a taxa de matrícula deve ser aumentada

Aqui, igualmente, mesmo tendo uma representação completa do programa inteiro, é necessário refinamento adicional. Agora empregamos variáveis específicas. Precisamos de contadores para registrar as aprovações e reparações de um contador para controlar o processo de laço e de uma variável para armazenar a entrada do usuário. A instrução em pseudocódigo

Iniciarizar as variáveis

pode ser refinada assim:

Iniciarizar as aprovações com zero

Iniciarizar as reparações com zero

Iniciarizar o aluno com um

Repare que apenas os contadores para o número de aprovações, para o número de reparações e para o número de alunos são inicializados. A instrução em pseudocódigo

Ler as 10 notas do teste e contar aprovações e reparações

exige um laço que lê sucessivamente o resultado de cada exame. Aqui sabemos com antecedência que há precisamente 10 resultados de exame, de modo que um laço controlado por contador é apropriado. Dentro do laço (isto é, aninhada dentro do laço) uma estrutura de seleção dupla determina se cada resultado de exame é uma aprovação ou uma reparação e incrementa o contador apropriado de acordo. O refinamento da instrução de pseudocódigo precedente é, portanto,

Enquanto (While) o contador de alunos for menor do que ou igual a 10

Ler o próximo resultado de exame

Se (If) o aluno foi aprovado

Adicionar um às aprovações

senão (else)

Adicionar um às reparações

Adicionar um ao contador de alunos

Repare no uso de linhas em branco para destacar a estrutura de controle *if/else* a fim de melhorar a legibilidade do programa. A instrução em pseudocódigo

Imprimir um resumo dos resultados do exame e decidir se a taxa de matrícula deve ser aumentada

pode ser refinada como segue:

Imprimir o número de aprovações

Imprimir o número de reparações

Se (If) mais de oito alunos foram aprovados

Imprimir "Aumentar a taxa de matrícula"

O segundo refinamento completo aparece na Fig. 4.10. Observe que o pseudocódigo também utiliza linhas em branco para destacar a estrutura *while* por questões de legibilidade do programa.

Iniciarizar aprovações com zero

Iniciarizar reprovações com zero

Iniciarizar aluno com um

Enquanto (While) o contador de alunos for menor do que ou igual a 10

Ler o próximo resultado de exame

Se (If) o aluno foi aprovado

Adicionar um às aprovações

senão (else)

Adicionar um às reprovações

Adicionar um ao contador de alunos

Imprimir o número de aprovações

Imprimir o número de reprovações

Se (If) mais de oito alunos foram aprovados

Imprimir "Aumentar a taxa de matrícula"

Fig. 4.10 Pseudocódigo para o problema dos resultados de um exame.

Esse pseudocódigo agora está suficientemente refinado para ser convertido para Java. O programa Java e dois exemplos de execução são mostrados na Fig. 4.11.

```

1 // Fig. 4.11: Analysis.java
2 // Análise dos resultados de um exame.
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 public class Analysis {
8
9     // método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        // inicialização de variáveis em declarações
13        int passes = 0,           // número de aprovações
14            failures = 0,         // número de reprovações
15            student = 1,          // contador de alunos
16            result;              // um resultado de exame
17            String input,          // valor digitado pelo usuário
18            output;               // string de saída
19
20        // processa 10 alunos; laço controlado por contador
21        while ( student <= 10 ) {
22
23            // obtém resultado do usuário
24            input = JOptionPane.showInputDialog(
25                "Enter result (1=pass,2=fail)" );
26
27            // converte o resultado para int
28            result = Integer.parseInt( input );

```

Fig. 4.11 Programa Java para o problema dos resultados de um exame (parte 1 de 3).

```

29          // processa o resultado
30      if ( result == 1 )
31          passes = passes + 1;
32      else
33          failures = failures + 1;
34
35      student = student + 1;
36  }
37
38
39      // fase de conclusão
40      output = "Passed: " + passes +
41                  "\nFailed: " + failures;
42
43      if( passes > 8 )
44          output = output + "\nRaise Tuition";
45
46      JOptionPane.showMessageDialog( null, output,
47          "Analysis of Examination Results",
48          JOptionPane.INFORMATION_MESSAGE );
49
50      System.exit( 0 );    // termina o aplicativo
51
52  } // fim do método main
53
54 } // fim da classe Analysis

```

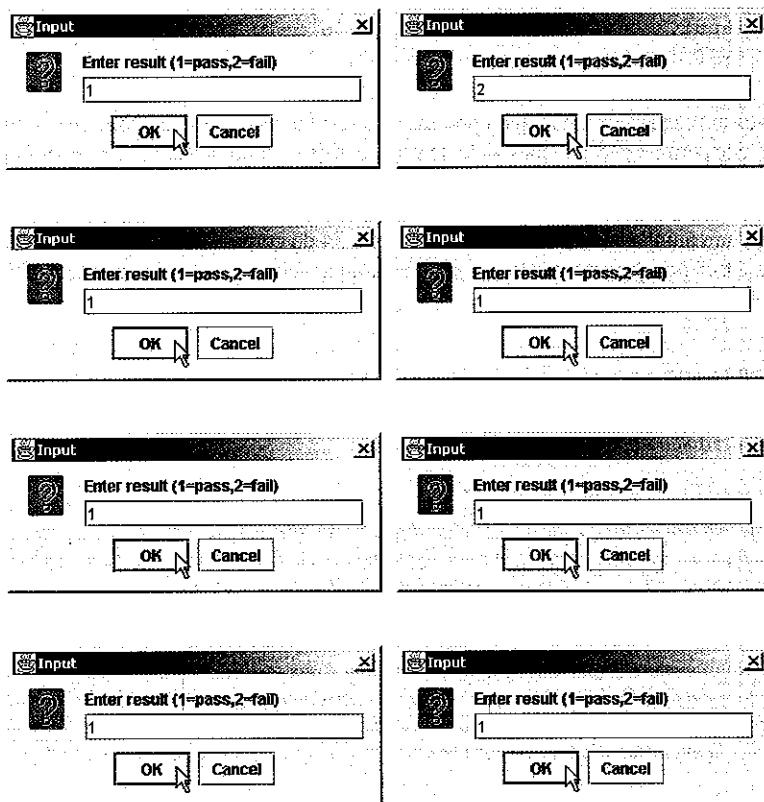


Fig. 4.11 Programa Java para o problema dos resultados de um exame (parte 2 de 3).

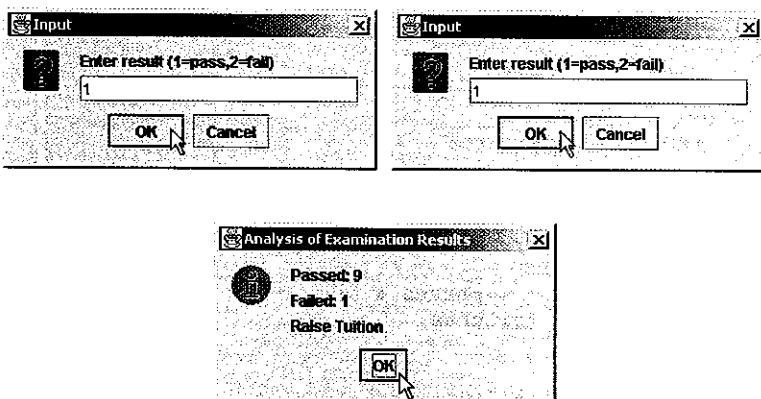


Fig. 4.11 Programa Java para o problema dos resultados de um exame (parte 3 de 3).

As linhas 13 a 18

```
int passes = 0,           // número de aprovações
    failures = 0,          // número de reprovações
    student = 1,            // contador de alunos
    result;                 // um resultado de exame
String input,              // valor digitado pelo usuário
    output;                // string de saída
```

declaram as variáveis utilizadas em **main** para processar os resultados do exame. Observe que tiramos proveito de um recurso de Java que permite que a inicialização de variável seja incorporada em declarações (a **passes** e a **failures** é atribuído 0 e a **student** é atribuído 1). Os programas que empregam laços podem exigir a inicialização no início de cada repetição; essa inicialização normalmente ocorreria em instruções de atribuição.

Observe a estrutura aninhada **if/else** nas linhas 31 a 34 do corpo da estrutura **while**. Observe também o uso da referência a **String output** nas linhas 40, 41 e 44, para construir o *string* que será exibido em um diálogo de mensagem nas linhas 46 a 48.



Boa prática de programação 4.12

Inicializar variáveis locais quando são declaradas em métodos ajuda o programador a evitar mensagens do compilador advertindo sobre dados não-inicializados.



Observação de engenharia de software 4.8

A experiência tem mostrado que a parte mais difícil de resolver um problema em um computador é desenvolver o algoritmo para a solução. Uma vez que um algoritmo correto tenha sido especificado, o processo de produção de um programa Java funcional a partir do algoritmo é normalmente simples.



Observação de engenharia de software 4.9

Muitos programadores experientes escrevem programas sem jamais utilizar ferramentas de desenvolvimento de programa como pseudocódigo. Esses programadores acreditam que seu objetivo final é resolver o problema em um computador e que escrever pseudocódigo só retarda a produção de saídas finais. Embora possa funcionar para problemas simples e familiares, isso pode levar a erros sérios em projetos grandes e complexos.

4.11 Operadores de atribuição

Java fornece vários operadores de atribuição para abreviar expressões de atribuição. Por exemplo, você pode abreviar a instrução

```
c = c + 3;
```

com o *operador de atribuição de adição*, `+=`, como

```
c += 3;
```

O operador `+=` adiciona o valor da expressão à direita do operador ao valor da variável à esquerda do operador, e armazena o resultado na variável à esquerda do operador. Qualquer instrução na forma

```
variável = variável operador expressão;
```

onde *operador* é um dos operadores binários `+`, `-`, `*`, `/` ou `%` (ou outros que discutiremos mais adiante no texto), pode ser escrita na forma

```
variável operador = expressão;
```

Assim, a expressão de atribuição `c += 3` adiciona 3 a `c`. A Fig. 4.12 mostra os operadores aritméticos de atribuição, exemplos de expressões que utilizam esses operadores e explicações sobre o que os operadores fazem.

Operador de atribuição	Expressão de exemplo	Explicação	Atribui
<i>Suponha: int c = 3, d = 5, e = 4, f = 6, g = 12;</i>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 a c
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 a d
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 a e
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>	2 a f
<code>%=</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3 a g

Fig. 4.12 Operadores aritméticos de atribuição.

Dica de desempenho 4.1



Os programadores podem escrever programas um pouco mais rapidamente e os compiladores podem compilar programas um pouco mais rapidamente quando são utilizados operadores de atribuição abreviados. Alguns compiladores geram códigos que executam mais rapidamente quando são utilizados operadores de atribuição abreviados.

Dica de desempenho 4.2



Muitas das dicas de desempenho mencionadas neste texto resultam em melhorias nominais, de modo que o leitor pode ser tentado a ignorá-las. Melhoras significativas de desempenho são freqüentemente obtidas quando uma melhora supostamente nominal é colocada em um laço que pode ser repetido um grande número de vezes.

4.12 Operadores de incremento e decremento

Java fornece o *operador de incremento* unário, `++`, e o *operador de decremento* unário, `--`, que são resumidos na Fig. 4.13. O programa pode incrementar o valor de uma variável chamada `c` por 1 com o operador de incremento `++` em vez das expressões `c = c + 1` ou `c += 1`. Se o operador de incremento ou decremento é colocado antes de uma variável, ele passa a ser chamado de *operador de pré-incremento* ou *pré-decremento*, respectivamente. Se o operador de incremento ou decremento é colocado depois de uma variável, ele passa a ser chamado de *operador de pós-incremento* ou *pós-decremento*, respectivamente.

Operador	Chamado de	Expressão de exemplo	Explicação
<code>++</code>	pré-incremento	<code>++a</code>	Incrementa <code>a</code> por 1, depois utiliza o novo valor de <code>a</code> na expressão em que <code>a</code> reside.
<code>++</code>	pós-incremento	<code>a++</code>	Utiliza o valor atual de <code>a</code> na expressão em que <code>a</code> reside, depois incrementa <code>a</code> por 1.
<code>--</code>	pré-decremento	<code>--b</code>	Decrementa <code>b</code> por 1, depois utiliza o novo valor de <code>b</code> na expressão em que <code>b</code> reside.
<code>--</code>	pós-decremento	<code>b--</code>	Utiliza o valor atual de <code>b</code> na expressão em que <code>b</code> reside, depois decrementa <code>b</code> por 1.

Fig. 4.13 Operadores de incremento e decremento.

Pré-incrementar (pré-decrementar) uma variável faz com que a variável seja incrementada (decrementada) por 1, e depois o novo valor da variável é utilizado na expressão em que ela aparece. Pós-incrementar (pós-decrementar) a variável faz com que o valor atual da variável seja utilizado na expressão em que ela aparece, depois o valor da variável é incrementado (decrementado) por 1.

O aplicativo da Fig. 4.14 demonstra a diferença entre a versão de pré-incremento e a versão de pós-incremento do operador de incremento `++`. Pós-incrementar a variável `c` faz com que ela seja incrementada depois de ser utilizada na chamada do método `System.out.println` (linha 13). Pré-incrementar a variável `c` faz com que ela seja incrementada antes de ser utilizada na chamada do método `System.out.println` (linha 20).

O programa exibe o valor de `c` antes e depois de o operador `++` ser utilizado. O operador de decremeno (`--`) funciona de maneira semelhante.



Boa prática de programação 4.13

Os operadores unários devem ser colocados ao lado de seus operandos, sem espaços no meio.

```

1 // Fig. 4.14: Increment.java
2 // Pré-incrementando e pós-incrementando
3
4 public class Increment {
5
6     // método main inicia a execução do aplicativo Java
7     public static void main( String args[] )
8     {
9         int c;
10
11         c = 5;
12         System.out.println( c );           // imprime 5
13         System.out.println( c++ );        // imprime 5 depois pós-incrementa
14         System.out.println( c );           // imprime 6
15
16         System.out.println();            // pula uma linha
17
18         c = 5;
19         System.out.println( c );           // imprime 5
20         System.out.println( ++c );        // pré-incrementa depois imprime 6
21         System.out.println( c );           // imprime 6
22
23     } // fim do método main
24
25 } // fim da classe Increment

```

Fig. 4.14 A diferença entre pré-incrementar e pós-incrementar (parte 1 de 2).

```

5
5
6

5
6
6

```

Fig. 4.14 A diferença entre pré-incrementar e pós-incrementar (parte 2 de 2).

A linha 16

```
System.out.println(); // pula uma linha
```

utiliza `System.out.println` para gerar como saída uma linha em branco. Se `println` não receber nenhum argumento, ele simplesmente envia para a saída um caractere de nova linha.

Os operadores aritméticos de atribuição e os operadores de incremento e decremento podem ser usados para simplificar instruções de um programa. Por exemplo, as três instruções de atribuição na Fig 4.11 (linhas 32, 34 e 36),

```

passes = passes + 1;
failures = failures + 1;
student = student + 1;

```

podem ser escritas de maneira mais concisa com operadores de atribuição como

```

passes += 1;
failures += 1;
student += 1;

```

com operadores de pré-incremento como

```

++passes;
++failures;
++student;

```

ou com operadores de pós-incremento como

```

passes++;
failures++;
student++;

```

É importante aqui comentar que, ao incrementar ou decrementar uma variável em uma instrução isolada, as formas de pré-incremento e de pós-incremento têm o mesmo efeito e as formas de pré-decremento e pós-decremento têm o mesmo efeito. Somente quando uma variável aparece no contexto de uma expressão maior é que pré-incrementar e pós-incrementar a variável têm efeitos diferentes (e igualmente para pré-decrementar e pós-decrementar).



Erro comum de programação 4.12

Tentar utilizar o operador de incremento ou decremento em uma expressão que não seja um lvalue é um erro de sintaxe. O lvalue é uma variável ou expressão que pode aparecer no lado esquerdo de uma operação de atribuição. Por exemplo, escrever `++(x + 1)` é um erro de sintaxe, porque `(x + 1)` não é um lvalue.

O gráfico na Fig. 4.15 mostra a precedência e a associatividade dos operadores apresentados até este ponto. Os operadores são mostrados de cima para baixo em ordem decrescente de precedência. A segunda coluna descreve a associatividade dos operadores em cada nível de precedência. Repare que o operador condicional (`? :`), os operadores unários de incremento (`++`), decremento (`--`), adição (`+`), subtração (`-`) e as coerções, assim como os operadores de atribuição `=`, `+=`, `-=`, `*=`, `/=` e `%=`, associam-se da direita para a esquerda. Todos os outros operadores no gráfico de precedência de operadores da Fig. 4.15 se associam da esquerda para a direta. A terceira coluna dá nomes aos grupos de operadores.

Operadores	Associatividade	Tipo
<code>()</code>	da esquerda para a direita	parênteses
<code>++ --</code>	da direita para a esquerda	únario pós-fixo
<code>++ -- + - (tipo)</code>	da direita para a esquerda	únario
<code>* / %</code>	da esquerda para a direita	de multiplicação
<code>+ -</code>	da esquerda para a direita	de adição
<code>< <= > >=</code>	da esquerda para a direita	relacional
<code>== !=</code>	da esquerda para a direita	de igualdade
<code>? :</code>	da direita para a esquerda	condicional
<code>= += -= *= /= %=</code>	da direita para a esquerda	de atribuição

Fig. 4.15 Precedência e associatividade dos operadores discutidos até agora.

4.13 Tipos de dados primitivos

A tabela na Fig. 4.16 lista os tipos de dados primitivos em Java. Os tipos primitivos são os blocos de construção para os mais complicados. Assim como suas linguagens predecessoras, C e C++, Java exige que todas as variáveis tenham um tipo definido antes que elas possam ser utilizadas em um programa. Por essa razão, Java é conhecida como uma *linguagem fortemente tipada*.

Em programas C e C++, os programadores tinham de escrever freqüentemente versões separadas dos programas para suportar plataformas de computador diferentes, porque não havia garantia de que os tipos de dados primitivos seriam idênticos de um computador para outro. Por exemplo, o valor `int` em uma máquina poderia ser representado por 16 bits (2 bytes) de memória enquanto o valor `int` em outra máquina poderia ser representado por 32 bits (4 bytes) de memória. Em Java, valores `int` são sempre de 32 bits (4 bytes).



Dica de portabilidade 4.1

Ao contrário das linguagens de programação C e C++, os tipos primitivos em Java são portáveis entre todas as plataformas de computador que suportam Java. Este e muitos outros recursos de portabilidade de Java permitem que os programadores escrevam programas uma só vez, sem saber qual a plataforma de computador em que o programa será executado. Esse atributo é às vezes conhecido como "WORA (Write Once, Run Anywhere – escreva uma vez, rode em qualquer lugar)".

Cada tipo de dados na Fig. 4.16 está listado com seu tamanho em *bits* (existem 8 bits em um *byte*) e seu intervalo de valores. Como os projetistas de Java querem que ele seja o mais portável possível, eles escolheram utilizar padrões reconhecidos internacionalmente para os formatos de caractere (Unicode) e para os números de ponto flutuante (IEEE 754).

Quando as variáveis de instância dos tipos de dados primitivos são declaradas em uma classe, atribuem-se a elas automaticamente valores *default*, a menos que especificado de maneira contrária pelo programador. As variáveis de instância dos tipos `char`, `byte`, `short`, `int`, `long`, `float` e `double` é atribuído o valor 0 por *default*. As variáveis do tipo `boolean` recebem por *default* o valor `false`.

Tipo	Tamanho em bits	Valores	Padrão
<code>boolean</code>	8	<code>true</code> ou <code>false</code>	
<code>char</code>	16	<code>'\u0000'</code> a <code>'\uFFFF'</code> (0 a 65535)	(conj. de caracteres Unicode ISO)

Fig. 4.16 Os tipos de dados primitivos em Java (parte 1 de 2).

Tipo	Tamanho em bits	Valores	Padrão
byte	8	-128 a +127 (- 2^7 a 2^7-1)	
short	16	-32.768 a +32.767 (- 2^{15} a $2^{15}-1$)	
int	32	-2.147.483.648 a +2.147.483.647 (- 2^{31} a $2^{31}-1$)	
long	64	-9.223.372.036.854.775.808 a +9.223.372.036.854.775.807 (- 2^{63} a $2^{63}-1$)	
float	32	<i>Intervalo negativo:</i> -3,4028234663852886E+38 a -1,40129846432481707E-45 <i>Intervalo positivo:</i> 1,40129846432481707E-45 a 3,4028234663852886E+38	(ponto flutuante IEEE 754)
double	64	<i>Intervalo negativo:</i> -1,7976931348623157E+308 a -4,94065645841246544E-324 <i>Intervalo positivo:</i> 4,94065645841246544E-324 a 1,7976931348623157E+308	(ponto flutuante IEEE 754)

Fig. 4.16 Os tipos de dados primitivos em Java (parte 2 de 2).

4.14 (Estudo de caso opcional) Pensando em objetos: identificando os atributos das classes

Na seção “Pensando em objetos” 3.8, começamos a primeira fase de um projeto orientado a objetos (OOD) para o nosso simulador de elevador – identificar as classes necessárias para implementar o simulador. Começamos listando os substantivos na definição do problema e depois criamos uma classe separada para cada categoria de substantivos e frases com substantivos que executa um trabalho importante na simulação do elevador. Representamos, então, as classes e seus relacionamentos em um diagrama de classes UML (Fig. 3.23). As classes possuem *atributos* (dados) e *operações* (comportamentos). Os atributos de classes são implementados em programas Java como variáveis; as operações de classes são implementadas como métodos. Nesta seção, determinamos muitos dos atributos de classes necessários para implementar o simulador do elevador. No Capítulo 5, examinamos como estes atributos representam o *estado*, ou a condição, de um objeto. No Capítulo 6, determinaremos o comportamento das classes. No Capítulo 7, concentramo-nos nas interações, freqüentemente chamadas de *colaborações*, entre os objetos no simulador de elevador.

Considere os atributos de alguns objetos do mundo real: os atributos de uma pessoa incluem altura e peso, por exemplo. Os atributos de um rádio incluem a sintonia, o ajuste de volume e se ele está ajustado para AM ou FM. Os atributos de um carro incluem as leituras do velocímetro, a quantidade de combustível no tanque, que marcha está engrenada, etc. Os atributos de um computador pessoal incluem o fabricante (por exemplo, Sun, Apple, IBM ou Compaq), tipo de tela (por exemplo, monocromática ou em cores), tamanho da memória principal (em *megabytes*), tamanho do disco rígido (em *gigabytes*), etc.

Podemos identificar os atributos das classes em nosso sistema procurando palavras e frases descritivas na definição do problema. Para cada palavra ou frase descritiva que encontramos, criamos um atributo e atribuímos aque-

le atributo a uma classe. Também criamos atributos para representar quaisquer dados adicionais que uma classe possa necessitar (à medida que a necessidade destes dados se torna clara ao longo do processo).

Começamos a análise da definição do problema procurando atributos distintos para cada classe. A Fig. 4.17 lista as palavras ou frases da definição do problema que descrevem cada classe. A frase “O usuário pode criar uma quantidade qualquer de pessoas na simulação” implica que o modelo introduzirá diversos objetos **Person** durante a execução. Precisamos de um valor inteiro que represente a quantidade de pessoas na simulação a cada momento dado, porque podemos querer monitorar, ou identificar, as pessoas em nosso modelo. Como mencionado na Seção 2.9, o objeto **ElevatorModel** age como o “representante” para o modelo (embora o modelo consista em diversas classes) para interação com outras partes do sistema (neste caso, o usuário é uma parte do sistema), de modo que atribuímos o atributo **numberOfPeople** à classe **ElevatorModel**.

A classe **Elevator** contém diversos atributos. As frases “está se movimentando” e “é chamado” descrevem possíveis estados de **Elevator** (apresentamos os estados na próxima seção “Pensando em objetos”), de modo que incluímos **moving** (se movendo) e **summoned** (chamado) como atributos do tipo **boolean**. **Elevator** também chega em um “andar de destino”, de modo que incluímos o atributo **destinationFloor**, que representa o **Floor** no qual o **Elevator** vai chegar. Embora a definição do problema não mencione explicitamente que o **Elevator** sai de um **Floor** atual, podemos prever um outro atributo chamado **currentFloor** que representa o **Floor** no qual o elevador está parado. A definição do problema especifica que “tanto o elevador quanto cada um dos andares têm capacidade para apenas uma pessoa”, de modo que incluímos o atributo **capacity** para a classe **Elevator** (e para a classe **Floor**) e ajustamos seus valores para **1**. Por último, a definição do problema especifica que o elevador “leva cinco segundos para se movimentar entre os andares”, de modo que introduzimos o atributo **travelTime** e ajustamos seu valor para **5**.

A classe **Person** contém diversos atributos. O usuário precisa ser capaz de “criar uma pessoa única”, o que implica que cada objeto **Person** deve ter um identificador único. Atribuímos o atributo inteiro **ID** ao objeto **Person**. O atributo **ID** ajuda a identificar aquele objeto **Person**. Além disso, a definição do problema especifica que a **Person** pode estar “esperando naquele andar para entrar no elevador”. Portanto, “esperando” é um estado que o

Classe	Palavras e frases descritivas
ElevatorModel	quantidade de pessoas na simulação
ElevatorShaft	[nenhuma palavra ou frase descritiva]
Elevator	se movendo chamado andar atual andar de destino capacidade para apenas uma pessoa cinco segundos para se mover de um andar até o outro
Person	única esperando/se movendo andar atual
Floor	primeiro ou segundo; capacidade para apenas uma pessoa
FloorButton	pressionado / desligado
ElevatorButton	pressionado / desligado
FloorDoor	porta fechada / porta aberta
ElevatorDoor	porta fechada / porta aberta
Bell	[nenhuma palavra ou frase descritiva]
Light	iluminada / apagada

Fig. 4.17 Palavras e frases descritivas da definição do problema.

objeto **Person** pode assumir. Embora não mencionado explicitamente, se não está esperando pelo **Elevator**, a **Person** está se movendo em direção ao (ou se afastando do) **Elevator**. Atribuímos o atributo **moving**, do tipo **boolean**, à classe **Person**. Quando este atributo é ajustado para **false**, a **Person** “está esperando”. Por último, a frase “naquele andar” implica que a **Person** ocupa um andar. Não podemos atribuir uma referência a um **Floor** à classe **Person**, porque estamos interessados somente em atributos. Entretanto, queremos incluir a posição do objeto **Person** no modelo, de modo que incluímos o atributo **currentFloor** (andar atual), o qual pode ter um valor de 1 ou 2.

A classe **Floor** tem um atributo **capacity**. A definição do problema especificou que o usuário poderia querer situar a pessoa “no primeiro ou no segundo andar” – portanto, o objeto **Floor** precisa de um valor que distinga aquele objeto **Floor** como o primeiro ou segundo andar, de modo que incluímos o atributo **floorNumber**.

De acordo com a definição do problema, os botões **ElevatorButton** e **FloorButton** são “pressionados” por uma **Person**. Os botões também podem ser “desligados”. O estado de cada botão é ou “pressionado” ou “desligado”. Incluímos o atributo **pressed**, do tipo **boolean**, nas duas classes de botões. Quando **pressed** é **true**, o objeto botão está pressionado; quando **pressed** é **false**, o objeto botão está desligado. As classes **ElevatorDoor** e **FloorDoor** exibem características semelhantes. Os dois objetos podem estar “abertos” ou “fechados”, de modo que incluímos o atributo **open**, do tipo **boolean**, nas duas classes de portas. A classe **Light** também se enquadra nesta categoria – a luz está “iluminada” (ligada) ou “apagada”, de modo que incluímos o atributo **on**, do tipo **boolean**, na classe **Light**. Repare que, embora a definição do problema mencione que a campainha toca, não há nenhuma menção sobre quando a campainha “está tocando”, de modo que não incluímos um atributo **ring** separado para a classe **Bell**. À medida que progredirmos ao longo deste estudo de caso, continuaremos a adicionar, modificar e apagar informações sobre as classes em nosso sistema.

A Fig. 4.18 é um diagrama de classes que lista alguns dos atributos para cada classe em nosso sistema – as palavras e frases descritivas na Fig. 4.17 nos ajudam a gerar estes atributos. Note que a Fig. 4.18 não mostra associações entre objetos – mostramos estas associações na Fig. 3.23. No diagrama de casos da UML, os atributos de uma classe são colocados no compartimento do meio do retângulo da classe. Considere o seguinte atributo **open** da classe **ElevatorDoor**:

```
open : Boolean = false
```

Esta listagem contém três componentes de informação sobre o atributo. O nome do atributo é **open**. O tipo do atributo é **Boolean**². O tipo depende da linguagem usada para escrever o sistema de *software*. Em Java, por exemplo, o valor pode ser um tipo primitivo, como **boolean** ou **float**, bem como um tipo definido pelo usuário, como uma classe – começamos nosso estudo de classes no Capítulo 8, em que veremos que cada nova classe é um novo tipo de dado.

Também podemos indicar um valor inicial para cada atributo. O atributo **open** na classe **ElevatorDoor** tem um valor inicial **false**. Se um atributo em particular não tem um valor inicial especificado, somente seu nome e tipo (separados por dois pontos) são mostrados. Por exemplo, o atributo **ID** da classe **Person** é um inteiro – em Java, o atributo **ID** é do tipo **int**. Aqui não mostramos nenhum valor inicial, porque o valor deste atributo é um número que ainda não conhecemos; este número será determinado durante a execução por **ElevatorModel**. O atributo inteiro **currentFloor** para a classe **Person** também não está determinado antes da execução do programa – este atributo é determinado quando o usuário da simulação decide em que andar colocar a pessoa. Por enquanto, não nos preocupamos com os tipos ou valores iniciais dos atributos. Incluímos somente a informação que podemos vislumbrar facilmente na definição do problema.

Observe que a Fig. 4.18 não inclui atributos para a classe **ElevatorShaft**. Na verdade, a classe **ElevatorShaft** contém sete atributos que podemos determinar a partir do diagrama de classes da Fig. 3.23 – referências ao objeto **Elevator**, a dois objetos **FloorButton**, a dois objetos **FloorDoor** e a dois objetos **Light**. A classe **ElevatorModel** contém três atributos definidos pelo usuário – duas referências a objetos **Floor** e uma referência ao objeto **ElevatorShaft**. A classe **Elevator** também contém três atributos definidos pelo usuário

² Observe que os tipos de atributos na Fig. 4.18 estão na notação de UML. Associaremos os tipos de atributos **Boolean** e **Integer** no diagrama UML com os tipos de atributos **boolean** e **int** em Java, respectivamente. Descrevemos no Capítulo 3 que Java fornece uma “classe empacotadora de tipo” para cada tipo de dados primitivo. As classes empacotadoras de tipos têm a mesma notação que a notação UML para atributos de tipo; entretanto, quando implementamos nosso projeto em Java, a partir do Capítulo 8, usamos tipos de dados primitivos, para simplificar. Decidir sobre o uso de tipos de dados primitivos ou classes empacotadoras de tipos é um aspecto específico da implementação, que não deve ser mencionado na UML.

– referência ao objeto `ElevatorButton`, ao objeto `ElevatorDoor` e ao objeto `Bell`. Para economizar espaço, não mostraremos estes atributos adicionais em nossos diagramas de classes – vamos incluí-los, entretanto, no código nos apêndices.

O diagrama de classes da Fig. 4.18 oferece uma boa base para a estrutura de nosso modelo, mas o diagrama não está totalmente completo. Por exemplo, o atributo `currentFloor` na classe `Person` representa o andar no qual uma pessoa está localizada atualmente. Entretanto, em que andar está a pessoa quando aquela pessoa anda no elevador? Estes atributos ainda não representam suficientemente a estrutura do modelo. À medida que apresentamos mais da UML e do projeto orientado a objetos ao longo do Capítulo 22, continuaremos a fortalecer a estrutura de nosso modelo.

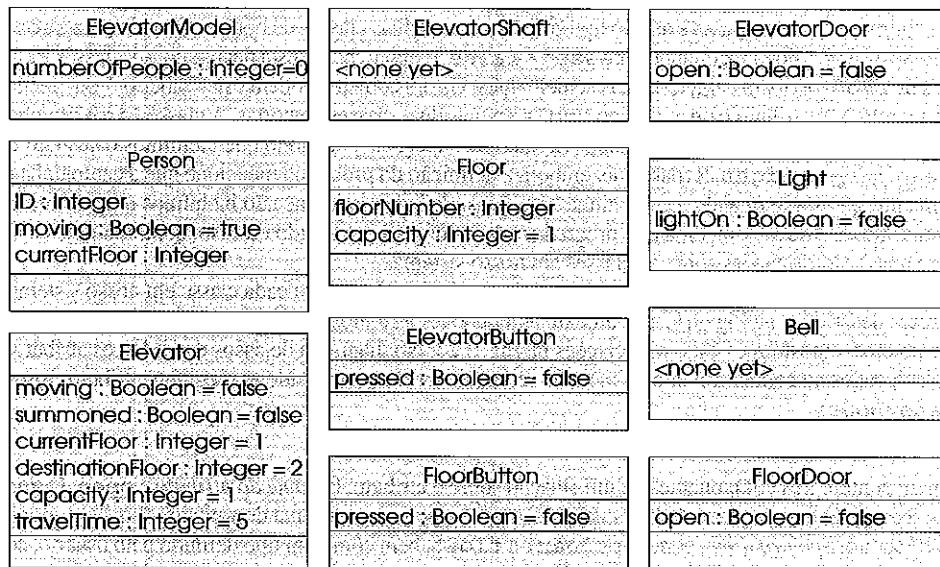


Fig. 4.18 Classes com atributos.

Resumo

- O procedimento para resolver um problema em termos das ações que serão executadas e a ordem em que essas ações devem ser executadas é chamado de algoritmo.
- Especificar a ordem em que as instruções serão executadas em um programa de computador chama-se controle do programa.
- O pseudocódigo ajuda o programador a “estudar” um programa antes de tentar escrevê-lo em uma linguagem de programação como Java.
- O refinamento passo a passo de cima para baixo é um processo para refinar o pseudocódigo mantendo uma representação completa do programa durante cada refinamento.
- As declarações são mensagens ao compilador dizendo os nomes e os atributos de variáveis e dizendo para reservar espaço para as variáveis.
- Utiliza-se uma estrutura de seleção para escolher entre cursos de ação alternativos.
- A estrutura de seleção `if` executa uma ação indicada somente quando a condição for verdadeira.
- A estrutura de seleção `if/else` especifica ações separadas que serão executadas quando a condição for verdadeira e quando for falsa.
- Sempre que mais de uma instrução precisa ser executada onde normalmente apenas uma única instrução é esperada, essas instruções devem ser incluídas entre chaves, formando um bloco. O bloco pode ser colocado em qualquer lugar em que uma instrução simples possa ser colocada.

- A instrução vazia, indicando que nenhuma ação será tomada, é indicada colocando-se um ponto-e-vírgula (`;`) onde normalmente haveria uma instrução.
- A estrutura de repetição específica que uma ação deve ser repetida enquanto alguma condição permanecer verdadeira.
- O formato para a estrutura de repetição `while` é

```
while ( condição )
    instrução
```

- O valor que contém uma parte fracionária é conhecido como número em ponto flutuante e é representado pelo tipo de dados `float` ou `double`.
- O operador unário de coerção (`double`) cria uma cópia em ponto flutuante temporária de seu operando.
- Java oferece os operadores de atribuição aritméticos `+=`, `-=`, `*=`, `/=` e `%=` que ajudam a abreviar certos tipos de expressões comuns.
- O operador de incremento, `++`, e o operador de decremento, `--`, incrementam ou decrementam uma variável por 1, respectivamente. Se o operador for prefixado à variável, a variável é incrementada ou decrementada por 1 primeiro e depois utilizada em sua expressão. Se o operador é pós-fixado à variável, a variável é utilizada em sua expressão e depois incrementada ou decrementada por 1.
- Os tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`) são os blocos de construção para tipos mais complexos em Java.
- Java exige que todas as variáveis tenham um tipo antes de poderem ser utilizadas em um programa. Por essa razão, Java é uma linguagem fortemente tipada.
- Os tipos primitivos em Java são portáveis entre todas as plataformas de computador que suportam Java.
- Java utiliza padrões reconhecidos internacionalmente para formatos de caractere (Unicode) e números em ponto flutuante (IEEE 754).
- As variáveis de instância dos tipos `char`, `byte`, `short`, `int`, `long`, `float` e `double` é atribuído o valor 0 por *default*. As variáveis do tipo `boolean` recebem o valor `false` por *default*.

Terminologia

<i>ação</i>	<i>laço infinito</i>
<i>algoritmo</i>	<i>modelo de ação/decisão</i>
<i>bloco</i>	<i>operador --</i>
<i>caracteres de espaço em branco</i>	<i>operador ?:</i>
<i>condição de continuação do laço</i>	<i>operador ++</i>
<i>conjunto de caracteres Unicode ISO</i>	<i>operador condicional (?:)</i>
<i>contador de laço</i>	<i>operador de coerção (tipo)</i>
<i>conversão implícita</i>	<i>operador de decreto (-)</i>
<i>corpo de um laço</i>	<i>operador de incremento (++)</i>
<i>decisão</i>	<i>operador de pós-decremento</i>
<i>divisão inteira</i>	<i>operador de pós-incremento</i>
<i>double</i>	<i>operador de pré-decremento</i>
<i>erro de lógica</i>	<i>operador de pré-incremento</i>
<i>erro de sintaxe</i>	<i>operador unário</i>
<i>estrutura de controle</i>	<i>operadores de atribuição aritméticos:</i>
<i>estrutura de repetição while</i>	<code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> e <code>%=</code>
<i>estrutura de seleção dupla</i>	<i>programação estruturada</i>
<i>estrutura de seleção if</i>	<i>promoção</i>
<i>estrutura de seleção if/else</i>	<i>pseudocódigo</i>
<i>estrutura de seleção única</i>	<i>refinamento passo a passo de cima para baixo</i>
<i>estruturas de controle aninhadas</i>	<i>repetição</i>
<i>estruturas de controle de entrada única/saída única</i>	<i>repetição controlada por contador</i>
<i>estruturas de controle empilhadas</i>	<i>repetição definida</i>
<i>estruturas de repetição</i>	<i>repetição indefinida</i>
<i>execução seqüencial</i>	<i>seleção</i>
<i>initialização</i>	<i>valor de sentinel</i>
<i>instrução vazia (;)</i>	

Exercícios de auto-revisão

- 4.1** Preencha os espaços em branco em cada uma das frases seguintes:
- Todos os programas podem ser escritos em termos de três tipos de estruturas de controle: _____, _____ e _____.
 - A estrutura de seleção _____ é utilizada para executar uma ação quando uma condição for verdadeira e outra ação quando essa condição for falsa.
 - A repetição de um conjunto de instruções um número específico de vezes chama-se repetição _____.
 - Quando não é conhecido com antecedência o número de vezes que um conjunto de instruções será repetido, pode-se usar o valor _____ para terminar a repetição.
- 4.2** Escreva quatro instruções Java diferentes que adicionam 1 à variável inteira **x**.
- 4.3** Escreva instruções Java para realizar cada uma das seguintes tarefas:
- Atribuir a soma de **x** e **y** a **z** e incrementar o valor de **x** por 1 depois do cálculo. Utilize somente uma instrução.
 - Testar se o valor da variável **count** é maior que 10. Se for, imprimir "Count is greater than 10".
 - Decrementar a variável **x** por 1, depois subtrair o resultado da variável **total**. Utilize somente uma instrução.
 - Calcular o resto após **q** ser dividido por **divisor** e atribuir o resultado a **q**. Escreva essa instrução de duas maneiras diferentes.
- 4.4** Escreva uma instrução Java para realizar cada uma das seguintes tarefas.
- Declarar que as variáveis **sum** e **x** são de tipo **int**.
 - Atribuir 1 à variável **x**.
 - Atribuir 0 à variável **sum**.
 - Adicionar a variável **x** à variável **sum** e atribua o resultado à variável **sum**.
 - Imprimir "The sum is: " e depois o valor da variável **sum**.
- 4.5** Combine as instruções que você escreveu no Exercício 4.4 em um aplicativo Java que calcula e imprime a soma dos inteiros de 1 a 10. Utilize a estrutura **while** para repetir as instruções de cálculo e incremento. O laço deve terminar quando o valor de **x** se tornar 11.
- 4.6** Determine o valor de cada variável depois que o cálculo é realizado. Suponha que, quando cada instrução começa a ser executada, todas as variáveis têm o valor inteiro 5.
- product *= x++;**
 - quotient /= ++x;**
- 4.7** Identifique e corrija os erros em cada um dos seguintes segmentos de código:
- while (c <= 5) {**
 product *= c;
 ++c;
 - if (gender == 1)**
 System.out.println("Woman");
else;
 System.out.println("Man");
- 4.8** O que há de errado com a seguinte estrutura de repetição **while**?
- ```
while (z >= 0)
 sum += z;
```

## **Respostas aos exercícios de auto-revisão**

- 4.1** a) Seqüência, seleção e repetição. b) **if/else**. c) controlado por contador ou definido. d) de sentinelas, de sinal, de *flag* ou fictício.
- 4.2**
- ```
x = x + 1;
x += 1;
++x;
x++;
```

- 4.3**
- a) `z = x++ + y;`
 - b) `if (count > 10)
 System.out.println("Count is greater than 10");`
 - c) `total -= --x;`
 - d) `q %= divisor;
 q = q % divisor;`
- 4.4**
- a) `int sum, x;`
 - b) `x = 1;`
 - c) `sum = 0;`
 - d) `sum += x; ou sum = sum + x;`
 - e) `System.out.println("The sum is: " + sum);`
- 4.5** O programa é o seguinte:

```

1 // Calcula a soma dos inteiros de 1 a 10
2 public class Calculate {
3     public static void main( String args[] )
4     {
5         int sum, x;
6
7         x = 1;
8         sum = 0;
9
10        while ( x <= 10 ) {
11            sum += x;
12            ++x;
13        }
14
15        System.out.println( "The sum is: " + sum );
16    }
17 }
```

- 4.6**
- a) `product = 25, x = 6;`
 - b) `quotient = 0, x = 6;`
- 4.7**
- a) Erro: está faltando a chave direita de fechamento do corpo da estrutura `while`.
Correção: adicionar a chave direita de fechamento depois da instrução `++c`;
 - b) Erro: ponto-e-vírgula depois de `else` resulta em um erro de lógica. A segunda instrução de saída sempre será executada.
Correção: remover o ponto-e-vírgula depois de `else`.
- 4.8** O valor da variável `z` nunca é alterado na estrutura `while`. Portanto, se a condição de continuação do laço (`z >= 0`) for verdadeira, cria-se um laço infinito. Para evitar a ocorrência do laço infinito, `z` deve ser decrementado de modo que acabe se tornando menor que 0.

Exercícios

- 4.9** Identifique e corrija os erros em cada uma das instruções seguintes. [Nota: pode haver mais de um erro em cada trecho de código]:

```

a) if ( age >= 65 );
    System.out.println( "Age greater than or equal to 65" );
else
    System.out.println( "Age is less than 65" );
b) int x = 1, total;
    while ( x <= 10 ) {
        total += x;
        ++x;
    }
```

```

c) While ( x <= 100 )
    total += x;
    ++x;
d) while ( y > 0 ) {
    System.out.println( y );
    ++y;

```

4.10 O que o programa seguinte imprime?

```

1 public class Mystery {
2
3     public static void main( String args[] )
4     {
5         int y, x = 1, total = 0;
6
7         while ( x <= 10 ) {
8             y = x * x;
9             System.out.println( y );
10            total += y;
11            ++x;
12        }
13    }
14    System.out.println( "Total is " + total );
15 }
16 }
```

Para os Exercícios de 4.11 a 4.14, execute os seguintes passos:

- Leia a definição do problema;
- Formule o algoritmo utilizando pseudocódigo e refinamento passo a passo de cima para baixo;
- Escreva um programa Java;
- Teste, depure e execute o programa Java;
- Processe três conjuntos completos de dados.

4.11 Os motoristas se preocupam com o consumo de combustível dos seus automóveis. Um motorista monitorou vários tanques cheios de gasolina registrando a quilometragem dirigida e a quantidade de combustível em litros utilizado para cada tanque cheio. Desenvolva um aplicativo Java que receba como entrada os quilômetros dirigidos e os litros de gasolina consumidos (ambos como inteiros) para cada tanque cheio. O programa deve calcular e exibir o consumo em quilômetros/litro para cada tanque cheio e imprimir a quilometragem combinada e a soma total de litros de combustível consumidos até esse ponto. Todos os cálculos de médias devem produzir resultados em ponto flutuante. Utilize diálogos de entrada para obter os dados do usuário.

4.12 Desenvolva um aplicativo Java que determine se um cliente de uma loja de departamentos excedeu o limite de crédito em uma conta corrente. Para cada cliente, os seguintes fatos estão disponíveis:

- Número da conta;
- Saldo no início do mês;
- Total de todos os itens comprados a crédito por este cliente neste mês;
- Total de todos os créditos lançados na conta do cliente neste mês; e
- Límite autorizado de crédito.

O programa deve ler cada um desses fatos como inteiros a partir de diálogos de entrada, calcular o novo saldo ($= \text{saldo inicial} + \text{compras a crédito} - \text{créditos}$), exibir o novo saldo e determinar se ele excede o limite de crédito do cliente. Para aqueles clientes cujo limite de crédito for excedido, o programa deve exibir a mensagem “Limite de crédito excedido”.

4.13 Uma grande empresa paga seu pessoal de vendas com base em comissões. O pessoal de vendas recebe R\$ 200,00 por semana mais 9% de suas vendas brutas durante essa semana. Por exemplo, o vendedor que realiza um total de vendas de R\$ 5000,00 de mercadoria em uma semana, recebe R\$ 200,00 mais 9% de R\$ 5000,00, ou um total de R\$ 650,00. Foi-lhe fornecida uma lista de itens vendidos para cada vendedor. Os valores desses itens são os seguintes:

Item	Valor
1	239,99

```

2      129,75
3      99,95
4      350,89

```

Desenvolva um aplicativo Java que receba como entrada o número de itens vendidos por um vendedor durante a última semana e calcule e exiba os rendimentos do vendedor. Não há nenhum limite para o número de itens vendidos por um vendedor.

4.14 Desenvolva um aplicativo Java que determina o salário bruto de cada um de três empregados. A empresa paga “hora normal” pelas primeiras 40 horas trabalhadas por cada empregado e “horas extras” com 50% de acréscimo para todas as horas trabalhadas além de 40 horas. Você recebe uma lista dos empregados da empresa, o número de horas trabalhadas por cada empregado na última semana e o salário-hora de cada empregado. Seu programa deve ler essas informações para cada empregado e deve determinar e exibir o salário bruto do empregado. Utilize diálogos para entrada dos dados.

4.15 O processo de localizar o valor maior (isto é, o valor máximo de um grupo de valores) é freqüentemente utilizado em aplicativos de computador. Por exemplo, um programa que determina o vencedor de uma competição de vendas leia o número de unidades vendidas por cada vendedor. O vendedor que vende mais unidades ganha a competição. Escreva um programa em pseudocódigo e depois um aplicativo em Java que receba como entrada uma série de 10 números de um único dígito como caracteres e determine e imprime o maior dos números. *Dica:* o programa deve utilizar três variáveis como segue:

- a) **counter**: um contador para contar até 10 (isto é, monitorar quantos números foram lidos e determinar quando todos os 10 números foram processados)
- b) **number**: o dígito atual lido pelo programa
- c) **largest**: o maior número encontrado até agora.

4.16 Escreva um aplicativo em Java que utiliza um laço para imprimir a seguinte tabela de valores:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

4.17 Utilizando uma abordagem semelhante à do Exercício 4.15, descubra os *dois* maiores valores entre 10 dígitos lidos. [Nota: você só pode ler o número uma vez.]

4.18 Modifique o programa na Fig. 4.11 para validar suas entradas. Para qualquer entrada, se o valor lido for diferente de 1 ou 2, continue repetindo o laço até o usuário digitar um valor correto.

4.19 O que imprime o seguinte programa?

```

1 public class Mystery2 {
2
3     public static void main( String args[] )
4     {
5         int count = 1;
6
7         while ( count <= 10 ) {
8             System.out.println(
9                 count % 2 == 1 ? "*****" : "+++++++" );
10            ++count;
11        }
12    }
13 }

```

4.20 O que imprime o seguinte programa?

```

1 public class Mystery3 {
2
3     public static void main( String args[] )
4     {
5         int row = 10, column;
6
7         while ( row >= 1 ) {
8             column = 1;
9
10            while ( column <= 10 ) {
11                System.out.print( row % 2 == 1 ? "<" : ">" );
12                ++column;
13            }
14        }
15        --row;
16        System.out.println();
17    }
18 }
19 }
```

4.21 (*Problema do else oscilante*) Determine a saída para cada uma das instruções seguintes quando *x* for 9 e *y* for 11 e quando *x* for 11 e *y* for 9. Observe que o compilador ignora os recuos em um programa Java. Da mesma forma, o compilador Java sempre associa um *else* com o *if* anterior, a menos que seja instruído a fazer de outro modo pela colocação de chaves ({}). À primeira vista, o programador não pode ter certeza do *if* a que um *else* corresponde; esta situação é conhecida como o problema do “else oscilante”. Eliminamos o recuo do código seguinte para tornar o problema mais desafiador. [Dica: aplique convenções de recuo que você aprendeu.]

- a) *if* (*x* < 10)
 if (*y* > 10)
 System.out.println("*****");
 else
 System.out.println("#####");
 System.out.println("\$\$\$\$\$");
- b) *if* (*x* < 10)
 if (*y* > 10)
 System.out.println("*****");
 }
 else
 System.out.println("#####");
 System.out.println("\$\$\$\$\$");
 }

4.22 (*Outro problema do else oscilante*) Modifique o código dado para produzir a saída mostrada em cada parte do problema. Utilize técnicas adequadas de recuo. Você não pode fazer nenhuma alteração diferente da inserção de chaves e da alteração dos recuos do código. O compilador ignora recuos em um programa Java. Eliminamos o recuo do código dado para tornar o problema mais desafiador. [Nota: é possível que não seja necessária nenhuma modificação para alguma das partes.]

```

if ( y == 8 )
if ( x == 5 )
System.out.println( "@@@@@" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
System.out.println( "&&&&&" );
```

- a) Pressupondo-se que $x = 5$ e $y = 8$, a seguinte saída é produzida:

```
@@@@@  
$$$$$  
&&&&
```

- b) Pressupondo-se que $x = 5$ e $y = 8$, a seguinte saída é produzida:

```
@@@@@
```

- c) Pressupondo-se que $x = 5$ e $y = 8$, a seguinte saída é produzida:

```
@@@@@  
&&&&
```

- d) Pressupondo-se que $x = 5$ e $y = 7$, a seguinte saída é produzida: [Nota: as últimas três instruções de saída depois do `else` fazem parte de uma instrução composta.]

```
#####  
$$$$$  
&&&&
```

4.23 Escreva um *applet* que lê o tamanho do lado de um quadrado e exibe um quadrado vazio desse tamanho com asteriscos, utilizando o método `drawString` dentro do método `paint` de seu *applet*. Utilize um diálogo de entrada para ler o tamanho fornecido pelo usuário. O programa deve trabalhar com quadrados de todos os tamanhos de lado possíveis, entre 1 e 20.

4.24 O palíndromo é um número ou uma frase de texto que é lido da mesma forma tanto da esquerda para a direita como da direita para a esquerda. Por exemplo, cada um dos inteiros de cinco dígitos seguintes é um palíndromo: 12321, 55555, 45554 e 11611. Escreva um aplicativo que lê um inteiro dos cinco dígitos e determina se ele é ou não um palíndromo. Se o número não for de cinco dígitos, exiba um diálogo de mensagem de erro para indicar o problema ao usuário. Quando o usuário fechar o diálogo de erro, permita que o usuário digite um novo valor.

4.25 Escreva um aplicativo que recebe como entrada um inteiro que contém só 0s e 1s (isto é, um inteiro “binário”) e imprime seu equivalente em decimal. [Dica: utilize os operadores de módulo e divisão para pegar os dígitos do “número binário” um de cada vez, da direita para a esquerda. Assim como ocorre no sistema de números decimais, no qual o dígito mais à direita tem um valor posicional de 1 e o próximo dígito à esquerda tem um valor posicional de 10, depois 100, depois 1000 etc., no sistema de números binários o dígito mais à direita tem um valor posicional de 1, o próximo dígito à esquerda tem um valor posicional de 2, depois 4, depois 8, etc. Portanto, o número decimal 234 pode ser interpretado como $4 * 1 + 3 * 10 + 2 * 100$. O equivalente decimal do binário 1101 é $1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$ ou $1 + 0 + 4 + 8$ ou 13.]

4.26 Escreva um aplicativo que exibe o seguinte padrão de tabuleiro de xadrez:

```
* * * * * * * *  
* * * * * * * *  
* * * * * * * *  
* * * * * * * *  
* * * * * * * *  
* * * * * * * *  
* * * * * * * *
```

O programa pode utilizar só três instruções de saída, uma na forma

```
System.out.print( "*" );
```

uma na forma

```
System.out.print( " " );
```

e uma na forma

```
System.out.println();
```

Observe que a instrução precedente indica que o programa deve dar saída a um único caractere de nova linha para avançar para a próxima linha na saída. [Dica: são necessárias estruturas de repetição nesse exercício.]

4.27 Escreva um aplicativo que exibe continuamente na janela de comando os múltiplos do inteiro 2, a saber 2, 4, 8, 16, 32, 64 etc. O laço não deve terminar (isto é, você deve criar um laço infinito). O que acontece quando você executa esse programa?

4.28 O que há de errado na seguinte instrução? Escreva a instrução correta para adicionar um à soma de **x** e **y**.

```
System.out.println( ++(x + y) );
```

4.29 Escreva um aplicativo que lê três valores diferentes de zero digitados pelo usuário em diálogos de entrada, determina se eles poderiam representar os lados de um triângulo e imprime a resposta.

4.30 Escreva um aplicativo que lê três inteiros diferentes de zero, determina se eles poderiam ser os lados de um triângulo retângulo e imprime a resposta.

4.31 Uma empresa quer transmitir dados por telefone, mas está preocupada com a possibilidade de seus telefones estarem grampeados. Todos seus dados são transmitidos como inteiros de quatro dígitos. Eles pedem para você escrever um programa que criptografa os dados para que possam ser transmitidos com mais segurança. O aplicativo deve ler um inteiro de quatro dígitos digitado pelo usuário em um diálogo de entrada e criptografá-lo como segue: substitua cada dígito por (*a soma desse dígito mais 7*) módulo 10. Depois troque o primeiro dígito pelo terceiro e troque o segundo dígito pelo quarto. A seguir, imprima o inteiro criptografado. Escreva um aplicativo separado que recebe como entrada um inteiro de quatro dígitos criptografado e o decifra para formar o número original.

4.32 O fatorial de um inteiro não-negativo *n* é escrito como *n!* (“pronuncia-se fatorial de *n*”) e é definido como segue:
 $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$ (para valores de *n* maiores do que ou iguais a 1)

e

$n! = 1$ (para *n* = 0).

Por exemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, o que dá 120.

- Escreva um aplicativo que lê um inteiro não-negativo de um diálogo de entrada, calcula e imprime o fatorial.
- Escreva um aplicativo que estima o valor da constante matemática *e* utilizando a fórmula

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

- Escreva um aplicativo que calcula o valor de e^x utilizando a fórmula:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

5

Estruturas de controle: parte 2

Objetivos

- Ser capaz de utilizar as estruturas de repetição **for** e **do/while** para executar instruções em um programa repetidamente.
- Compreender a seleção múltipla utilizando a estrutura de seleção **switch**.
- Ser capaz de utilizar as instruções de controle de programa **break** e **continue**.
- Ser capaz de utilizar operadores lógicos.

Quem pode controlar seu destino?

William Shakespeare, *Otelo*

A chave utilizada é sempre brilhante.

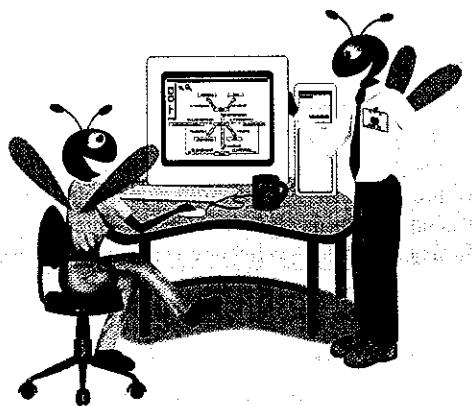
Benjamin Franklin

O homem é um animal ferramenteiro.

Benjamin Franklin

*A inteligência... é a capacidade de criar objetos artificiais,
especialmente ferramentas para fazer ferramentas.*

Henri Bergson



Sumário do capítulo

- 5.1 Introdução
- 5.2 Princípios básicos da repetição controlada por contador
- 5.3 A estrutura de repetição `for`
- 5.4 Exemplos com a estrutura `for`
- 5.5 A estrutura de seleção múltipla `switch`
- 5.6 A estrutura de repetição `do/while`
- 5.7 As instruções `break` e `continue`
- 5.8 As instruções rotuladas `break` e `continue`
- 5.9 Operadores lógicos
- 5.10 Resumo de programação estruturada
- 5.11 (Estudo de caso opcional) Pensando em objetos: identificando estados e atividades dos objetos

[Resumo](#) • [Terminologia](#) • [Exercícios de auto-revisão](#) • [Respostas aos exercícios de auto-revisão](#) • [Exercícios](#)

5.1 Introdução

O Capítulo 4 iniciou nossa introdução aos tipos de blocos de construção que estão disponíveis para a solução de problemas e os utilizamos para empregar princípios comprovados de construção de programas. Neste capítulo, continuamos nossa apresentação da teoria e dos princípios da programação estruturada, apresentando as estruturas de controle de Java restantes. Como no Capítulo 4, as técnicas de Java que você aprende aqui são aplicáveis à maioria das linguagens de alto nível. Quando iniciarmos nosso tratamento formal de programação baseada em objetos em Java no Capítulo 8, veremos que as estruturas de controle que estudamos neste capítulo e no Capítulo 4 são úteis na construção e na manipulação de objetos.

5.2 Princípios básicos da repetição controlada por contador

A repetição controlada por contador exige o seguinte:

1. o *nome* de uma variável de controle (ou contador de laço);
2. o *valor inicial* da variável de controle;
3. o *incremento* (ou o *decremento*) pelo qual a variável de controle é modificada a cada passagem pelo laço (também conhecido como *cada iteração do laço*); e
4. a condição que testa o *valor final* da variável de controle (isto é, se o laço deve continuar).

Para ver os quatro elementos da repetição controlada por contador, considere o *applet* simples mostrado na Fig. 5.1, que desenha 10 linhas com o método `paint` do *applet*. Lembre-se de que um *applet* exige que um documento separado de HTML carregue o *applet* no `appletviewer` ou em um navegador. Para os propósitos desse *applet*, a marca `<applet>` especifica uma largura de 275 pixels e uma altura de 110 pixels.

O método `paint` do *applet* (que o contêiner de *applets* chama quando ele executa o *applet*) opera assim: a declaração na linha 18 dá um nome à variável de controle (`counter`), declara que ela é um inteiro, reserva espaço para ela na memória e a configura com *valor inicial* de 1. As declarações que incluem a inicialização são, com efeito, instruções executáveis. A declaração e a inicialização de `counter` também poderiam ter sido realizadas com a declaração e a instrução

```
int counter; // declara o contador
counter = 1; // inicializa o contador com 1
```

```

1 // Fig. 5.1: Whilecounter.java
2 // Repetição controlada por contador
3
4 // Pacotes do núcleo de Java
5 import java.awt.Graphics;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JApplet;
9
10 public class WhileCounter extends JApplet {
11
12     // Desenha linhas no fundo do applet
13     public void paint( Graphics g )
14     {
15         // Chama versão herdada do método paint
16         super.paint( g );
17
18         int counter = 1;           // inicialização
19
20         while ( counter <= 10 ) { // condição de repetição
21             g.drawLine( 10, 10, 250, counter * 10 );
22             ++counter;           // incremento
23
24         } // fim da estrutura while
25
26     } // fim do método paint
27
28 } // fim da classe WhileCounter

```

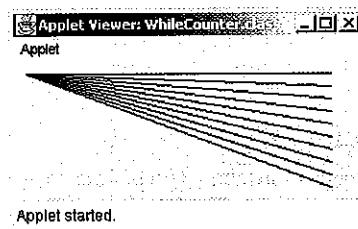


Fig. 5.1 Repetição controlada por contador.

A declaração não é executável, mas a instrução de atribuição o é. Utilizamos os dois métodos de inicialização de variáveis em todo este livro.

A linha 21 na estrutura `while` utiliza a referência `Graphics g`, que faz referência ao objeto `Graphics` do *applet*, para enviar a mensagem `drawLine` para o objeto `Graphics`, pedindo-lhe para desenhar uma linha. Lembre-se de que “enviar uma mensagem para um objeto” significa, na verdade, chamar um método para realizar uma tarefa. Um dos muitos serviços do objeto `Graphics` é desenhar linhas. Nos capítulos anteriores, também vimos que entre os demais serviços do objeto `Graphics` está incluído desenhar retângulos, *strings* e elipses. O método `drawLine` de `Graphics` exige quatro argumentos, que representam a primeira coordenada *x*, a primeira coordenada *y*, a segunda coordenada *x* e a segunda coordenada *y* da linha. Nesse exemplo, a segunda coordenada *y* muda de valor durante cada iteração do laço com o cálculo `counter * 10`. Esta alteração faz com que o segundo ponto em cada chamada para `drawLine` move-se 10 pixels para baixo na área de exibição do *applet*.

A linha 22 na estrutura `while` *incrementa* a variável de controle por 1 a cada iteração do laço. A condição de continuação do laço na estrutura `while` testa se o valor da variável de controle é menor ou igual a 10 (o *valor final* para o qual a condição é `true`). Observe que o programa executa o corpo desse `while` mesmo quando a variável de controle é 10. O laço termina quando a variável de controle excede 10 (isto é, `counter` torna-se 11).

O programa na Fig. 5.1 pode ficar mais conciso inicializando-se `counter` como 0 e pré-incrementando-se `counter` na condição da estrutura `while`, como segue:

```
while ( ++counter <= 10 )      // condição de repetição
    g.drawLine( 10, 10, 250, counter * 10 );
```

Esse código economiza uma instrução (e elimina a necessidade de chaves em torno do corpo do laço) porque a condição `while` executa o incremento antes de testar a condição (lembre-se de que a precedência de `++` é mais alta que aquela de `<=`). Codificar dessa maneira condensada exige prática.



Boa prática de programação 5.1

Os programas devem controlar a contagem dos laços com valores inteiros.



Erro comum de programação 5.1

Como os valores em ponto flutuante podem ser aproximados, controlar a contagem de laços com variáveis em ponto flutuante pode resultar em valores imprecisos do contador e testes de terminação não-exatos.



Boa prática de programação 5.2

Recue as instruções no corpo de cada estrutura de controle.



Boa prática de programação 5.3

Coloque uma linha em branco antes e depois de cada estrutura de controle importante para destacá-la no programa.



Boa prática de programação 5.4

Muitos níveis de aninhamento podem tornar um programa difícil de entender. Como regra geral, tente evitar a utilização de mais de três níveis de aninhamento.



Boa prática de programação 5.5

O espaçamento vertical acima e abaixo de estruturas de controle e o recuo do corpo das estruturas de controle dentro dos cabeçalhos dessas estruturas conferem aos programas uma aparência bidimensional que melhora a legibilidade.

5.3 A estrutura de repetição `for`

A estrutura de repetição `for` cuida de todos os detalhes da repetição controlada por contador. Para ilustrar a capacidade da estrutura `for`, vamos rescrever o *applet* da Fig. 5.1. O resultado é mostrado na Fig. 5.2. Lembre-se de que esse programa exige um documento separado de HTML para carregar o *applet* no *appletviewer*. Para o propósito desse *applet*, a marca `<applet>` especifica uma largura de 275 pixels e uma altura de 110 pixels.

O método `paint` do *applet* opera da seguinte forma: quando a estrutura `for` (linhas 20 e 21) começa a ser executada, a variável de controle `counter` é inicializada com 1 (os primeiros dois elementos de repetição controlada por contador e o *nome* da variável de controle e o seu *valor inicial*). Em seguida, o programa verifica a condição de continuação do laço, `counter <= 10`. A condição contém o *valor final* (10) da variável de controle. Como o valor inicial de `counter` é 1, a condição é satisfeita (`true`), de modo que a instrução do corpo (linha 21) desenha uma linha. Depois de executar o corpo do laço, o programa incrementa a variável `counter` na expressão `counter++`. Assim, o programa executa o teste de continuação do laço novamente, para determinar se o programa deve continuar com a próxima iteração do laço ou se ele deve terminar o laço. Neste ponto, o valor da variável de controle é igual a 2, de modo que a condição é verdadeira (i.e., o valor final não é excedido) e, portanto, o programa executa a instrução do corpo novamente (i.e., a próxima iteração do laço). Esse processo continua até que o valor de `counter` se torne 11, fazendo com que o teste de continuação do laço falle e a repetição termine. Então, o programa executa a primeira instrução depois da estrutura `for` (neste caso, o método `paint` termina porque o programa alcança o final de `paint`).

Repare que a Fig. 5.2 utiliza a condição de continuação do laço `counter <= 10`. Se o programador especificasse incorretamente `counter < 10` como condição, o laço seria executado apenas nove vezes. Esse engano é um erro comum de lógica chamado de *erro por um (off-by-one)*.



Erro comum de programação 5.2

Utilizar um operador relacional incorreto ou utilizar um valor final incorreto de um contador de laço na condição de uma estrutura while, for ou do/while pode causar um erro por um.

```

1 // Fig. 5.2: Forcounter.java
2 // Repetição controlada por contador com a estrutura for
3
4 // Pacotes do núcleo de Java
5 import java.awt.Graphics;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JApplet;
9
10 public class ForCounter extends JApplet {
11
12     // Desenha linhas no fundo do applet
13     public void paint( Graphics g )
14     {
15         // Chama versão herdada do método paint
16         super.paint( g );
17
18         // Inicialização, condição de repetição e incremento
19         // estão incluídos no cabeçalho da estrutura for.
20         for ( int counter = 1; counter <= 10; counter++ )
21             g.drawLine( 10, 10, 250, counter * 10 );
22
23     } // fim do método paint
24
25 } // fim da classe ForCounter

```

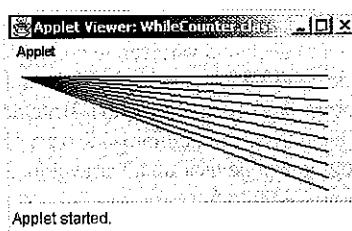


Fig. 5.2 Repetição controlada por contador com a estrutura **for**.



Boa prática de programação 5.6

Utilizar o valor final na condição de uma estrutura **while** ou **for** e utilizar o operador `<=` relacional ajudará a evitar erros por um. Para um laço que imprime os valores 1 a 10, a condição de continuação do laço deve ser `counter <= 10` em vez de `counter < 10` (que causa um erro por um) ou `counter < 11` (o que é correto). Muitos programadores preferem a chamada contagem baseada em zero, em que, para contar 10 vezes, `counter` seria inicializado com zero e o teste de continuação do laço seria `counter < 10`.

A Fig. 5.3 dá uma olhada mais de perto na estrutura **for** da Fig. 5.2. A primeira linha da estrutura **for** (incluindo a palavra-chave **for** e tudo o que aparece entre parênteses após **for**) é algumas vezes chamada de *cabeçalho da estrutura for*. Repare que a estrutura **for** “faz tudo”: ela especifica cada um dos itens necessários para a repetição controlada por contador com uma variável de controle. Se existe mais de uma instrução no corpo das estruturas **for**, é obrigatório o uso de chaves (`{` e `}`) para definir o corpo do laço.

O formato geral da estrutura **for** é

```
for ( expressão1; expressão2; expressão3 )
    instrução
```

onde *expressão1* dá nome à variável de controle do laço e fornece seu valor inicial, *expressão2* é a condição de continuação do laço (contendo o valor final da variável de controle) e *expressão3* modifica o valor da variável de controle para que a condição de continuação do laço, em algum momento, se torne falsa. Na maioria dos casos, a estru-

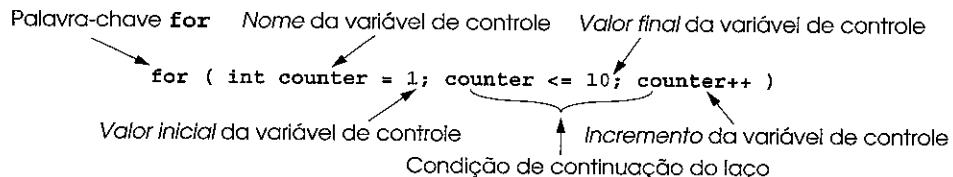


Fig. 5.3 Componentes de um cabeçalho de estrutura **for** típico.

Estrutura **for** pode ser representada por uma estrutura **while** equivalente, com *expressão1*, *expressão2* e *expressão3* colocados como segue:

```
expressão1;
while ( expressão2 ) {
    instrução
    expressão3;
}
```

Na Seção 5.7, mostramos um caso no qual a estrutura **for** não pode ser representada com uma estrutura **while** equivalente.

Se *expressão1* (a seção de inicialização) declara a variável de controle dentro dos parênteses do cabeçalho da estrutura **for** (isto é, o tipo da variável de controle é especificado antes do nome da variável), a variável de controle só pode ser utilizada no corpo da estrutura **for**. Essa utilização restrita do nome da variável de controle é conhecida como *escopo* da variável. O escopo de uma variável define onde o programa pode usar a variável. Por exemplo, mencionamos anteriormente que um programa pode usar uma variável local somente no método que declara a variável. O escopo é discutido em detalhes no Capítulo 6.



Erro comum de programação 5.3

Quando a variável de controle de uma estrutura **for** é definida inicialmente na seção de inicialização do cabeçalho da estrutura **for**, é um erro de sintaxe utilizar a variável de controle depois do corpo da estrutura.

Às vezes, *expressão1* e *expressão3* em uma estrutura **for** são listas de expressões separadas por vírgulas que permitem que o programador utilize múltiplas expressões de inicialização e/ou múltiplas expressões de incremento. Por exemplo, pode haver diversas variáveis de controle em uma única estrutura **for** que devem ser inicializadas e incrementadas.



Boa prática de programação 5.7

Coloque apenas expressões que envolvem as variáveis de controle nas seções de inicialização e incremento de uma estrutura **for**. As manipulações de outras variáveis devem aparecer antes do laço (se devem ser executadas somente uma vez, como instruções de inicialização) ou no corpo do laço (se devem ser executadas uma vez a cada iteração do laço, como instruções de incremento ou decremento).

As três expressões na estrutura **for** são opcionais. Se *expressão2* for omitida, Java assume que a condição de continuação do laço seja **true**, criando, assim, um laço infinito. Pode-se omitir *expressão1* se o programa inicializar a variável de controle antes do laço. Pode-se omitir a *expressão3* se o programa calcular o incremento com instruções no corpo do laço ou se o laço não exigir um incremento. A expressão de incremento na estrutura **for** atua como instrução independente (*stand-alone*) no fim do corpo da estrutura **for**, de modo que as expressões

```
counter = counter + 1
counter += 1
++counter
counter++
```

são equivalentes na parte de incremento da estrutura **for**. Muitos programadores preferem a forma **counter++**, porque uma estrutura **for** incrementa sua variável de controle depois que o corpo do laço é executado, e colocar **++** após o nome da variável incrementa a variável depois que o programa usa o seu valor. Portanto, a forma de pós-incremento parece mais natural. Pré-incrementar e pós-incrementar têm o mesmo efeito na expressão de incremento, porque o incremento não aparece em uma expressão maior. Os dois ponto-e-vírgulas na estrutura **for** são obrigatórios.

Erro comum de programação 5.4



*Utilizar vírgulas em vez dos dois ponto-e-vírgulas obrigatórios em um cabeçalho **for** é um erro de sintaxe.*

Erro comum de programação 5.5



*Colocar um ponto-e-vírgula imediatamente à direita do parêntese direito de um cabeçalho **for** torna o corpo dessa estrutura **for** uma instrução vazia. Normalmente, esse é um erro de lógica.*

As partes de inicialização, de condição de continuação do laço e de incremento de uma estrutura **for** podem conter expressões aritméticas. Por exemplo, suponha que **x = 2** e **y = 10**. Se **x** e **y** não são modificadas no corpo do laço, a instrução

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

é equivalente à instrução

```
for ( int j = 2; j <= 80; j += 5 )
```

O “incremento” de uma estrutura **for** pode ser negativo, caso em que ele é realmente um decremento e o laço na realidade conta para baixo.

Se a condição de continuação do laço for inicialmente **false**, o programa não executa o corpo da estrutura **for** não é executado. Em vez disso, a execução prossegue com a instrução seguinte à estrutura **for**.

Freqüentemente, os programas exibem o valor da variável de controle ou o usam em cálculos no corpo do laço. No entanto, esta prática não é obrigatória. É comum utilizar a variável de controle para controlar a repetição sem nunca mencioná-la no corpo da estrutura **for**.

Dica de teste e depuração 5.1



*Embora o valor da variável de controle possa ser alterado no corpo de um laço **for**, evite fazer isso porque essa prática pode levar a erros sutis.*

Representamos a estrutura **for** em um fluxograma de maneira muito semelhante à estrutura **while**. Por exemplo, o fluxograma da instrução **for**

```
for ( int counter = 1; counter <= 10; counter++ )
    g.drawLine( 10, 10, 250, counter * 10 );
```

é mostrado na Fig. 5.4. Esse fluxograma torna claro que a inicialização ocorre apenas uma vez e que o incremento ocorre a cada iteração *depois* que o programa executa a instrução do corpo. Observe que (além de pequenos círculos e setas) o fluxograma contém somente símbolos de ação e um símbolo de decisão. O programador preenche os retângulos e losangos com ações e decisões apropriadas para o algoritmo.

5.4 Exemplos com a estrutura **for**

Os exemplos mostrados a seguir mostram métodos de variação da variável de controle em uma estrutura **for**. Em cada caso, escrevemos o cabeçalho **for** apropriado. Observe a alteração no operador relacional para laços que decrementam a variável de controle.

- a) Faça a variável de controle variar de 1 a 100 em incrementos de 1.

```
for ( int i = 1; i <= 100; i++ )
```

- b) Faça a variável de controle variar de 100 a 1 em incrementos de -1 (i.e., decrementos de 1).

```
for ( int i = 100; i >= 1; i-- )
```

c) Faça a variável de controle variar de 7 a 77 em incrementos de 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Faça a variável de controle variar de 20 a 2 em incrementos de -2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Faça a variável de controle assumir a seguinte seqüência de valores: 2, 5, 8, 11, 14, 17, 20.

```
for ( int j = 2; j <= 20; j += 3 )
```

f) Faça a variável de controle assumir a seguinte seqüência de valores: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int j = 99; j >= 0; j -= 11 )
```

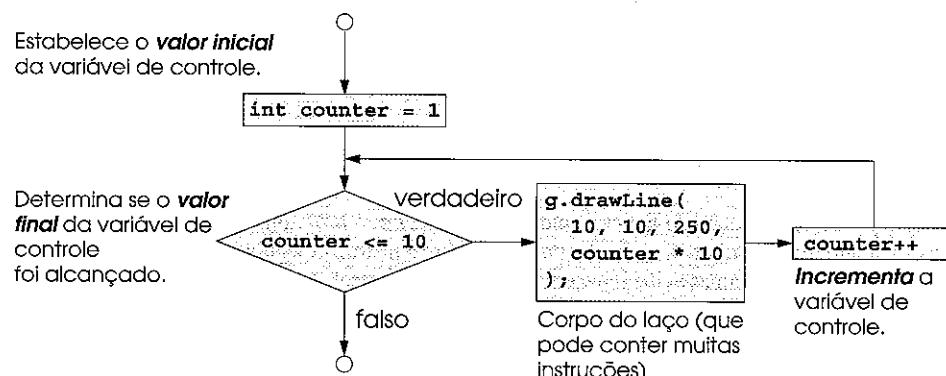


Fig. 5.4 Fluxograma de uma estrutura de repetição **for** típica.



Erro comum de programação 5.6

Não utilizar o operador relacional adequado na condição de continuação do laço de um laço que conta para baixo (como utilizar `i <= 1` em um laço que conta para baixo até 1) é normalmente um erro de lógica e produzirá resultados incorretos quando o programa for executado.

Os próximos dois programas de exemplo demonstram aplicações simples da estrutura de repetição **for**. O aplicativo da Fig. 5.5 utiliza a estrutura **for** para somar todos os inteiros pares de 2 a 100. Lembre-se de que o interpretador **java** é utilizado para executar um aplicativo a partir da janela de comando.

Observe que o corpo da estrutura **for** na Fig. 5.5 realmente poderia ser mesclado na parte mais à direita do cabeçalho **for** utilizando uma vírgula, como segue:

```
for ( int number = 2;
      number <= 100;
      sum += number, number += 2)
; // instrução vazia
```

De maneira semelhante, a inicialização **sum = 0** poderia ser mesclada na seção de inicialização da estrutura **for**.



Boa prática de programação 5.8

*Embora as instruções que precedem uma estrutura **for** e as instruções no corpo de uma estrutura **for** possam frequentemente ser mescladas no cabeçalho da estrutura **for**, evite fazer isso porque torna o programa mais difícil de ler.*

```

1 // Fig. 5.5: Soma.java
2 // Repetição controlada por contador com a estrutura for
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 public class Sum {
8
9     // método main começa a execução do aplicativo em Java
10    public static void main( String args[] )
11    {
12        int sum = 0;
13
14        // soma números inteiros de 2 a 100
15        for ( int number = 2; number <= 100; number += 2 )
16            sum += number;
17
18        // exibe resultados
19        JOptionPane.showMessageDialog( null, "The sum is " + sum,
20                                     "Sum Even Integers from 2 to 100",
21                                     JOptionPane.INFORMATION_MESSAGE );
22
23        System.exit( 0 );    // termina o aplicativo
24
25    } // fim do método main
26
27 } // fim da classe Sum

```

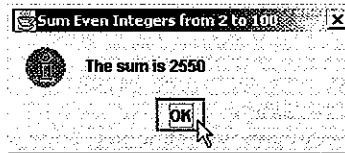


Fig. 5.5 Soma com a estrutura `for`.



Boa prática de programação 5.9

Limite o tamanho dos cabeçalhos de estruturas de controle a uma única linha, se possível.

O próximo exemplo utiliza a estrutura `for` para calcular juros compostos. Considere o seguinte problema:

Uma pessoa investe US\$1000,00 em uma conta de poupança que rende 5% de juros ao ano. Assumindo que todo o juro é deixado em depósito, calcule e imprima a quantidade de dinheiro na conta no final de cada ano por 10 anos. Utilize a seguinte fórmula para determinar essas quantidades:

$$a = p (1 + r)^n$$

onde

p é a quantidade original investida (isto é, o principal)

r é a taxa de juros anual

n é o número de anos

a é a quantidade em depósito no final do *n*-ésimo ano.

Esse problema envolve um laço que executa o cálculo indicado para cada um dos 10 anos que o dinheiro permanece em depósito. A solução é o aplicativo mostrado na Fig. 5.6.

A linha 17 no método `main` declara três variáveis `double` e inicializa duas delas – `principal` com `1000.0` e `rate` com `.05`. Java trata constantes em ponto flutuante, como `1000.0` e `.05` na Fig. 5.6, como do tipo `double`. De forma semelhante, Java trata números inteiros constantes, como `7` e `-22`, como do tipo `int`. As linhas 21 e 22 declaram a referência `moneyFormat`, da classe `NumberFormat`, e a inicializam chamando o método `static getCurrencyInstance` da classe `NumberFormat`. Este método devolve um objeto `NumberFormat` que pode formatar valores numéricos como dinheiro (p. ex., nos Estados unidos, os valores em dinheiro normalmente são precedidos um sinal de cífrão, \$). O argumento para o método – `Locale.US` – indica que os valores em dinheiro devem ser exibidos com um sinal de cífrão (\$), devem usar uma casa decimal para separar dóla-

```

1 // Fig. 5.6: Interest.java
2 // Calculando juros compostos
3
4 // Pacotes do núcleo de Java
5 import java.text.NumberFormat;
6 import java.util.Locale;
7
8 // Pacotes de extensão de Java
9 import javax.swing.JOptionPane;
10 import javax.swing.JTextArea;
11
12 public class Interest {
13
14     // método main inicia a execução do aplicativo
15     public static void main( String args[] )
16     {
17         double amount, principal = 1000.0, rate = 0.05,
18
19         // cria DecimalFormat para formatar números em ponto flutuante
20         // com dois dígitos à direita do ponto decimal
21         NumberFormat moneyFormat =
22             NumberFormat.getCurrencyInstance( Locale.US );
23
24         // cria JTextArea para exibir dados de saída
25         JTextArea outputTextArea = new JTextArea();
26
27         // define a primeira linha do texto em outputTextArea
28         outputTextArea.setText( "Year\tAmount on deposit\n" );
29
30         // calcula o novo valor para cada um dos dez anos
31         for ( int year = 1; year <= 10; year++ ) {
32
33             // calcula o novo valor para o ano especificado
34             amount = principal * Math.pow( 1.0 + rate, year );
35
36             // acrescenta uma linha de texto a outputTextArea
37             outputTextArea.append( year + "\t" +
38                 moneyFormat.format( amount ) + "\n" );
39
40     } // fim da estrutura for
41
42     // exibe resultados
43     JOptionPane.showMessageDialog( null, outputTextArea,
44         "Compound Interest", JOptionPane.INFORMATION_MESSAGE );
45
46     System.exit( 0 );    // termina o aplicativo

```

Fig. 5.6 Cálculo de juros compostos com a estrutura `for` (parte 1 de 2).

```

47
48     } // fim do método main
49
50 } // fim da classe Interest

```

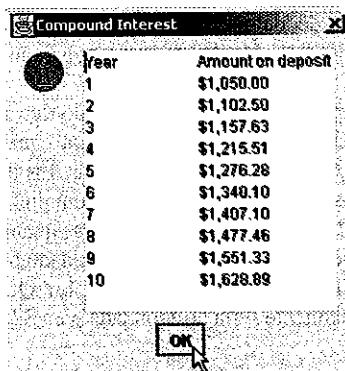


Fig. 5.6 Cálculo de juros compostos com a estrutura **for** (parte 2 de 2).

res de centavos e usar uma vírgula para delimitar milhares (p. ex., \$1,234.56)*. A classe **Locale** fornece constantes que podem ser usadas para personalizar este programa e representar valores em dinheiro para outros países, de modo que os formatos de dinheiro sejam exibidos adequadamente para cada *locale* (i.e., o formato em moeda local para cada país). A classe **NumberFormat** (importada na linha 5) está localizada no pacote **java.text** e a classe **Locale** (importada na linha 6) está localizada no pacote **java.util**.

A linha 25 declara a referência **outputTextArea**, da classe **JTextArea**, e a inicializa com um novo objeto da classe **JTextArea** (do pacote **javax.swing**). Uma **JTextArea** é um componente GUI que é capaz de exibir muitas linhas de texto. O diálogo de mensagem que exibe a **JTextArea** determina a largura e a altura da **JTextArea**, com base no **String** que ela contém. Apresentamos esse componente GUI agora porque veremos muitos exemplos por todo o texto em que as saídas do programa conterão linhas demais para exibir na tela. Esse componente GUI permitirá rolar as linhas de texto para que possamos ver toda a saída do programa. Os métodos para colocar texto em uma **JTextArea** incluem **setText** e **append**.

A linha 28 utiliza o método **setText** da classe **JTextArea** para colocar um **String** na **JTextArea** à qual **outputTextArea** faz referência. Inicialmente, **JTextArea** contém um **String** vazio (i. e., um **String** sem caracteres). A instrução precedente substitui o **string** vazio por um que contém os cabeçalhos de coluna para nossas duas colunas de saída – “**Year**” e “**Amount on Deposit**”. Os cabeçalhos de coluna são separados com um caractere de tabulação (a sequência de escape **\t**). Além disso, o **string** contém o caractere nova linha (sequência de escape **\n**) para indicar que qualquer texto adicional acrescentado à **JTextArea** deve iniciar na linha seguinte.

A estrutura **for** (linhas 31 a 40) executa seu corpo 10 vezes, fazendo a variável de controle **year** variar de 1 a 10 em incrementos de 1 (observe que **year** representa *n* na definição do problema). Java não inclui um operador de exponenciação. Em vez disso, utilizamos o método **static pow** da classe **Math** para esse propósito. **Math.pow(x, y)** calcula o valor de **x** elevado à **y**-ésima potência. O método **pow** recebe dois argumentos do tipo **double** e devolve um valor **double**. A linha 34 executa o cálculo pedido na definição do problema,

$$a = p(1 + r)^n$$

onde *a* é **amount**, *p* é **principal**, *r* é **rate** e *n* é **year**.

As linhas 37 e 38 acrescentam (**append**) mais texto ao fim da **outputTextArea**. O texto inclui o valor atual de **year**, um caractere de tabulação (para posicionar na segunda coluna), o resultado da chamada do método **moneyFormat.format(amount)** – que formata **amount** como valor em moeda norte-americana – e um caractere de nova linha (para posicionar o cursor na **JTextArea** na começo da próxima linha).

*N. de R. Nos EUA, utiliza-se o ponto, e não a vírgula, para separar a parte inteira da decimal de um número.

As linhas 43 e 44 exibem os resultados em um diálogo de mensagem. Até agora, a mensagem exibida sempre foi um `String`. Nesse exemplo, o segundo argumento é `outputTextArea` – um componente GUI. Um recurso interessante da classe `JOptionPane` é que a mensagem que ela exibe com `showMessageDialog` pode ser um `String` ou um componente GUI como uma `JTextArea`. Nesse exemplo, o diálogo de mensagem dimensiona a si mesmo para acomodar a `JTextArea`. Utilizamos essa técnica várias vezes no início deste capítulo para exibir saídas longas baseadas em texto. Mais tarde, demonstramos como adicionar a capacidade de rolagem à `JTextArea` para que o usuário possa visualizar uma saída de programa que é muito grande para ser exibida inteiramente na tela.

Repare que as variáveis `amount`, `principal` e `rate` são do tipo `double`. Fizemos isso para simplificar porque estamos lidando com partes fracionárias de dólares e, portanto, precisamos de um tipo que permita pontos decimais em seus valores. Infelizmente, isso pode causar problemas. Eis uma explicação simples do que pode dar errado ao se utilizar `float` ou `double` para representar quantias em dinheiro (supondo-se que as quantias em dinheiro são exibidas com dois dígitos à direita da casa decimal): duas quantidades de dinheiro `double` armazenadas na máquina poderiam ser 14,234 (que normalmente seria arredondado para 14,23 para fins de exibição) e 18,673 (que normalmente seria arredondado para 18,67 para fins de exibição). Quando se somam essas quantidades, produz-se a soma interna 32,907, que normalmente seria arredondada para 32,91 para fins de exibição. Assim sua impressão poderia ser

$$\begin{array}{r} 14.23 \\ + 18.67 \\ \hline 32.91 \end{array}$$

mas uma pessoa que somasse os números impressos obteria a soma 32,90. Você foi avisado!



Boa prática de programação 5.1

Não utilize variáveis de tipo `float` ou `double` para executar cálculos precisos com valores em dinheiro. A imprecisão de números de ponto flutuante pode causar erros que resultarão em valores em dinheiro incorretos. Nos exercícios, exploramos o uso de inteiros para realizar cálculos em dinheiro. [Nota: alguns fornecedores independentes vendem bibliotecas de classes para executar cálculos precisos com dinheiro.]

Observe que o corpo da estrutura `for` contém o cálculo `1.0 + rate`, que aparece como argumento para o método `Math.pow`. Na verdade, esse cálculo produz o mesmo resultado cada vez que o laço é executado, de modo que repetir o cálculo em todas as iterações do laço é desperdício.



Dica de desempenho 5.1

Evite colocar expressões cujos valores não se alteram dentro dos laços. Mas, mesmo se isso for feito, muitos compiladores sofisticados atuais com otimização colocarão automaticamente tais expressões fora dos laços no código gerado em linguagem de máquina.



Dica de desempenho 5.2

Muitos compiladores contêm recursos de otimização que melhoraram o código que você escreve, mas ainda é melhor escrever direito o código desde o início.

5.5 A estrutura de seleção múltipla `switch`

Discutimos a estrutura `if` de seleção única e a estrutura `if/else` de seleção dupla. De vez em quando, o algoritmo conterá uma série de decisões em que o algoritmo testa uma variável ou expressão separadamente para cada um dos vários valores inteiros constantes (isto é, valores dos tipos `byte`, `short`, `int` e `char`) que a variável ou expressão pode assumir e toma ações diferentes com base nesses valores. Java oferece a estrutura de seleção múltipla `switch` para tratar uma tomada de decisão como esta. O applet da Fig. 5.7 demonstra o desenho de linhas, retângulos ou elipses com base em um inteiro que o usuário digita em um diálogo de entrada.

```

1 // Fig. 5.7: Switchtest.java
2 // Desenhando linhas, retângulos ou elipses com base na opção do usuário
3
4 // Pacotes do núcleo de Java
5 import java.awt.Graphics;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class SwitchTest extends JApplet {
11     int choice; // opção do usuário quanto a que forma desenhar
12
13     // inicializa o applet obtendo a escolha do usuário
14     public void init()
15     {
16         String input; // valor digitado pelo usuário
17
18         // obtém a opção do usuário
19         input = JOptionPane.showInputDialog(
20             "Enter 1 to draw lines\n" +
21             "Enter 2 to draw rectangles\n" +
22             "Enter 3 to draw ovals\n" );
23
24         // converte o que o usuário digitou em um int
25         choice = Integer.parseInt( input );
26     }
27
28     // desenha formas no fundo do applet
29     public void paint( Graphics g )
30     {
31         // chama a versão herdada do método paint
32         super.paint( g );
33
34         // repete 10 vezes, contando de 0 a 9
35         for ( int i = 0; i < 10; i++ ) {
36
37             // determina a forma a desenhar com base na opção do usuário
38             switch( choice ) {
39
40                 case 1:
41                     g.drawLine( 10, 10, 250, 10 + i * 10 );
42                     break; // fim do processamento do case
43
44                 case 2:
45                     g.drawRect( 10 + i * 10, 10 + i * 10,
46                         50 + i * 10, 50 + i * 10 );
47                     break; // fim do processamento do case
48
49                 case 3:
50                     g.drawOval( 10 + i * 10, 10 + i * 10,
51                         50 + i * 10, 50 + i * 10 );
52                     break; // fim do processamento do case
53
54                 default:
55                     g.drawString("Invalid value entered",
56                         10, 20 + i * 15 );
57
58             } // fim da estrutura switch
59
60         } // fim da estrutura for

```

Fig. 5.7 Um exemplo que utiliza switch (parte 1 de 2).

```
61
62     } // fim do método paint
63
64 } // fim da classe SwitchTest
```

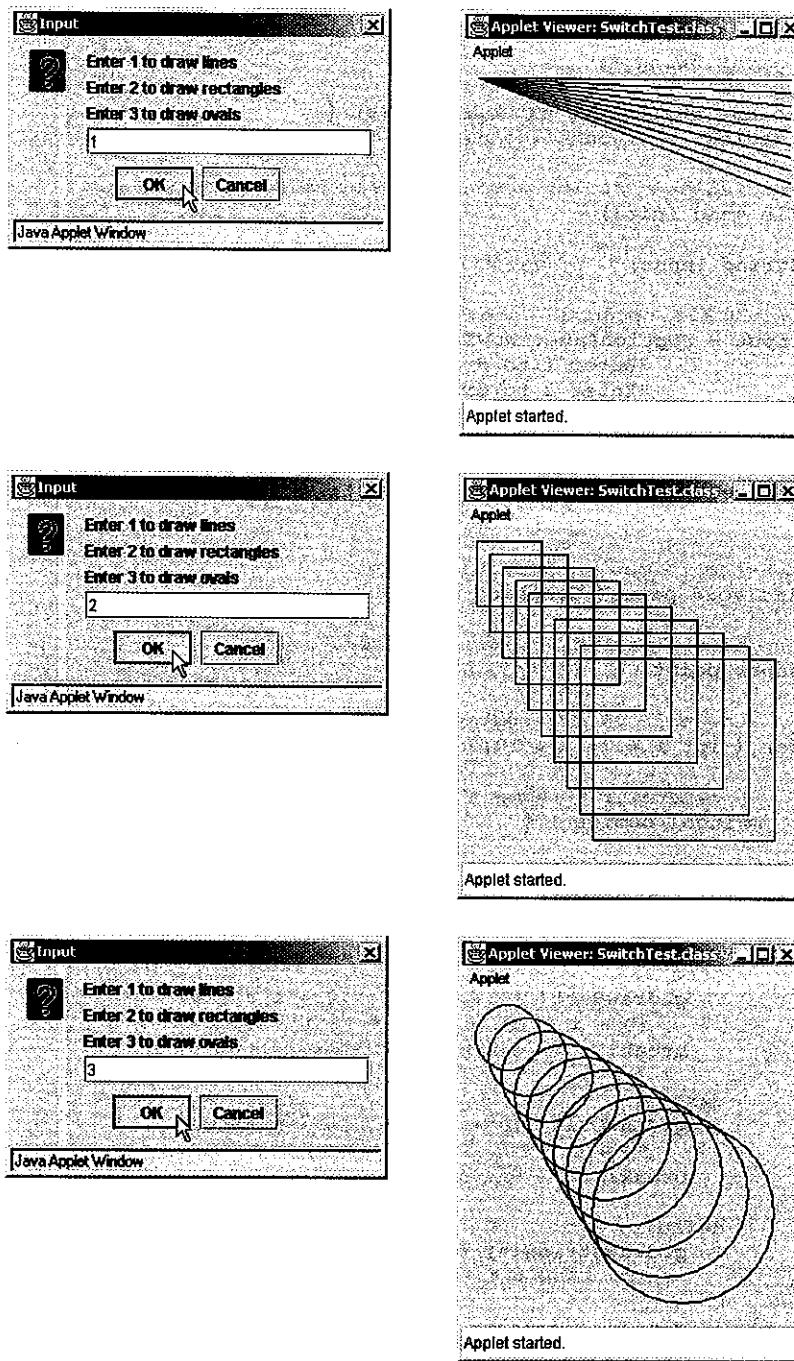


Fig. 5.7 Um exemplo que utiliza `switch` (parte 2 de 2).

A linha 11 no applet **SwitchTest** define a variável de instância **choice** do tipo **int**. Essa variável armazena os dados de entrada do usuário que determinam que tipo de forma desenhar em **paint**.

O método **init** (linhas 14 a 26) declara a variável local **input** do tipo **String** na linha 16. Essa variável armazena o **String** que o usuário digita no diálogo de entrada. As linhas 19 a 22 exibem o diálogo de entrada com o método **static JOptionPane.showInputDialog** e pedem para o usuário digitar 1 para desenhar linhas, 2 para desenhar retângulos ou 3 para desenhar elipses. A linha 25 converte os dados de entrada de um **String** para um **int** e atribui o resultado a **choice**.

O método **paint** (linhas 29 a 62) contém uma estrutura **for** (linhas 35 a 60) que executa o laço 10 vezes. Neste exemplo, o cabeçalho da estrutura **for**, na linha 35, utiliza contagem baseada em zero. Os valores de **i** para as 10 iterações do laço são 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9 e o laço termina quando o valor de **i** se torna 10. [Nota: como você sabe, o contêiner de *applets* chama o método **paint** depois dos métodos **init** e **start**. O contêiner de *applets* também chama o método **paint** sempre que a área do *applet* na tela precisa ser redesenhada – p. ex., depois que uma outra janela que cobria a área do *applet* é movida para uma posição diferente na tela.]

Aninhada no corpo da estrutura **for** há uma estrutura **switch** (linhas 38 a 58) que desenha formas com base na leitura de um inteiro digitado pelo usuário no método **init**. A estrutura **switch** consiste em uma série de **rótulos case** e um **caso default** opcional.

Quando o fluxo de controle alcança a estrutura **switch**, o programa avalia a *expressão de controle* (**choice**) entre parênteses que segue a palavra-chave **switch**. O programa compara o valor da expressão de controle (que deve ser avaliada como um valor integral do tipo **byte**, **char**, **short** ou **int**) com cada **rótulo case**. Suponha que o usuário digitou o inteiro 2 como opção. O programa compara 2 com cada **case** no **switch**. Se ocorrer uma coincidência (**case 2 :**), o programa executa as instruções para aquele **case**. Para o inteiro 2, as linhas 44 a 47 desenham um retângulo, utilizando quatro argumentos, que representam a coordenada superior esquerda **x**, a coordenada superior esquerda **y** e a largura e a altura do retângulo, e a estrutura **switch** encerra imediatamente com a instrução **break**. Então, o programa incrementa a variável contador na estrutura **for** e reavalia a condição de continuação do laço para determinar se deve executar outra iteração do laço.

A instrução **break** faz com que o fluxo de controle do programa prossiga com a primeira instrução depois da estrutura **switch** (nesse caso, alcançamos o fim do corpo da estrutura **for**, de modo que o controle flui para a expressão de incremento da variável de controle no cabeçalho da estrutura **for**). Sem **break**, os **cases** em uma instrução **switch** seriam todos executados. Cada vez que ocorre uma coincidência na estrutura, as instruções para todo os **cases** restantes serão executadas. (Esse recurso é perfeito para a programação da canção iterativa “The Twelve Days of Christmas”.) Se nenhuma coincidência ocorrer entre o valor da expressão de controle e um rótulo **case**, o caso **default** é executado e o programa desenha uma mensagem de erro no *applet*.

Cada **case** pode ter múltiplas ações. A estrutura **switch** é diferente de outras estruturas porque não exige chaves que envolvem múltiplas ações em cada **case**. A Fig. 5.8 mostra o fluxograma da estrutura **switch** genérica (utilizando um **break** em cada **case**). [Nota: como exercício, faça um fluxograma para a estrutura **switch** genérica sem **breaks**.]

O fluxograma torna claro que cada instrução **break** no final de um **case** faz com que o controle saia imediatamente da estrutura **switch**. A instrução **break** não é necessária para o último **case** na estrutura **switch** (ou o caso **default** quando ele aparece por último) porque o programa continua automaticamente com a próxima instrução depois do **switch**.

Novamente, observe que, além de pequenos círculos e setas, o fluxograma contém apenas símbolos de ação e símbolos de decisão. É responsabilidade do programador preencher os retângulos e losangos com ações e decisões apropriadas para o algoritmo. Embora o aninhamento de estruturas de controle seja comum, é raro encontrar estruturas **switch** aninhadas em um programa.



Erro comum de programação 5.7

Esquecer uma instrução break quando esta for necessária em uma estrutura switch é um erro de lógica.



Boa prática de programação 5.11

Incluir um caso default nas instruções switch. Os casos não explicitamente testados em uma instrução switch sem um caso default são ignorados. Incluir um caso default concentra o programador na necessidade de processar condições excepcionais. Há situações em que nenhum processamento default é necessário.

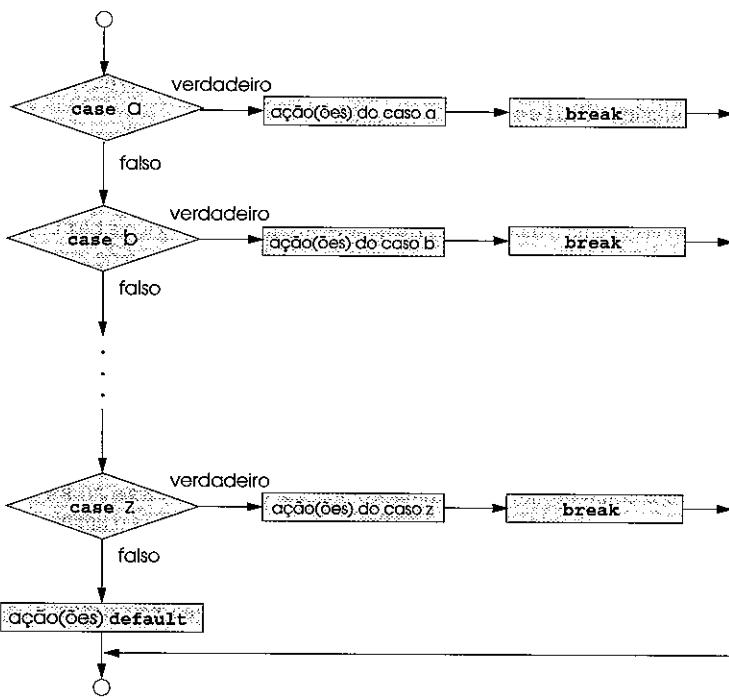


Fig. 5.8 A estrutura de seleção múltipla `switch`.



Boa prática de programação 5.12

Embora `cases` e o caso `default` em uma estrutura `switch` possam ocorrer em qualquer ordem, é considerada uma boa prática de programação colocar a cláusula `default` por último.



Boa prática de programação 5.13

Em uma estrutura `switch`, quando a cláusula `default` é listada por último, o `break` para essa instrução `case` não é exigido. Alguns programadores incluem este `break` para clareza e simetria com outros casos.

Observe que listar rótulos `case` juntos (como `case 1 : case 2 :` sem instruções entre os casos) resulta na execução do mesmo conjunto de ações para cada um dos casos.

Ao utilizar a estrutura `switch`, lembre-se de que a expressão depois de cada `case` pode apenas ser uma *expressão integral constante* (isto é, qualquer combinação de constantes de caracteres e constantes inteiras que é avaliada como um valor inteiro constante). Uma constante de caractere é representada como o caractere específico entre aspas simples, como '`A`'. Uma constante inteira é simplesmente um valor inteiro. A expressão depois de cada `case` também pode ser uma *variável constante* – isto é, uma variável que contém um valor que não muda no programa inteiro. Tal variável é declarada com a palavra-chave `final` (discutida no Capítulo 6). Quando discutimos a programação orientada a objetos no Capítulo 9, apresentamos uma maneira mais elegante de implementar a lógica `switch`. Utilizamos uma técnica chamada de *polimorfismo* para criar programas que em geral são mais claros, mais fáceis de manter e mais fáceis de estender do que os programas que utilizam a lógica `switch`.

5.6 A estrutura de repetição do/while

A estrutura de repetição `do/while` é semelhante à estrutura `while`. Na estrutura `while`, o programa testa a condição de continuação do laço no começo do laço, antes de executar seu corpo. A estrutura `do/while` testa a condição de continuação do laço *depois* de executar o corpo do laço; portanto, o corpo do laço sempre é executado pelo menos uma vez. Quando uma estrutura `do/while` termina, a execução continua com a instrução depois da cláusula `while`. Observe que não é necessário utilizar chaves na estrutura `do/while` se houver apenas uma instrução no corpo. Entretanto, a maioria dos programadores inclui as chaves, para evitar confusão entre as estruturas `while` e `do/while`. Por exemplo,

```
while ( condição )
```

normalmente é a primeira linha de uma estrutura `while`. A estrutura `do/while` sem chaves em torno do corpo que contém uma única instrução aparece como

```
do
    instrução
while ( condição );
```

que pode ser confuso. O leitor pode interpretar erroneamente a última linha – `while (condição);` – como uma estrutura `while` que contém uma instrução vazia (o ponto-e-vírgula sozinho). Portanto, para evitar confusão, a estrutura `do/while` com uma instrução é freqüentemente escrita como segue:

```
do {
    instrução
} while ( condição );
```

Boa prática de programação 5.14



Alguns programadores sempre incluem chaves em uma estrutura `do/while`, mesmo se não forem necessárias. Isso ajuda a eliminar a ambigüidade entre a estrutura `while` e a estrutura `do/while` que contém uma única instrução.

Erro comum de programação 5.8



Os laços infinitos ocorrem quando a condição de continuação do laço em uma estrutura `while`, `for` ou `do/while` nunca se torna `false`. Para evitar essa situação, certifique-se de que não haja um ponto-e-vírgula logo depois do cabeçalho de uma estrutura `while` ou `for`. Em um laço controlado por contador, certifique-se de que a variável de controle seja incrementada (ou decrementada) no corpo do laço. Em um laço controlado por sentinelas, certifique-se de que o valor da sentinelas seja digitado em algum momento.

O applet na Fig. 5.9 utiliza uma estrutura `do/while` para desenhar 10 círculos aninhados com o método `drawOval` da classe `Graphics`.

```
1 // Fig. 5.9: Dowhiletest.java
2 // Utilizando a estrutura de repetição do/while
3
4 // Pacotes do núcleo de Java
5 import java.awt.Graphics;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JApplet;
9
10 public class DoWhileTest extends JApplet {
11
12     // desenha linhas no fundo do applet
13     public void paint( Graphics g )
14     {
15         // chama a versão herdada do método paint
16         super.paint( g );
17
18         int counter = 1;
```

Fig. 5.9 Utilizando a estrutura de repetição `do/while` (parte 1 de 2).

```

19
20     do {
21         g.drawOval( 110 - counter * 10, 110 - counter * 10,
22                     counter * 20, counter * 20 );
23         ++counter;
24     } while ( counter <= 10 ); // fim da estrutura do/while
25
26 } // fim do método paint
27
28 } // fim da classe DoWhileTest

```

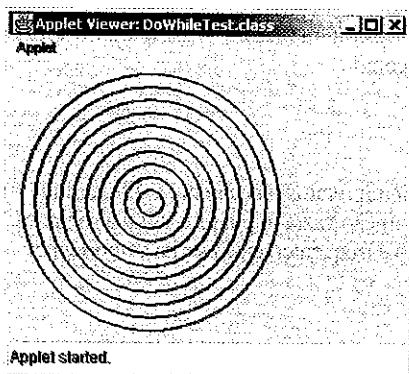


Fig. 5.9 Utilizando a estrutura de repetição `do/while` (parte 2 de 2).

No método `paint` (linhas 13 a 26), a linha 18 declara a variável de controle `counter` e a inicializa com 1. Ao entrar na estrutura `do/while`, as linhas 21 e 22 enviam a mensagem `drawOval` para o objeto `Graphics` ao qual `g` faz referência. Os quatro argumentos que representam a coordenada *x* superior esquerda, a coordenada *y* superior esquerda, a largura e a altura da *caixa delimitadora* da elipse (um retângulo imaginário no qual a elipse toca o centro de todos os quatro lados do retângulo) são calculados com base no valor da variável de controle `counter`. O programa desenha primeiro a elipse mais interna. O canto superior esquerdo da caixa delimitadora para cada elipse subsequente move-se para mais perto do canto superior esquerdo do *applet*. Ao mesmo tempo, a largura e a altura da caixa delimitadora são aumentados, para assegurar que cada nova elipse contenha todas as primeiras elipses. A linha 23 incrementa `counter`. Então, o programa avalia o teste de continuação do laço na parte inferior do laço. O fluxograma `do/while` (Fig. 5.10) deixa claro que o programa não avalia a condição de continuação do laço enquanto a ação não for executada pelo menos uma vez.

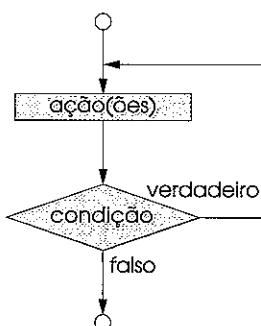


Fig. 5.10 Fluxograma da estrutura de repetição `do/while`.

5.7 As instruções break e continue

As instruções `break` e `continue` alteram o fluxo de controle. A instrução `break`, quando executada em uma estrutura `while`, `for`, `do/while` ou `switch`, causa a saída imediata dessa estrutura. A execução continua com a primeira instrução depois da estrutura. Utilizações comuns da instrução `break` são escapar no começo de um laço ou pular o restante de uma estrutura `switch` (como na Fig. 5.7). A Fig. 5.11 demonstra a instrução `break` em uma estrutura de repetição `for`.

```

1 // Fig. 5.11: Breaktest.java
2 // Utilizando a instrução break em uma estrutura for
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 public class BreakTest {
8
9     // método main inicia a execução do aplicativo em Java
10    public static void main( String args[] )
11    {
12        String output = "";
13        int count;
14
15        // repete 10 vezes
16        for ( count = 1; count <= 10; count++ ) {
17
18            // se a contagem for igual a 5, termina o laço
19            if ( count == 5 )
20                break; // interrompe o laço somente se count == 5
21
22            output += count + " ";
23
24        } // fim da estrutura for
25
26        output += "\nBroke out of loop at count = " + count;
27        JOptionPane.showMessageDialog( null, output );
28
29        System.exit( 0 ); // termina o aplicativo
30
31    } // fim do método main
32
33 } // fim da classe BreakTest

```

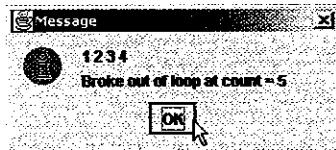


Fig. 5.11 Utilizando a instrução `break` em uma estrutura `for`.

Quando a estrutura `if` na linha 19 da estrutura `for` detecta que `count` é 5, a instrução `break` na linha 20 é executada. Essa instrução termina a estrutura `for` e o programa prossegue para a linha 26 (logo depois do `for`). A linha 26 completa o `string` a exibir em um diálogo de mensagem na linha 27. O corpo do laço é executado completamente apenas quatro vezes.

A instrução `continue`, quando executada em uma estrutura `while`, `for` ou `do/while`, pula as instruções restantes no corpo do laço e prossegue com a próxima iteração do laço. Nas estruturas `while` e `do/while`, o programa avalia o teste de continuação do laço imediatamente depois da instrução `continue` ser executada. Em estruturas `for`, a expressão de incremento é executada e depois o programa avalia o teste de continuação do laço. Anteriormente, afirmamos que a estrutura `while` poderia ser utilizada na maioria dos casos para representar a estrutura `for`. A única exceção ocorre quando a expressão de incremento na estrutura `while` fica depois da instrução `continue`. Nesse caso, o incremento não é executado antes de o programa avaliar a condição de continuação da repetição, de modo que a estrutura `while` não é executada da mesma maneira que a estrutura `for` o é. A Fig. 5.12

utiliza a instrução **continue** em uma estrutura **for** para pular a instrução de concatenação de *strings* (linha 22) quando a estrutura **if** (linha 18) determina que o valor de **count** é 5. Quando a instrução **continue** é executada, o controle do programa continua com o incremento da variável de controle na estrutura **for**.



Boa prática de programação 5.15

*Alguns programadores consideram que **break** e **continue** violam a programação estruturada. Como os efeitos dessas instruções podem ser obtidos através de técnicas de programação estruturada, esses programadores não utilizam **break** e **continue**.*



Dica de desempenho 5.3

*As instruções **break** e **continue**, quando utilizadas adequadamente, têm desempenho mais rápido que as técnicas estruturadas correspondentes.*



Observação de engenharia de software 5.2

Há um conflito entre conseguir engenharia de software de qualidade e conseguir o software de melhor desempenho. Freqüentemente, um desses objetivos é alcançado à custa do outro. Para todas as situações, exceto as de desempenho muito alto, aplique a seguinte regra geral: primeiro, faça seu código simples e correto; depois, somente se necessário, torne-o rápido e pequeno.

```

1 // Fig. 5.12: ContinueTest.java
2 // Utilizando a instrução continue em uma estrutura for
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 public class ContinueTest {
8
9     // o método main inicia a execução do aplicativo em Java
10    public static void main( String args[] )
11    {
12        String output = "";
13
14        // repete 10 vezes
15        for ( int count = 1; count <= 10; count++ ) {
16
17            // se a contagem for 5, continua com a próxima iteração do laço
18            if ( count == 5 )
19                continue; // pula o código restante no laço
20                // somente se count == 5
21
22            output += count + " ";
23
24        } // fim da estrutura for
25
26        output += "\nUsed continue to skip printing 5";
27        JOptionPane.showMessageDialog( null, output );
28
29        System.exit( 0 ); // termina o aplicativo
30
31    } // fim do método main
32
33 } // fim da classe ContinueTest

```

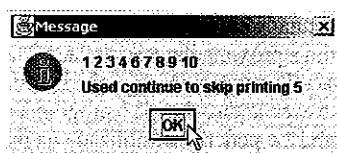


Fig. 5.12 Utilizando a instrução **continue** em uma estrutura **for**.

5.8 As instruções rotuladas `break` e `continue`

A instrução `break` pode interromper apenas uma estrutura `while`, `for`, `do/while` ou `switch` que a envolve imediatamente. Para interromper um conjunto aninhado de estruturas, você pode utilizar a *instrução rotulada break*. Essa instrução, quando executada em uma estrutura `while`, `for`, `do/while` ou `switch`, ocasiona a saída imediata dessa estrutura e de um número qualquer de estruturas de repetição que a envolvem; a execução do programa é retomada na primeira instrução depois do *bloco rotulado* que a envolve (isto é, um conjunto de instruções colocado entre chaves e precedido por um rótulo). O bloco pode ser uma estrutura de repetição (o corpo seria o bloco) ou um bloco no qual a estrutura de repetição é o primeiro código executável. Instruções rotuladas `break` são comumente utilizadas para terminar estruturas de laço aninhadas que contém estruturas `while`, `for`, `do/while` ou `switch`. A Fig. 5.13 demonstra uma instrução rotulada `break` em uma estrutura aninhada `for`.

O bloco (linhas 14 a 37) inicia com um *rótulo* (um identificador seguido por dois-pontos) na linha 14; aqui utilizamos o rótulo “`stop:`”. O bloco está colocado entre chaves no final da linha 14 e linha 37 e inclui a estrutura aninhada `for` (linhas 17 a 32) e a instrução de concatenação de *strings* na linha 35. Quando a estrutura `if` na linha 23 detecta que `row` é igual a 5, a instrução `break` na linha 24 é executada. Esta instrução termina tanto a estrutura `for` na linha 20 quanto a estrutura `for` que a contém, na linha 17. O programa prossegue imediatamente para a linha 39 – a primeira instrução depois do bloco rotulado. O corpo da estrutura `for` mais interna é executado completamente apenas quatro vezes. Repare que a instrução de concatenação de *strings* na linha 35 nunca é executada, porque ela está incluída no corpo do bloco rotulado e a estrutura `for` externa nunca se completa.

```

1 // Fig. 5.13 BreakLabelTest.java
2 // Utilizando a instrução break com um rótulo
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 public class BreakLabelTest {
8
9     // o método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        String output = "";
13
14        stop: { // bloco rotulado
15
16            // conta 10 linhas
17            for ( int row = 1; row <= 10; row++ ) {
18
19                // conta 5 colunas
20                for ( int column = 1; column <= 5 ; column++ ) {
21
22                    // se a linha for igual a 5, pula para o fim do bloco "stop"
23                    if ( row == 5 )
24                        break stop; // pula para o fim do bloco stop
25
26                    output += "*  ";
27
28                } // fim da estrutura interna for
29
30                output += "\n";
31
32            } // fim da estrutura externa for
33
34            // a linha seguinte é pulada
35            output += "\nLoops terminated normally";
36
37        } // fim do bloco rotulado
38

```

Fig. 5.13 Utilizando uma instrução rotulada `break` em uma estrutura aninhada `for` (parte 1 de 2).

```

39     JOptionPane.showMessageDialog(
40         null, output, "Testing break with a label",
41         JOptionPane.INFORMATION_MESSAGE );
42
43     System.exit( 0 ); // termina o aplicativo
44
45 } // fim do método main
46
47 } // fim da classe BreakLabelTest

```

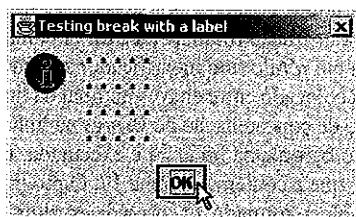


Fig. 5.13 Utilizando uma instrução rotulada `break` em uma estrutura aninhada `for` (parte 2 de 2).

A instrução `continue` prossegue com a próxima iteração (repetição) da estrutura `while`, `for` ou `do/while` imediatamente envolvente. A *instrução rotulada continue*, quando executada em uma estrutura de repetição (`while`, `for` ou `do/while`), pula as instruções restantes no corpo dessa estrutura e qualquer número de estruturas de repetição que a envolvem e prossegue com a próxima iteração da estrutura de repetição rotulada (isto é, uma estrutura de repetição precedida por um rótulo) que a envolve. Nas estruturas rotuladas `while` e `do/while`, o programa avalia o teste de continuação do laço logo depois da instrução `continue` ser executada. Em uma estrutura rotulada `for`, a expressão de incremento é executada, depois o teste de continuação do laço é avaliado. A Fig. 5.14 utiliza a rotulada instrução rotulada `continue` em uma estrutura `for` aninhada para permitir que a execução `continue` com a próxima iteração da estrutura externa `for`.

A estrutura rotulada `for` (linhas 14 a 32) inicia no rótulo `nextRow`. Quando a estrutura `if` na linha 24 da estrutura interna `for` detecta que `column` é maior que `row`, a instrução `continue` na linha 25 é executada e o controle do programa continua com o incremento da variável de controle do laço externo `for`. Embora a estrutura `for` interna conte de 1 a 10, o número de caracteres * enviados para saída em uma linha nunca excede o valor de `row`.



Dica de desempenho 5.4

O programa na Fig. 5.14 pode ficar mais simples e mais eficiente mediante a substituição da condição na estrutura `for` da linha 21 por `column <= row` e a remoção da estrutura `if` nas linhas 24 a 25 do programa.

5.9 Operadores lógicos

Até agora, estudamos somente *condições simples* como `count <= 10`, `total > 1000` e `number != sentinelValue`. Essas condições foram expressas em termos dos operadores relacionais `>`, `<`, `>=` e `<=` e os operadores de igualdade `==` e `!=`. Cada decisão testou uma condição. Para testar múltiplas condições no processo de tomada de uma decisão, executamos esses testes em instruções separadas ou em estruturas aninhadas `if` ou `if/else`.

Java fornece *operadores lógicos* que podem ser utilizados para formar condições mais complexas combinando condições simples. Os operadores lógicos são `&&` (*E lógico*), `&` (*E lógico booleano*), `||` (*OU lógico*), `|` (*OU lógico booleano inclusivo*), `^` (*OU lógico booleano exclusivo*) e `!` (*NÃO lógico*, também chamado de *negação lógica*).

```

1 // Fig. 5.14: ContinueLabelTest.java
2 // Utilizando a instrução continue com um rótulo
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6

```

Fig. 5.14 Utilizando uma instrução rotulada `continue` em uma estrutura aninhada `for` (parte 1 de 2).

```

7  public class ContinueLabelTest {
8
9      // método main inicia a execução do aplicativo Java
10     public static void main( String args[] )
11     {
12         String output = "";
13
14         nextRow: // rótulo-alvo da instrução continue
15
16         // conta 5 linhas
17         for ( int row = 1; row <= 5; row++ ) {
18             output += "\n";
19
20             // conta 10 colunas por linha
21             for ( int column = 1; column <= 10; column++ ) {
22
23                 // se a coluna for maior que a linha, inicia próxima linha
24                 if ( column > row )
25                     continue nextRow; // próxima iteração do
26                                     // laço rotulado
27
28                 output += "* ";
29
30             } // fim da estrutura interna for
31
32         } // fim da estrutura externa for
33
34         JOptionPane.showMessageDialog(
35             null, output, "Testing continue with a label",
36             JOptionPane.INFORMATION_MESSAGE );
37
38         System.exit( 0 ); // termina o aplicativo
39
40     } // fim do método main
41
42 } // fim da classe ContinueLabelTest

```

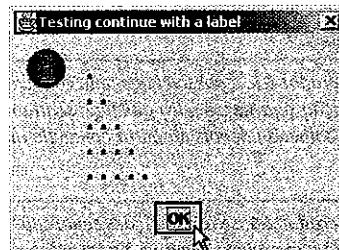


Fig. 5.14 Utilizando uma instrução rotulada `continue` em uma estrutura aninhada `for` (parte 2 de 2).

Suponha que queremos assegurar em algum ponto de um programa que duas condições sejam, *ambas*, verdadeiras, antes de escolher um certo caminho de execução. Nesse caso, podemos utilizar o operador lógico `&&` como segue:

```
if ( sexo == 1 && idade >= 65 )
    ++senhorasIdosas;
```

Essa instrução `if` contém duas condições simples. A condição `sexo == 1` pode ser avaliada, por exemplo, para determinar se a pessoa é uma mulher. A condição `idade >= 65` é avaliada para determinar se uma pessoa é um idoso. As duas condições simples são avaliadas primeiro, uma vez que as precedências de `==` e `>=` são mais altas que a precedência de `&&`. A instrução `if` então considera a condição combinada

```
sexo == 1 && idade >= 65
```

Essa condição é verdadeira se e somente se ambas as condições simples forem verdadeiras. Se essa condição combinada for de fato verdadeira, a instrução do corpo da estrutura `if` incrementa a variável `senhorasIdosas` por 1. Se qualquer uma ou ambas as condições simples forem falsas, o programa pula o incremento e prossegue para a instrução seguinte da estrutura `if`. A condição combinada precedente pode ficar mais legível adicionando-se parênteses redundantes:

```
( sexo == 1 ) && ( idade >= 65 )
```

A tabela da Fig. 5.15 resume o operador `&&`. A tabela mostra todas as quatro combinações possíveis de valores `false` e `true` para *expressão1* e *expressão2*. Essas tabelas são freqüentemente chamadas de *tabelas-verdade*. Java avalia como `false` ou `true` todas as expressões que incluem operadores relacionais, operadores de igualdade e/ou operadores lógicos.

Agora, vamos considerar o operador `||` (OU lógico). Suponha que desejemos assegurar que qualquer uma *ou* as duas condições sejam verdadeiras antes de escolhermos um certo caminho de execução. Nesse caso, utilizamos o operador `||`, como no segmento de programa seguinte:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
    System.out.println ( "Student grade is A" );
```

Essa instrução também contém duas condições simples. A condição `semesterAverage >= 90` é avaliada para determinar se o aluno merece um “A” no curso por causa de um desempenho sólido ao longo de todo o semestre. A condição `finalExam >= 90` é avaliada para determinar se o aluno merece um “A” no curso por causa de um desempenho destacado no exame final. A instrução `if` considera a condição combinada

```
semesterAverage >= 90 || finalExam >= 90
```

e premia o aluno com um “A” se qualquer uma ou as ambas condições simples forem `true`. Observe que a única vez em que a mensagem “**Student grade is A**” não é impressa é quando ambas as condições simples forem `false`. A Fig. 5.16 é a tabela-verdade para o operador lógico (`||`).

O operador `&&` tem uma precedência mais alta que o operador `||`. Ambos os operadores associam-se da esquerda para a direta. A expressão que contém operadores `&&` ou `||` é avaliada só até que a verdade ou a falsidade seja detectada. Portanto, a avaliação da expressão

```
sexo == 1 && idade >= 65
```

irá parar imediatamente se `sexo` não for igual a `1` (isto é, a expressão inteira for `false`) e continua se `sexo` for igual a `1` (isto é, a expressão inteira poderia ainda ser `true` se a condição `idade >= 65` fosse `true`). Esse recurso de desempenho para a avaliação de expressões com E lógico e OU lógico chama-se *avaliação em curto-circuito*.



Erro comum de programação 5.9

Em expressões que utilizam o operador `&&`, é possível que uma condição – vamos chamá-la de condição dependente – possa exigir que outra condição seja `true` para que faça sentido avaliar a condição dependente. Neste caso, a condição dependente deve ser colocada depois da outra condição ou pode ocorrer um erro.



Dica de desempenho 5.5

Em expressões que utilizam o operador `&&`, se as condições separadas forem independentes entre si, coloque como condição mais à esquerda aquela que provavelmente é `false`. Em expressões que utilizam o operador `||`, colo-

expressão1	expressão2	expressão1 && expressão2
<code>false</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>false</code>
<code>true</code>	<code>true</code>	<code>true</code>

Fig. 5.15 Tabela-verdade para o operador `&&` (E lógico).

que como condição mais à esquerda aquela que provavelmente é `true`. Isso pode reduzir o tempo de execução de um programa.

Os operadores *E lógico booleano* (`&`) e *OU lógico booleano inclusivo* (`|`), funcionam de maneira idêntica à dos operadores lógicos regulares E e OU lógicos, com uma exceção: os operadores lógicos booleanos sempre avaliam seus dois operandos (i.e., não há avaliação em curto-círcuito). Portanto, a expressão

```
sexo == 1 & idade >= 65
```

avalia `idade >= 65` independentemente de `sexo` ser ou não igual a 1. Esse método é útil se o operando à direita do operador lógico booleano E ou do operador lógico booleano inclusivo OU, tiver um *efeito colateral* necessário – uma modificação no valor de uma variável. Por exemplo, a expressão

```
aniversario == true | ++idade >= 65
```

garante que a condição `++idade >= 65` será avaliada. Portanto, a variável `idade` será incrementada na expressão precedente independentemente de a expressão total ser `true` ou `false`.

expressão1	expressão2	expressão1 expressão2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.16 Tabela-verdade para o operador `||` (OU lógico).



Boa prática de programação 5.16

Por questões de clareza, evite expressões com efeitos colaterais nas condições. Os efeitos colaterais podem parecer inteligentes, mas freqüentemente dão mais problemas do que valem.

A condição que contém o operador lógico booleano exclusivo *OU* (`^`) é `true` se e somente se um de seus operandos resultar em um valor `true` e um deles resultar em um valor `false`. Se ambos os operandos forem `true` ou ambos forem `false`, o resultado da condição inteira será `false`. A Fig. 5.17 é uma tabela-verdade para o operador lógico booleano exclusivo *OU* (`^`). Também é garantido que esse operador avaliará ambos os seus operandos (isto é, não há avaliação em curto-círcuito).

Java fornece o operador `!` (negação lógica) para permitir ao programador “inverter” o significado de uma condição. Diferentemente dos operadores lógicos `&&`, `&`, `||`, `|` e `^` que combinam duas condições (i.e., eles são operadores binários), o operador lógico de negação tem apenas uma única condição como operando (i.e., é um operador unário). O operador lógico de negação é colocado antes de uma condição para escolher um caminho de execução se a condição original (sem o operador lógico de negação) for `false`, como no seguinte segmento de programa:

```
if ( ! ( grade == sentinelValue ) )
    System.out.println( "The next grade is " + grade );
```

Os parênteses em torno da condição `grade == sentinelValue` são necessários, uma vez que o operador lógico de negação tem uma precedência mais alta que o operador de igualdade. A Fig. 5.18 é uma tabela-verdade para o operador lógico de negação.

Na maioria dos casos, o programador pode evitar a utilização da negação lógica expressando a condição diferentemente com um operador relacional ou de igualdade apropriado. Por exemplo, a instrução precedente também pode ser escrita como segue:

```
if ( grade != sentinelValue )
    System.out.println( "The next grade is " + grade );
```

Essa flexibilidade pode ajudar um programador a expressar uma condição de uma maneira mais conveniente.

expressão1	expressão2	expressão1 ^ expressão2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 5.17 Tabela-verdade para o operador lógico booleano exclusivo OU (^).

expressão	! expressão
false	true
true	false

Fig. 5.18 Tabela-verdade para o operador ! (NÃO lógico).

O aplicativo da Fig. 5.19 demonstra todos os operadores lógicos e operadores lógicos booleanos fornecendo suas tabelas-verdade. O programa utiliza concatenação de *strings* para criar o *string* que é exibido em uma *JTextArea*.

```

1 // // Fig. 5.19: LogicalOperators.java
2 // Demonstrando os operadores lógicos
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class LogicalOperators {
8
9     // método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        // cria JTextArea para exibir os resultados
13        JTextArea outputArea = new JTextArea( 17, 20 );
14
15        // acrescenta JTextArea a um JScrollPane para que
16        // o usuário possa rolar para ver os resultados
17        JScrollPane scroller = new JScrollPane( outputArea );
18
19        String output =
20
21        // cria a tabela-verdade para o operador &&
22        output += "Logical AND (&&)" +
23            "\nfalse && false: " + ( false && false ) +
24            "\nfalse && true: " + ( false && true ) +
25            "\ntrue && false: " + ( true && false ) +
26            "\ntrue && true: " + ( true && true );
27
28        // cria a tabela-verdade para o operador ||
29        output += "\n\nLogical OR (||)" +
30            "\nfalse || false: " + ( false || false ) +
31            "\nfalse || true: " + ( false || true ) +

```

Fig. 5.19 Demonstrando os operadores lógicos (parte 1 de 2).

```

32         "\ntrue || false: " + ( true || false ) +
33         "\ntrue || true: " + ( true || true );
34
35     // cria a tabela-verdade para o operador &
36     output += "\n\nBoolean logical AND (&)" +
37         "\nfalse & false: " + ( false & false ) +
38         "\nfalse & true: " + ( false & true ) +
39         "\ntrue & false: " + ( true & false ) +
40         "\ntrue & true: " + ( true & true );
41
42     // cria a tabela-verdade para o operador |
43     output += "\n\nBoolean logical inclusive OR ()" +
44         "\nfalse | false: " + ( false | false ) +
45         "\nfalse | true: " + ( false | true ) +
46         "\ntrue | false: " + ( true | false ) +
47         "\ntrue | true: " + ( true | true );
48
49     // cria a tabela-verdade para o operador ^
50     output += "\n\nBoolean logical exclusive OR (^)" +
51         "\nfalse ^ false: " + ( false ^ false ) +
52         "\nfalse ^ true: " + ( false ^ true ) +
53         "\ntrue ^ false: " + ( true ^ false ) +
54         "\ntrue ^ true: " + ( true ^ true );
55
56     // cria a tabela-verdade para o operador !
57     output += "\n\nLogical NOT (!)" +
58         "\n!false: " + ( !false ) +
59         "\n!true: " + ( !true );
60
61     outputArea.setText( output ); // coloca os resultados na JTextArea
62
63     JOptionPane.showMessageDialog( null, scroller,
64         "Truth Tables", JOptionPane.INFORMATION_MESSAGE );
65
66     System.exit( 0 ); // termina o aplicativo
67
68 } // fim do método main
69
70 } // fim da classe LogicalOperators

```

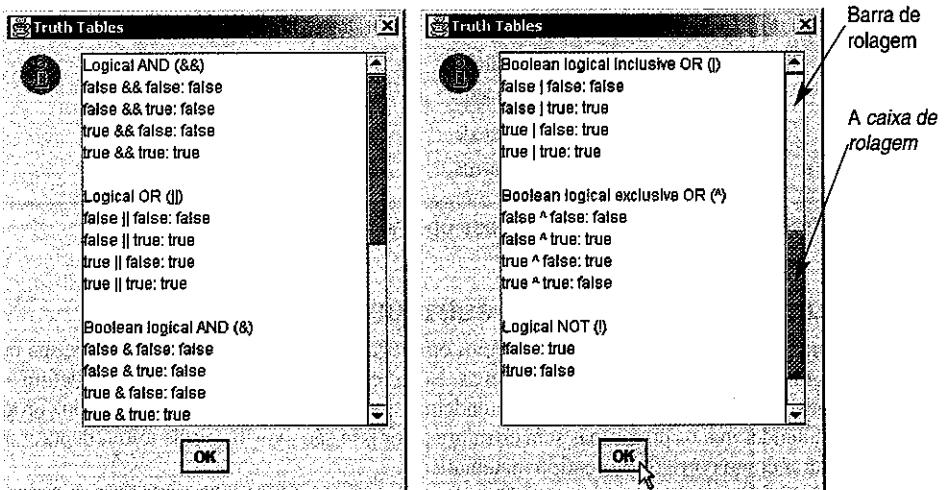


Fig. 5.19 Demonstrando os operadores lógicos (parte 2 de 2).

Na saída da Fig. 5.19, os *strings* “true” e “false” indicam os valores **true** e **false** para os operandos em cada condição. O resultado da condição é mostrado como **true** ou **false**. Repare que, quando você concatena um valor **boolean** a um **String**, Java automaticamente adiciona o *string* “false” ou “true” com base no valor **boolean**.

A linha 13 no método **main** cria uma **JTextArea**. Os números entre parênteses indicam que **JTextArea** tem 17 linhas e 20 colunas. A linha 17 declara a referência **scroller** para um **JScrollPane** e a inicializa com um novo objeto **JScrollPane**. A classe **JScrollPane** (a partir de pacote **javax.swing**) fornece um componente GUI com funcionalidade de rolagem. O objeto **JScrollPane** é inicializado com o componente GUI para o qual ele vai fornecer a funcionalidade de rolagem (**outputArea** neste exemplo). Esta inicialização acrescenta o componente GUI ao **JScrollPane**. Quando você executa esse aplicativo, repare na *barra de rolagem* no lado direito de **JTextArea**. Você pode clicar nas *setas* na parte superior ou inferior da barra de rolagem para rolar para cima ou para baixo, respectivamente, o texto na **JTextArea** uma linha por vez. Você também pode arrastar a *caixa de rolagem* para cima ou para baixo para rolar o texto rapidamente.

As linhas 22 a 59 constroem o *string* **output** que é exibido na **outputArea**. A linha 61 utiliza o método **setText** para substituir o texto em **outputArea** pelo *string* **output**. As linhas 63 e 64 exibem um diálogo de mensagem. O segundo argumento, **scroller**, indica que o **scroller** (e **outputArea** a ele anexada) devem ser exibidos como a mensagem no diálogo de mensagem.

A tabela na Fig. 5.20 mostra a precedência e a associatividade dos operadores Java introduzidos até este ponto. Os operadores são mostrados de cima para baixo em ordem decrescente de precedência.

Operadores	Associatividade	Tipo
()	da esquerda para a direita	parênteses
++ --	da direita para a esquerda	pós-fixo unário
++ -- + - ! (tipo)	da direita para a esquerda	unário
* / %	da esquerda para a direita	multiplicativo
+ -	da esquerda para a direita	aditivo
< <= > >=	da esquerda para a direita	relacional
== !=	da esquerda para a direita	igualdade
&	da esquerda para a direita	E lógico booleano
^	da esquerda para a direita	OU lógico booleano exclusivo
 	da esquerda para a direita	OU lógico booleano inclusivo
&&	da esquerda para a direita	E lógico
 	da esquerda para a direita	OU lógico
? :	da direita para a esquerda	condicional
= += -= *= /= %=	da direita para a esquerda	atribuição

Fig. 5.20 Precedência e associatividade dos operadores discutidos até agora.

5.10 Resumo de programação estruturada

Da mesma forma que os arquitetos projetam edifícios empregando o conhecimento coletivo de sua profissão, assim também os programadores devem projetar programas. Nossa campo é mais jovem que a arquitetura e nossa sabedoria coletiva é consideravelmente mais esparsa. Aprendemos que programação estruturada produz programas que são mais fáceis de entender que os programas sem estrutura e por isso são mais fáceis de testar, depurar, modificar e até mesmo demonstrar como corretos no sentido matemático.

A Fig. 5.21 resume as estruturas de controle de Java. Pequenos círculos são utilizados na figura para indicar o único ponto de entrada e o único ponto de saída de cada estrutura. Conectar símbolos de fluxograma individuais ar-

bitrariamente pode levar a programas sem estrutura. Portanto, a profissão de programação escolheu combinar símbolos de fluxograma para formar um conjunto limitado de estruturas de controle e construir programas estruturados combinando adequadamente estruturas de controle de duas maneiras simples.

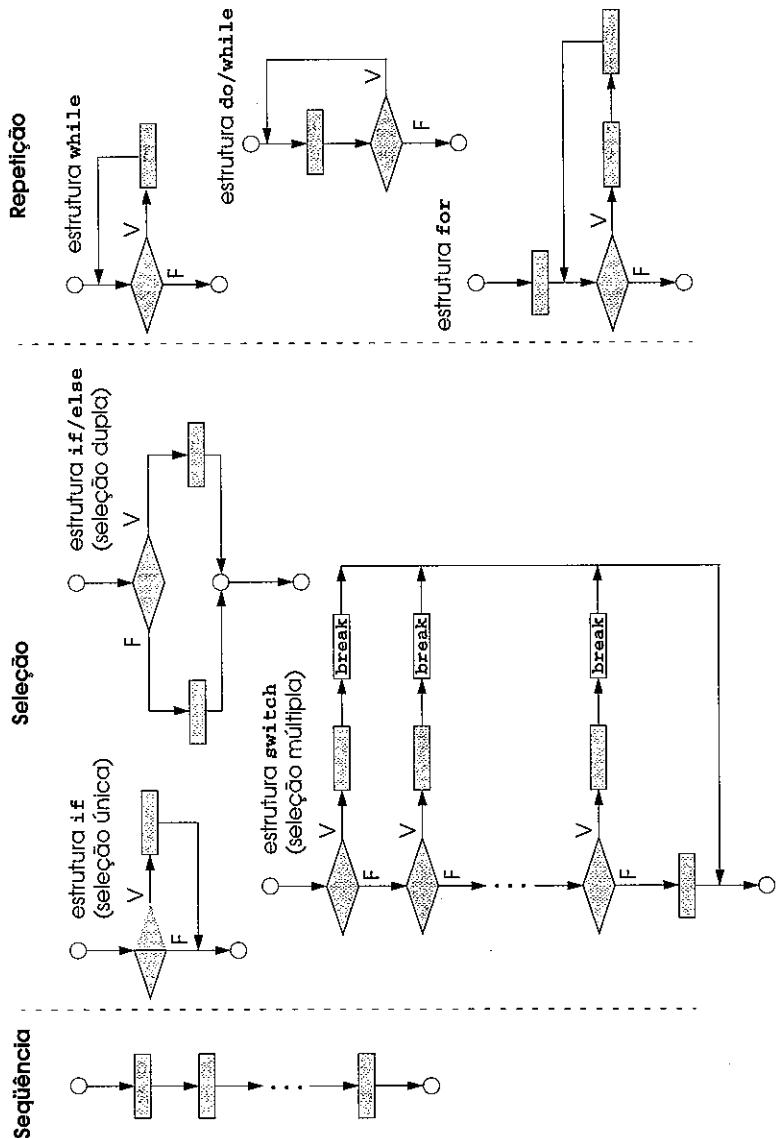


Fig. 5.21 As estruturas de controle de entrada única/saída única de Java.

Por simplicidade, são utilizadas apenas estruturas de controle de entrada única/saída única; há somente uma maneira de entrar e somente uma maneira de sair de cada estrutura de controle. Conectar estruturas de controle em seqüência para formar programas estruturados é simples: o ponto de saída de uma estrutura de controle é conectado ao ponto de entrada da próxima estrutura de controle (isto é, as estruturas de controle simplesmente são colocadas

uma depois da outra em um programa); chamamos esse método de “empilhamento de estruturas de controle”. As regras para formar programas estruturados também permitem que as estruturas de controle sejam aninhadas.

A Fig. 5.22 mostra as regras para formar programas adequadamente estruturados. As regras assumem que o símbolo de fluxograma em retângulo pode ser utilizado para indicar qualquer ação, incluindo entrada/saída.

Aplicar as regras da Fig. 5.22 sempre resulta em um fluxograma estruturado com uma aparência elegante de blocos de construção (Fig. 5.23). Por exemplo, aplicar repetidamente a Regra 2 ao fluxograma mais simples resulta em um fluxograma estruturado que contém muitos retângulos em seqüência (Fig. 5.24). A Regra 2 gera uma pilha de estruturas de controle; vamos chamar a Regra 2 de *regra de empilhamento*. [Nota: os símbolos de cima e de baixo da Fig. 5.23 representam o início e o fim de um programa, respectivamente.]

A Regra 3 é chamada de *regra de aninhamento*. Aplicar repetidamente a Regra 3 ao fluxograma mais simples resulta em um fluxograma com estruturas de controle elegantemente aninhadas. Por exemplo, na Fig. 5.25, o retângulo no fluxograma mais simples é primeiro substituído por uma estrutura (*if/else*) de seleção dupla. Então, a Regra 3 é aplicada novamente para os dois retângulos na estrutura de seleção dupla, substituindo cada um desses retângulos por estruturas de seleção dupla. As caixas tracejadas em torno de cada estrutura de seleção dupla representam o retângulo que foi substituído por uma estrutura de seleção dupla.

Regras para formar programas estruturados

- 1) Iniciar com o “fluxograma mais simples” (Fig. 5.23).
- 2) Qualquer retângulo (ação) pode ser substituído por dois retângulos (ações) em seqüência.
- 3) Qualquer retângulo (ação) pode ser substituído por qualquer estrutura de controle (seqüência, *if*, *if/else*, *switch*, *while*, *do/while* ou *for*).
- 4) As Regras 2 e 3 podem ser aplicadas quantas vezes você quiser e em qualquer ordem.

Fig. 5.22 Regras para formar programas estruturados.

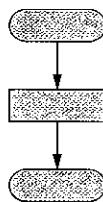


Fig. 5.23 O fluxograma mais simples.

A Regra 4 gera estruturas maiores, mais envolvidas e mais profundamente aninhadas. Os fluxogramas que emergem da aplicação das regras da Fig. 5.22 constituem o conjunto de todos os fluxogramas estruturados possíveis e, por isso, o conjunto de todos os programas estruturados possíveis.

A beleza da abordagem estruturada é que utilizamos apenas sete pedaços elementares de entrada única/saída única e os montamos de apenas duas maneiras simples. A Fig. 5.26 mostra os tipos de blocos de construção empilhados que emergem da aplicação da Regra 2 e os tipos de blocos de construção aninhados que emergem da aplicação da Regra 3. A figura também mostra o tipo de blocos de construção sobrepostos que não pode aparecer em fluxogramas estruturados (por causa da eliminação da instrução *goto*).

Se as regras na Fig. 5.22 forem seguidas, você não consegue criar um fluxograma não-estruturado (como aquele da Fig. 5.27). Se você não tem certeza se um fluxograma particular é estruturado, aplique as regras da Fig. 5.22 na ordem inversa, para tentar reduzir o fluxograma ao fluxograma mais simples. Se o fluxograma for reduzível ao fluxograma mais simples, o fluxograma original é estruturado; caso contrário, ele não é.

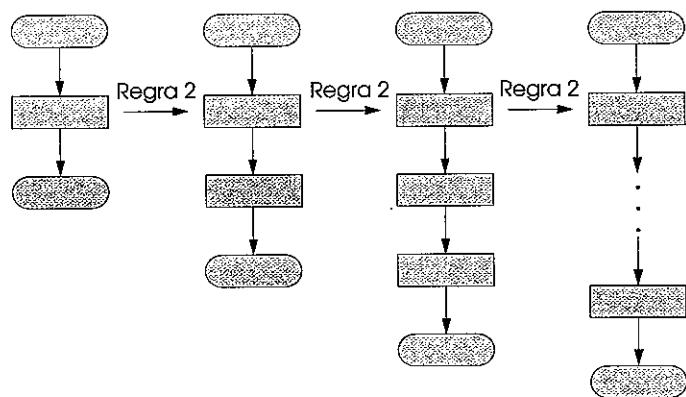


Fig. 5.24 Aplicando repetidamente a Regra 2 da Fig. 5.22 ao fluxograma mais simples.

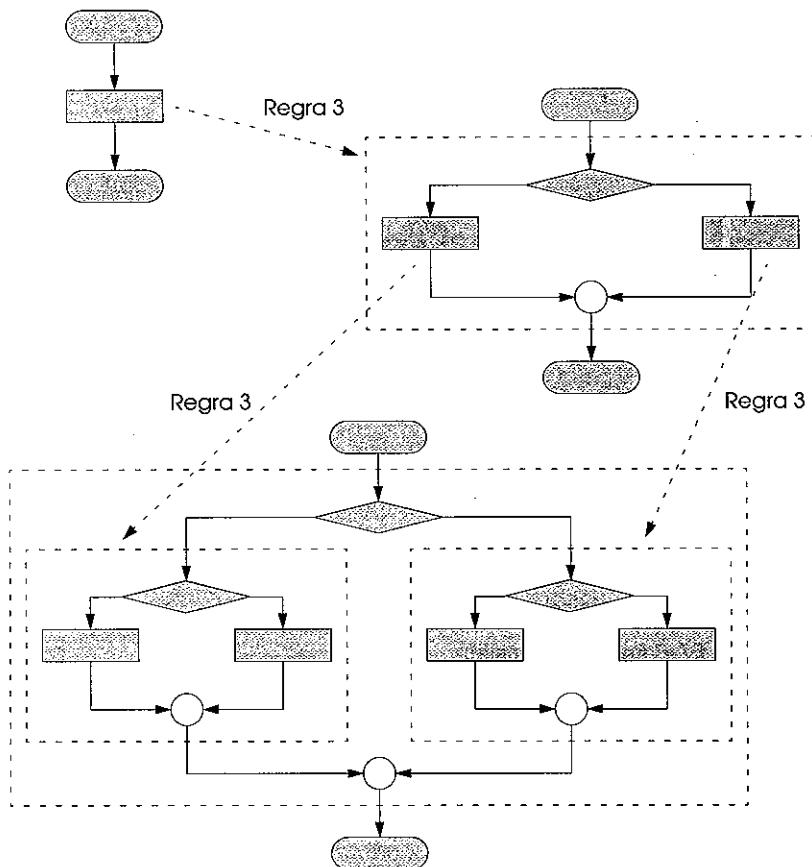


Fig. 5.25 Aplicando a Regra 3 da Fig. 5.22 ao fluxograma mais simples.

A programação estruturada promove a simplicidade. Bohm e Jacopini mostraram que apenas três formas de controle são necessárias – seqüência, seleção e repetição.

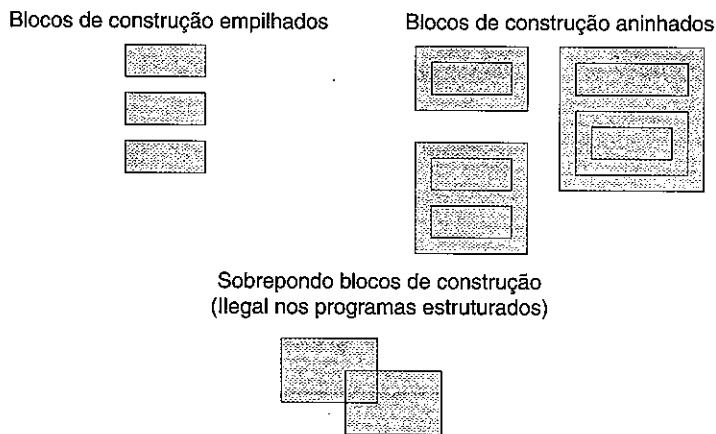


Fig. 5.26 Blocos de construção empilhados, aninhados e sobrepostos.

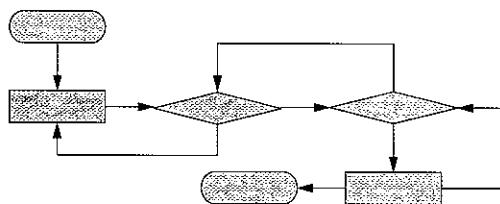


Fig. 5.27 Um fluxograma não-estruturado.

A seqüência é trivial. A seleção é implementada de uma de três maneiras:

- uma estrutura **if** (seleção única);
- uma estrutura **if/else** (seleção dupla);
- uma estrutura **switch** (seleção múltipla).

Na verdade, é simples e direto provar que a estrutura simples **if** é suficiente para fornecer qualquer forma de seleção; tudo que pode ser feito com a estrutura **if/else** e a estrutura **switch** pode ser implementado combinando-se estruturas **if** (embora talvez não tão elegantemente).

A repetição é implementada de uma de três maneiras:

- uma estrutura **while**;
- uma estrutura **do/while**;
- uma estrutura **for**.

É simples provar que a estrutura **while** é suficiente para fornecer qualquer forma de repetição. Tudo que pode ser feito com a estrutura **do/while** e a estrutura **for** pode ser feito com a estrutura **while** (embora talvez não tão elegantemente).

A combinação desses resultados ilustra que qualquer forma de controle que possa ser necessária um dia em um programa Java pode ser expressa em termos de:

- seqüênci;a;
- estrutura **if** (seleção); ou
- estrutura **while** (repetição).

E essas estruturas de controle podem ser combinadas apenas de duas maneiras – empilhamento e aninhamento. De fato, a programação estruturada promove a simplicidade.

Neste capítulo, discutimos como compor programas a partir de estruturas de controle que contêm ações e decisões. No Capítulo 6, apresentamos outra unidade de estrutura de programa chamada de *método*. Aprendemos a compor programas grandes combinando métodos que, por sua vez, são compostos de estruturas de controle. Além disso, discutimos como os métodos promovem a reutilização do *software*. No Capítulo 8, discutimos em mais detalhes outra unidade de estruturação de programa de Java chamada de *classe*. Depois criamos objetos a partir de classes e prosseguimos com nosso tratamento da programação orientada a objetos.

5.11 (Estudo de caso opcional) Pensando em objetos: identificando estados e atividades dos objetos

Na Seção 4.14 de “Pensando em objetos”, determinamos muitos dos atributos de classes necessários para implementar o simulador de elevador e os adicionamos ao diagrama de classes da Fig. 4.18. Nesta seção, mostramos como estes atributos representam o *estado*, ou a condição, de um objeto. Identificamos o conjunto de estados possíveis que nossos objetos podem assumir e discutimos como estes objetos mudam de estado em resposta a *mensagens*. Também discutimos o fluxo de trabalho, ou as *atividades*, que um objeto executa em nossa simulação de elevador.

Diagramas de mapa de estados

Os objetos em um sistema têm *estados*. O estado descreve a condição de um objeto num determinado instante no tempo. Os *diagramas de mapa de estados* (também denominados *diagramas de estados*) nos dão uma maneira de expressar como, e sob quais condições, os objetos em um sistema mudam de estado. Diferentemente dos diagramas de classes e de objetos apresentados em seções de estudo de caso anteriores, os diagramas de mapa de estados modelam o comportamento do sistema.

A Fig. 5.28 é um diagrama simples de mapa de estados, que modela os estados de um objeto da classe **FloorButton** ou da classe **ElevatorButton**. A UML representa cada estado, em um diagrama de mapa de estados, como um *retângulo com cantos arredondados*, com o nome do estado colocado dentro do retângulo. Um *círculo cheio* com uma seta presa a ele indica o estado inicial (neste caso, o estado “Não pressionado”). Repare que este diagrama de mapa de estados modela o atributo **pressed**, do tipo **boolean**, no diagrama de classes da Fig. 4.18. O atributo é inicializado com **false**, ou o estado “Não pressionado”, de acordo com o diagrama de mapa de estados.

As setas indicam *transições* entre estados. O objeto pode passar de um estado para outro em resposta a uma *mensagem*. Por exemplo, as classes **FloorButton** e **ElevatorButton** mudam do estado “Não pressionado” para o estado “Pressionado” em resposta a uma mensagem **buttonPressed** e o atributo **pressed** muda para um valor **true**. O nome da mensagem que causa a transição é escrito junto à linha que corresponde àquela transição (explicamos as mensagens na Seção 7.10 e na Seção 10.22).

Objetos de outras classes, como **Light**, **Elevator** e **Person**, têm diagramas de mapa de estados semelhantes. A classe **Light** tem um estado “ligada” e um estado “desligada” – as transições entre esses estados ocorrem como resultado dos eventos “ligar” e “desligar”, respectivamente. A classe **Elevator** e a classe **Person** têm, cada uma, um estado “se movendo” e um estado “esperando” – as transições entre esses estados ocorrem como resultado dos eventos “iniciar movimento” e “parar movimento”, respectivamente.

Diagramas de atividades

O *diagrama de atividades* é semelhante ao diagrama de mapa de estados, no sentido de que ambos modelam aspectos do comportamento do sistema. Ao contrário de um diagrama de mapa de estados, o diagrama de atividades mo-

dela o fluxo de trabalho de um objeto durante a execução do programa. O diagrama de atividades é um fluxograma que modela as *ações* que o objeto vai executar e em que ordem. O diagrama de atividades na Fig. 5.29 modela as atividades de uma pessoa. O diagrama começa com a pessoa se movendo em direção ao botão do andar. Se a porta está aberta, a pessoa espera que o passageiro atual do elevador (se existe um) saia do elevador¹ e entre nele. Se a porta está fechada, a pessoa pressiona o botão do andar e espera que o elevador abra a porta. Quando a porta se abre, a pessoa espera que o passageiro do elevador saia (se existir um) e depois entra no elevador. A pessoa pressiona o botão do elevador, o que faz o elevador se mover para o outro andar, a menos que o elevador já esteja atendendo àquele andar; a pessoa então espera que as portas abram novamente e sai do elevador depois que as portas abrem.

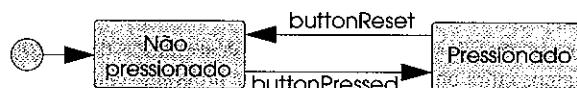


Fig. 5.28 Diagrama de mapa de estados para objetos `FloorButton` e `ElevatorButton`.

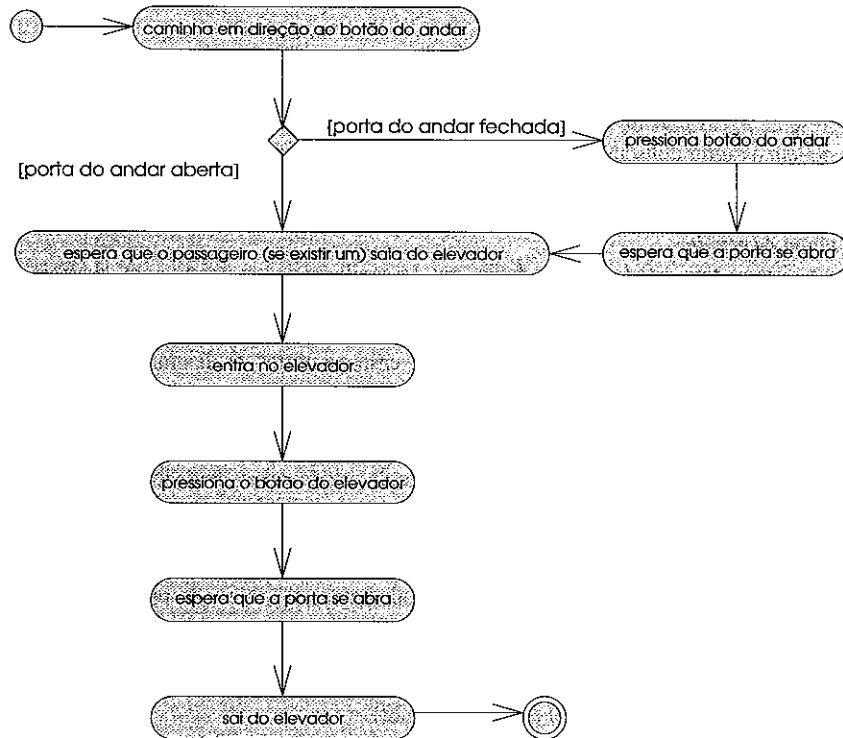


Fig. 5.29 Diagrama de atividades para um objeto `Person`.

¹ Usamos *multithreading* e métodos `synchronized` na Seção 15.12 para garantir que o passageiro que está andando no elevador saia antes de a pessoa que está esperando pelo elevador entrar.

A UML representa atividades nos diagramas de atividades como elipses. O nome da atividade é colocado dentro da elipse. Uma linha cheia com uma seta conecta duas atividades, indicando a ordem na qual as atividades são executadas. Assim como no diagrama de mapa de estados, o *círculo cheio* indica a atividade inicial. Neste caso, a pessoa caminhando em direção ao botão do andar é a atividade inicial. Depois que a pessoa caminha em direção ao botão do andar, o fluxo de atividades chega em um *desvio* (indicado pelo símbolo de um *losango pequeno*). Este ponto determina a próxima atividade com base na *condição de guarda* associada (entre colchetes, acima da transição), que indica que a transição ocorre se esta condição for satisfeita. Por exemplo, na Fig. 5.29, se a porta no andar estiver fechada, a pessoa pressiona o botão do andar, espera até que a porta se abra, espera que o passageiro (se existe um) saia do elevador e, depois, entra no elevador. Entretanto, se a porta do andar estiver aberta, a pessoa espera que o passageiro (se existe um) saia do elevador, depois entra no elevador. Independentemente de a porta estar aberta ou fechada no último símbolo de decisão, a pessoa agora pressiona o botão do elevador (o que faz as portas fecharem e o elevador se movimentar para o outro andar), a pessoa espera que a porta do elevador se abra – quando esta porta se abre, a pessoa sai do elevador. Os diagramas de atividades são semelhantes ao fluxogramas para as estruturas de controle apresentados nos Capítulos 4 e 5 – os dois tipos de diagramas usam losangos para alterar o fluxo de controle entre atividades.

A Fig. 5.30 mostra um diagrama de atividades para o elevador. O diagrama começa quando um botão é pressionado. Se o botão for do elevador, o elevador ajusta **summoned** (chamado) para falsa (explicamos esta variável booleana em seguida), fecha a porta do elevador, se movimenta até o outro andar, desliga o botão do elevador, toca a campainha e abre a porta do elevador. Se o botão for de andar, o próximo desvio determina a próxima transição, dependendo de o elevador estar se movendo ou não. Se o elevador estiver ocioso, o próximo desvio determina qual botão de andar gerou a chamada. Se a chamada foi originada no andar atual em que o elevador está localizado, o ele-

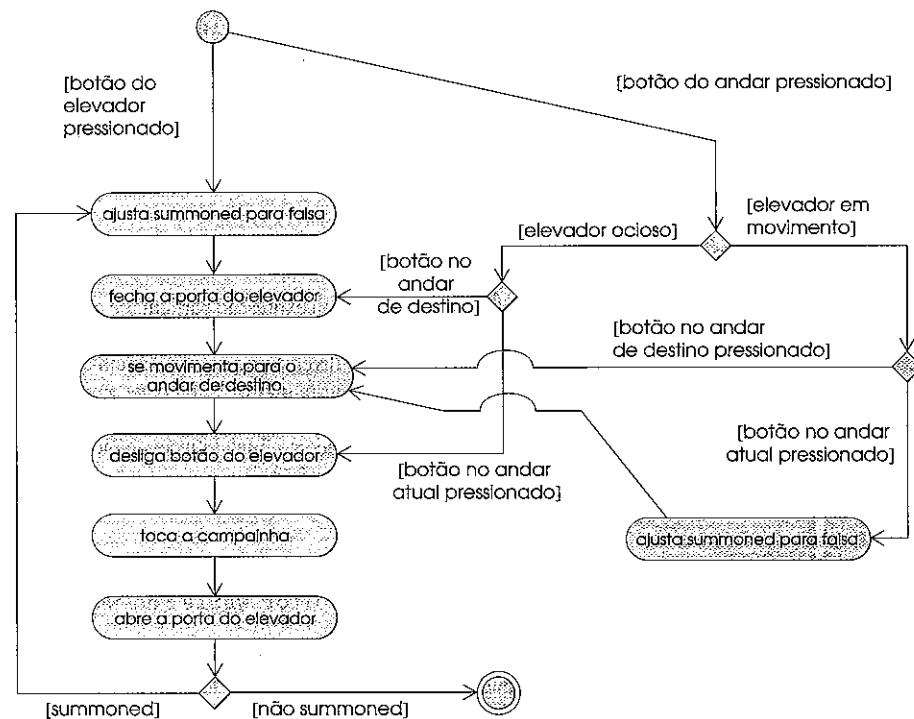


Fig. 5.30 Diagrama de atividades para um objeto `Elevator`.

vador desliga o botão, toca a campainha e abre a porta. Se a chamada se originou no andar oposto, o elevador fecha a porta e se movimenta para o andar oposto (destino), onde o elevador desliga o botão, toca a campainha e abre a porta. Agora pense na atividade se o elevador estiver se movendo. Um desvio separado determina qual botão de andar gerou a chamada. Se a chamada foi originada no andar de destino, o elevador continua se movimentando para aquele andar. Se a chamada se originou no andar do qual o elevador partiu, o elevador continua se movimentando para o andar de destino, mas precisa lembrar de retornar para o andar que chamou. O atributo `summoned`, exibido originalmente na Fig. 4.18, é ajustado para `true`, de modo que o elevador se lembre de retornar para o outro andar depois que abrir a porta.

Demos os primeiros passos para modelar o comportamento do sistema e mostramos como os atributos de um objeto determinam a atividade daquele objeto. Na Seção 6.17 de “Pensando em objetos”, investigamos os comportamentos de todas as classes para dar uma interpretação mais precisa do comportamento do sistema “preenchendo” o comportamento final das classes em nosso diagrama de classes.

Resumo

- A estrutura de repetição `for` trata de todos os detalhes da repetição controlada por contador. O formato geral da estrutura `for` é

```
for (expressão1; expressão2; expressão3)
    instrução
```

onde `expressão1` inicializa a variável de controle do laço, `expressão2` é a condição de continuação do laço e `expressão3` incrementa a variável de controle, de modo que a condição de continuação do laço, em algum momento, se torne falsa.

- `JTextArea` é um componente GUI que é capaz de exibir muitas linhas de texto.
- O método `setText` substitui o texto em uma `JTextArea`. O método `append` adiciona texto ao fim do texto em uma `JTextArea`.
- O método `static getCurrencyInstance` da classe `NumberFormat` devolve um objeto `NumberFormat` que pode formatar valores numéricos como valores em dinheiro. O argumento `Locale.US` indica que os valores em dinheiro devem ser exibidos iniciando com um símbolo de cifrão (\$), devem usar uma casa decimal para separar os dólares e os centavos e usar uma vírgula para delimitar milhares.
- A classe `Locale` oferece constantes que podem ser usadas para personalizar programas para representar valores em dinheiro para outros países, de modo que os formatos de valores em dinheiro sejam exibidos apropriadamente para cada local.
- A classe `NumberFormat` está localizada no pacote `java.text`.
- A classe `Locale` está localizada no pacote `java.util`.
- Um recurso interessante da classe `JOptionPane` é que a mensagem que ela exibe com `showMessageDialog` pode ser um `String` ou um componente GUI, como uma `JTextArea`.
- A estrutura `switch` trata uma série de condições nas quais uma variável ou expressão particular é comparada com valores que ela pode assumir e diferentes ações são tomadas. Na maioria dos programas, é necessário incluir uma instrução `break` depois das instruções para cada `case`. Para fazer com que diversos `cases` executem as mesmas instruções, basta listar os rótulos de `case` juntos, antes das instruções. A estrutura `switch` só pode comparar com expressões constantes integrais.
- A estrutura de repetição `do/while` testa a condição de continuação do laço no final do laço, de modo que o corpo do laço será executado pelo menos uma vez. O formato para a estrutura `do/while` é

```
do {
    instrução
} while (condição);
```

- A instrução `break`, quando executada em uma das estruturas de repetição (`for`, `while` e `do/while`), causa a saída imediata da estrutura.
- A instrução `continue`, quando executada em uma das estruturas de repetição (`for`, `while` e `do/while`), pula qualquer instrução restante no corpo da estrutura e prossegue com o teste para a próxima iteração do laço.
- Para sair de um conjunto aninhado de estruturas, use a instrução rotulada `break`. Esta instrução, quando executada em uma estrutura `while`, `for`, `do/while` ou `switch`, provoca a saída imediata daquela estrutura e qualquer quantida-

de de estruturas de repetição que a incluem; a execução do programa é retomada na primeira instrução após o bloco rotulado que a envolve.

- A instrução rotulada **continue**, quando executada em uma estrutura de repetição (**while**, **for** ou **do/while**), pula as instruções restantes no corpo daquela estrutura e em qualquer quantidade de estruturas de repetição que a incluem e prossegue para a próxima iteração da estrutura de repetição rotulada que a envolve.
- Podem-se utilizar os operadores lógicos para formar condições complexas combinando condições. Os operadores lógicos são **&&**, **&**, **|**, **||**, **^** e **!**, que significam E lógico, E lógico booleano, OU lógico, OU lógico booleano inclusivo, OU lógico booleano exclusivo e NÃO lógico (negação), respectivamente.
- A classe **JScrollPane** fornece a um componente GUI a funcionalidade de rolagem.

Terminologia

<i>avaliação em curto-circuito</i>	<i>estruturas de controle empilhadas</i>
<i>barra de rolagem</i>	<i>estruturas de repetição</i>
<i>bloco rotulado</i>	<i>instrução rotulada break</i>
break	<i>instrução rotulada composta</i>
<i>caixa de rolagem</i>	<i>instrução rotulada continue</i>
<i>caixa de uma barra de rolagem</i>	<i>laço infinito</i>
<i>caso default em um switch</i>	Locale.US
classe JScrollPane	long
classe JTextArea	<i>método append da classe JTextArea</i>
classe Locale	<i>negação lógica (!)</i>
classe NumberFormat	<i>operador !</i>
<i>condição de continuação do laço</i>	<i>operador </i>
continue	<i>operadores lógicos</i>
<i>E lógico (&&)</i>	<i>OU lógico ()</i>
<i>E lógico booleano (&) operador &&</i>	<i>OU lógico booleano exclusivo (^)</i>
<i>erro por um (off-by-one)</i>	<i>OU lógico booleano inclusivo ()</i>
<i>estrutura de repetição do/while</i>	<i>pacote java.text</i>
<i>estrutura de repetição for</i>	<i>pacote java.util</i>
<i>estrutura de repetição rotulada</i>	<i>repetição controlada por contador</i>
<i>estrutura de repetição while</i>	<i>repetição definida</i>
<i>estrutura de seleção switch</i>	<i>rótulo case</i>
<i>estruturas de controle aninhadas</i>	<i>seleção múltipla</i>
<i>estruturas de controle de entrada única/saída única</i>	

Exercícios de auto-revisão

5.1 Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.

- O caso **default** é necessário na estrutura de seleção **switch**.
- A instrução **break** é necessária no caso **default** de uma estrutura de seleção **switch**.
- A expressão $(x > y \&\& a < b)$ é verdadeira se $x > y$ for verdadeira ou $a < b$ for verdadeira.
- A expressão que contém o operador **||** é verdadeira se um ou ambos de seus operandos forem verdadeiros.

5.2 Escreva uma instrução em Java ou um conjunto de instruções em Java para realizar cada uma das tarefas seguintes:

- Somar os inteiros ímpares entre 1 e 99 utilizando uma estrutura **for**. Suponha que as variáveis inteiros **sum** e **count** foram declaradas.
- Calcular o valor de 2.5 elevado à potência 3 com o método **pow**.
- Imprimir os inteiros de 1 a 20 com o laço **while** e a variável contadora **x**. Suponha que a variável **x** foi declarada, mas não foi inicializada. Imprima apenas cinco inteiros por linha. [Dica: utilize o cálculo $x \% 5$. Quando o valor desta expressão for 0, imprima um caractere de nova linha; caso contrário, imprima um caractere de tabulação.] Suponha que seja um aplicativo; utilize o método **System.out.println()** para gerar como saída o caractere de nova linha e utilize o método **System.out.print('\'t')** para gerar como saída o caractere de tabulação.]
- Repete o Exercício 5.2 (c) com uma estrutura **for**.

5.3 Localize o erro em cada um dos seguintes segmentos de código e explique como corrigi-los.

- a) `x = 1;`


```
while ( x <= 10 );
      x++;
}
```

 - b) `for (y = .1; y != 1.0; y += .1)
 System.out.println(y);`
 - c) `switch (n) {
 case 1:
 System.out.println("The number is 1");
 case 2:
 System.out.println("The number is 2");
 break;
 default:
 System.out.println("The number is not 1 or 2");
 break;
}`
 - d) O código seguinte deve imprimir os valores 1 a 10.
`n = 1;`
-
- ```
while (n < 10)
 System.out.println(n++);
```

### **Respostas aos exercícios de auto-revisão**

**5.1** a) Falsa. O caso `default` é opcional. Se nenhuma ação `default` for necessária, não há necessidade de um caso `default`. b) Falsa. Utiliza-se a instrução `break` para sair da estrutura `switch`. A instrução `break` não é necessária para o último caso em uma estrutura `switch`. c) Falsa. Ambas as expressões relacionais devem ser verdadeiras a fim de que a expressão inteira seja verdadeira ao se utilizar o operador `&&`. d) Verdadeira.

**5.2** As respostas ao Exercício 5.2 são as seguintes:

- a) `sum = 0;
for ( count = 1; count <= 99; count += 2 )
 sum += count;`
  - b) `Math.pow( 2.5, 3 )`
  - c) `x = 1;`
- 
- ```
while ( x <= 20 ) {
    System.out.print( x );

    if ( x % 5 == 0 )
        System.out.println();
    else
        System.out.print( '\t' );

    ++x;
}
```
- d) `for (x = 1; x <= 20; x++) {
 System.out.print(x);

 if (x % 5 == 0)
 System.out.println();`

```

        else
            System.out.print( '\t' );
    }
}

```

ou

```

for ( x = 1; x <= 20; x++ )

if ( x % 5 == 0 )
    System.out.println( x );
else
    System.out.print( x + "\t" );
}

```

- 5.3** As respostas para o Exercício 5.3 são as seguintes:
- Erro: o ponto-e-vírgula depois do cabeçalho de **while** causa um laço infinito e está faltando uma chave à esquerda.
Correção: substitua o ponto-e-vírgula por uma { ou remova o ; e a }.
 - Erro: utilizar um número em ponto flutuante para controlar uma estrutura de repetição **for** pode não funcionar, porque os números em ponto flutuante são representados apenas aproximadamente pela maioria dos computadores.
Correção: utilize um inteiro e realize o cálculo adequado a fim de obter os valores que você deseja:

```

for ( y = 1; y != 10; y++ )
    System.out.println( float ) y / 10 );
}

```
 - Erro: falta a instrução **break** no fim das instruções para o primeiro **case**.
Correção: adicione uma instrução **break** no fim das instruções para o primeiro **case**. Note que esta omissão não é necessariamente um erro, se o programador quiser que a instrução do **case 2**: seja executada sempre que o **case 1**: for executado.
 - Erro: operador relacional impróprio usado na condição de repetição de continuação do **while**.
Correção: utilize <= em vez de <, ou altere 10 para 11.

Exercícios

- 5.4** Localize o(s) erro(s) em cada um dos seguintes segmentos de código:
- For** (x = 100, x >= 1, x++)

```

System.out.println( x );
}

```
 - O código a seguir deve imprimir se o inteiro **value** é par ou ímpar:

```

switch ( value % 2 ) {

    case 0:
        System.out.println( "Even integer" );

    case 1:
        System.out.println( "Odd integer" );
}
}

```
 - O código seguinte deve gerar como saída os inteiros ímpares de 19 a 1:

```

for ( x = 19; x >= 1; x += 2 )
    System.out.println( x );
}

```
 - O código seguinte deve gerar como saída os inteiros pares de 2 a 100:

```

counter = 2;

do {
    System.out.println( counter );
    counter += 2;
} while ( counter < 100 );
}

```

5.5 O que faz o seguinte programa?

```

1 public class Printing {
2
3     public static void main( String args[] ) {
4
5         for ( int i = 1; i <= 10; i++ ) {
6
7             for ( int j = 1; j <= 5; j++ )
8                 System.out.print( '@' );
9
10            System.out.println();
11
12        }
13
14    }
15
16 }
```

5.6 Escreva um aplicativo que localiza o menor de vários inteiros. Suponha que o primeiro valor lido especifica o número de valores a serem digitados pelo usuário.

5.7 Escreva um aplicativo que calcula o produto dos inteiros ímpares de 1 a 15 e depois exibe os resultados em um diálogo de mensagem.

5.8 O método *fatorial* é freqüentemente utilizado em problemas de probabilidade. O fatorial de um inteiro positivo n (escrito $n!$ e pronunciado como “fatorial de n ”) é igual ao produto dos inteiros positivos de 1 a n . Escreva um aplicativo que calcula os fatoriais dos inteiros de 1 a 5. Exiba os resultados em formato de tabela em uma *JTextArea* que é exibida em um diálogo de mensagem. Que dificuldade poderia impedir-lo de calcular o fatorial de 20?

5.9 Modifique o programa de juros compostos da Fig. 5.6 para repetir seus passos para taxas de juros de 5, 6, 7, 8, 9 e 10%. Utilize um laço *for* para variar a taxa de juros. Adicione funcionalidade de rolagem à *JTextArea* de modo que seja possível rolar o texto para ver a saída.

5.10 Escreva um aplicativo que exibe os seguintes padrões separadamente um embaixo do outro. Utilize laços *for* para gerar os padrões. Todos os asteriscos (*) devem ser impressos por uma única instrução na forma `System.out.print('*');` (Essa instrução faz com que os asteriscos sejam impressos lado a lado.) Uma instrução na forma `System.out.println();` pode ser utilizada para posicionar na próxima linha. Uma instrução na forma `System.out.print(' ');` pode ser utilizada para exibir um espaço para os dois últimos padrões. Não deve haver outras instruções de saída no programa. (*Dica:* os últimos dois padrões exigem que cada linha inicie com um número apropriado de espaços em branco.)

(a)	(b)	(c)	(d)
*	*****	*****	*
**	*****	*****	**
***	*****	*****	***
****	*****	*****	****
*****	*****	*****	*****
*****	*****	*****	*****
*****	*****	*****	*****
*****	*****	*****	*****
*****	***	***	*****
*****	**	**	*****
*****	*	*	*****

5.11 Uma aplicação interessante dos computadores é desenhar gráficos e gráficos de barras (às vezes chamados de “histogramas”). Escreva um *applet* que lê cinco números, cada um entre 1 e 30. Para cada número lido, seu programa deve desenhar uma linha que contém aquele número de asteriscos adjacentes. Por exemplo, se seu programa lê o número sete, ele deve imprimir `*****`.

5.12 Uma empresa de pedidos pelo correio vende cinco produtos diferentes cujos preços de varejo são: produto 1, \$2,98; produto 2, \$4,50; produto 3, \$9,98; produto 4, \$4,49; e produto 5, \$6,87. Escreva um aplicativo que leia uma série de pares de números como segue:

- Número do produto;
- Quantidade vendida em um dia.

O programa deve utilizar uma estrutura `switch` para ajudar a determinar o preço de varejo de cada produto. Ele deve calcular e exibir o valor total no varejo de todos os produtos vendidos na semana passada. Utilize um `TextField` para ler o número de produto digitado pelo usuário. Utilize um laço controlado por sentinelas para determinar quando o programa deve parar de repetir o laço e exibir os resultados finais.

5.13 Modifique o programa da Fig. 5.6 para utilizar apenas inteiros para calcular os juros compostos. [Dica: trate todas as quantidades em dinheiro como números inteiros em centavos. Depois “divida” o resultado em sua parte inteira e sua parte em centavos utilizando as operações de divisão e módulo, respectivamente. Insira um ponto entre a parte em dólares e a parte em centavos.]

5.14 Suponha que $i = 1$, $j = 2$, $k = 3$ e $m = 2$. O que imprime cada uma das seguintes instruções?

- `System.out.println(i == 1);`
- `System.out.println(j == 3);`
- `System.out.println(i >= 1 && j < 4);`
- `System.out.println(m <= 99 & k < m);`
- `System.out.println(j >= i || k == m);`
- `System.out.println(k + m < j | 3 - j >= k);`
- `System.out.println(!(k > m));`

5.15 Escreva um aplicativo que imprime uma tabela dos equivalentes em binário, octal e hexadecimal dos números decimais no intervalo 1 a 256. Se você não estiver familiarizado com esses sistemas de numeração, leia o Apêndice E primeiro. Coloque os resultados em uma `JTextArea` com funcionalidade de rolagem. Exiba a `JTextArea` em um diálogo de mensagem.

5.16 Calcule o valor de π com a série infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

Imprima uma tabela que mostra o valor de π aproximado por um termo dessa série, por dois termos, por três termos, etc. Quantos termos dessa série você teria de utilizar antes de conseguir 3,14? 3,141? 3,1415? 3,14159?

5.17 (*Triplas de Pitágoras*) Um triângulo retângulo pode ter lados cujos comprimentos são todos inteiros. O conjunto de três valores inteiros para os comprimentos dos lados de um triângulo retângulo é chamado de tripla de Pitágoras. O comprimento dos três lados devem satisfazer a relação de que a soma dos quadrados de dois dos lados seja igual ao quadrado da hipotenusa. Escreva um aplicativo para localizar todas as triplas de Pitágoras para `side1`, `side2` e `hypotenuse`, todos não maiores que 500. Utilize um laço `for` triplicamente aninhado que testa todas as possibilidades. Esse método é um exemplo de computação de “força bruta”. Você aprenderá em cursos mais avançados de ciência da computação que há muitos problemas interessantes para os quais não há abordagem algorítmica conhecida a não ser utilizar a pura força bruta.

5.18 Modifique o Exercício 5.10 para combinar seu código de quatro triângulos de asteriscos separados em um único aplicativo que imprime todos os quatro padrões lado a lado fazendo uso inteligente de laços aninhados `for`.

*	*****	*****	*
**	*****	*****	**
***	*****	*****	***
****	*****	*****	****
*****	*****	*****	*****
*****	***	***	*****
*****	**	**	*****
*****	*	*	*****

5.19 (*Leis de De Morgan*) Neste capítulo, discutimos os operadores lógicos `&&`, `&`, `||`, `|`, `^` e `!`. As Leis de De Morgan às vezes podem expressar uma expressão lógica da forma mais conveniente para nós. Essas leis afirmam que a expressão `!(condição1 && condição2)` é logicamente equivalente à expressão `(!condição1 || !condição2)`. Além disso, a expressão `!(condição1 || condição2)` é logicamente equivalente à expressão `(!condição1 && !condição2)`. Utilize as Leis de De Morgan para escrever expressões equivalentes para cada uma das expressões seguintes e depois escreva um programa para mostrar que a expressão original e a nova expressão em cada caso são equivalentes:

- `!(x < 5) && !(y >= 7)`
- `!(a == b) || !(g != 5)`
- `!(x <= 8) && (y > 4)`
- `!(i > 4) || (j <= 6)`

5.20 Escreva um aplicativo que imprime a seguinte forma de losango. Você pode utilizar instruções de saída que imprimem um único asterisco (*), um único espaço ou um único caractere de nova linha. Maximize sua utilização de repetição (com estruturas aninhadas `for`) e minimize o número de instruções de saída.

```

*
 ***
 *****
 ******
 *****
 ****
 ***
 *
```

5.21 Modifique o programa que você escreveu no Exercício 5.20 para ler um número ímpar no intervalo 1 a 19 que especifica o número de linhas no losango. O programa deve exibir um losango do tamanho apropriado.

5.22 Uma crítica à instrução `break` e à instrução `continue` é que elas não são estruturadas. De fato, as instruções `break` e instruções `continue` podem sempre ser substituídas por instruções estruturadas, embora fazer isso possa ser inconveniente. Descreva de maneira geral como você removeria qualquer instrução `break` de um laço em um programa e substituiria essa instrução por alguma equivalente estruturada. [Dica: a instrução `break` deixa um laço a partir de dentro do corpo do laço. A outra maneira de deixar um laço é reprovando o teste de continuação do laço. Considere a possibilidade de utilizar no teste de continuação do laço um segundo teste, que indica “saída prévia por causa de uma condição ‘break’”.] Utilize a técnica que você desenvolveu aqui para remover a instrução `break` do programa da Fig. 5.11.

5.23 O que faz o seguinte segmento de programa?

```

for ( i = 1; i <= 5; i++ ) {
    for ( j = 1; j <= 3; j++ ) {
        for ( k = 1; k <= 4; k++ )
            System.out.print( '*' );
        System.out.println();
    }
    System.out.println();
}
```

5.24 Descreva de maneira geral como você removeria qualquer instrução `continue` de um laço em um programa e substituiria essa instrução por alguma equivalente estruturada. Utilize a técnica que você desenvolveu aqui para remover a instrução `continue` do programa da Fig. 5.12.

5.25 (A canção “The Twelve Days of Christmas”*) Escreva um aplicativo que utilize repetição e estruturas **switch** para imprimir a canção “The Twelve Days of Christmas”. Deve-se utilizar uma estrutura **switch** para imprimir o dia (isto é, “First”, “Second”, etc.). Deve-se utilizar uma estrutura **switch** separada para imprimir o restante de cada verso.

* N. do T. Popular canção infantil natalina nos países de língua inglesa. A seguir é fornecida uma versão tabulada para facilitar a visualização da estrutura da canção. Na Internet pode-se ouvir a canção, conhecer sua história, encontrar interpretações sobre seus significados religiosos ocultos e descobrir diversas versões humorísticas, seguindo os resultados da pesquisa do nome da canção em sistemas de busca como o Google e o Yahoo.

On the first day of Christmas, my true love gave to me	a partridge in a pear tree.
On the second day of Christmas, my true love gave to me and a partridge in a pear tree.	two turtle doves,
On the third day of Christmas, my true love gave to me two turtle doves, and a partridge in a pear tree.	three french hens,
On the fourth day of Christmas, my true love gave to me three french hens, two turtle doves, and a partridge in a pear tree.	four calling birds,
On the fifth day of Christmas, my true love gave to me four calling birds, three french hens, two turtle doves, and a partridge in a pear tree.	five gold rings,
On the sixth day of Christmas, my true love gave to me five gold rings, four calling birds, three french hens, two turtle doves, and a partridge in a pear tree.	six geese a-laying,
On the seventh day of Christmas, my true love gave to me six geese a-laying, five gold rings, four calling birds, three french hens, two turtle doves, and a partridge in a pear tree.	seven swans a-swimming,
On the eighth day of Christmas, my true love gave to me seven swans a-swimming, six geese a-laying, five gold rings, four calling birds, three french hens, two turtle doves, and a partridge in a pear tree.	eight maids a-milking,
On the ninth day of Christmas, my true love gave to me eight maids a-milking, seven swans a-swimming, six geese a-laying, five gold rings, four calling rings, three french hens, two turtle doves, and a partridge in a pear tree.	nine ladies waiting,

On the **tenth** day of Christmas, my true love gave to me **ten lords a-leaping,**
nine ladies waiting,
eight maids a-milking,
seven swans a-swimming,
six geese a-laying,
five gold rings,
four calling birds,
three french hens,
two turtle doves,
and a partridge in a pear tree.

On the **eleventh** day of Christmas, my true love gave to me **eleven pipers piping,**
ten lords a-leaping,
nine ladies waiting,
eight maids a-milking,
seven swans a-swimming,
six geese a-laying,
five gold rings,
four calling birds,
three french hens,
two turtle doves,
and a partridge in a pear tree.

On the **twelfth** day of Christmas, my true love gave to me **twelve drummers drumming,**
eleven pipers piping,
ten lords a-leaping,
nine ladies waiting,
eight maids a-milking,
seven swans a-swimming,
six geese a-laying,
five gold rings,
four calling birds,
three french hens,
two turtle doves,
and a partridge in a pear tree.

6

Métodos

Objetivos

- Entender como construir programas modularmente a partir de pequenos pedaços denominados *métodos*.
- Apresentar os métodos comuns de matemática disponíveis na Java API.
- Ser capaz de criar novos métodos.
- Entender os mecanismos utilizados para passar informações entre métodos.
- Apresentar técnicas de simulação com geração de números aleatórios.
- Entender como a visibilidade dos identificadores é limitada a regiões específicas dos programas.
- Entender como escrever e utilizar métodos que chamam a si próprios.

A forma sempre segue a função.

Louis Henri Sullivan

E pluribus unum.

(Um composto de muitos.)

Virgílio

Chama o dia de ontem, faze que o tempo atraso retorne.

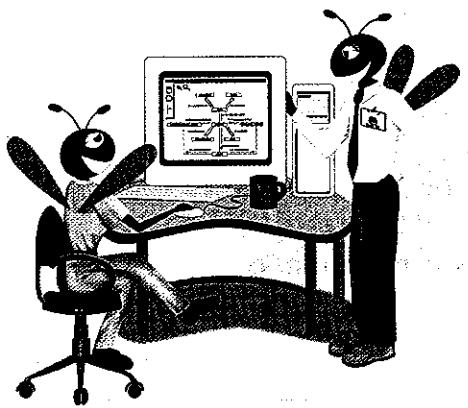
William Shakespeare, Ricardo II

Chamem-me Ismael.

Herman Melville, Moby Dick

Quando você me chamar assim, sorria.

Owen Wister



Sumário do capítulo

- 6.1 Introdução
- 6.2 Módulos de programas em Java
- 6.3 Os métodos da classe Math
- 6.4 Métodos
- 6.5 Definições de métodos
- 6.6 Promocão de argumentos
- 6.7 Pacotes da Java API
- 6.8 Geração de números aleatórios
- 6.9 Exemplo: um jogo de azar
- 6.10 Duração dos identificadores
- 6.11 Regras de escopo
- 6.12 Recursão
- 6.13 Exemplo que utiliza recursão: a série de Fibonacci
- 6.14 Recursão versus iteração
- 6.15 Sobreulação de métodos
- 6.16 Métodos da classe JOptionPane
- 6.17 (Estudo de caso opcional) Pensando em objetos: identificando operações de classes

Resumo, termômetro, exercícios de auto-revisão e respostas aos exercícios de auto-revisão.
Exercícios.

6.1 Introdução

A maioria dos programas de computador que resolvem problemas do mundo real é muito maior que os programas apresentados nos primeiros capítulos deste texto. A experiência mostrou que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pedaços pequenos e simples, ou *módulos*. Essa técnica se chama *dividir para conquistar*. Este capítulo descreve muitos recursos fundamentais da linguagem Java que facilitam o projeto, a implementação, a operação e a manutenção de programas grandes.

6.2 Módulos de programas em Java

Os módulos em Java são chamados de *métodos* e *classes*. Os programas Java são escritos combinando-se novos métodos e classes que o programador escreve com métodos e classes “pré-empacotados” disponíveis na *Java API* (também conhecida como *biblioteca de classes Java*) e em vários outras bibliotecas de métodos e classes. Neste capítulo, concentrarmo-nos nos métodos; começamos a discutir as classes em detalhes no Capítulo 8.

A Java API fornece uma rica coleção de classes e métodos para realizar cálculos matemáticos comuns, manipulações de *strings*, manipulações de caracteres, operações de entrada/saída, verificação de erros e muitas outras operações úteis. Esse conjunto de módulos facilita o trabalho do programador, porque os módulos fornecem muitos dos recursos de que os programadores precisam. Os métodos da Java API são fornecidos como parte do Java 2 Developer’s Kit (J2SDK).



Boa prática de programação 6.1

Familiarize-se com a rica coleção de classes e métodos na Java API e com as ricas coleções de classes disponíveis em várias bibliotecas de classes.



Observação de engenharia de software 6.1

Evite reinventar a roda. Quando possível, utilize classes e métodos da Java API em vez de escrever novas classes e métodos. Isso reduz o tempo de desenvolvimento de programas e evita a inserção de novos erros.

**Dica de desempenho 6.1**

Não tente reescrever as classes e os métodos existentes na Java API para torná-los mais eficientes. Você normalmente não será capaz de aumentar o desempenho dessas classes e métodos.

O programador pode escrever métodos para definir tarefas específicas que um programa pode utilizar muitas vezes durante sua execução. Trata-se dos *métodos definidos pelo programador*. As instruções reais que definem os métodos são escritas somente uma vez e são ocultadas de outros métodos.

Um método é *invocado* (isto é, chamado para realizar sua tarefa designada) por uma *chamada de método*. A chamada de método especifica o nome do método e fornece informações (como *argumentos*) de que o método chamado precisa para fazer sua tarefa. Quando a chamada do método termina, o método ou devolve um resultado para o *método que chamou* (ou *chamador*) ou simplesmente devolve o controle para o método que chamou. Uma analogia comum para isso é a forma hierárquica de gerenciamento. O patrão (o *método que chama*, ou *chamador*) pede para o trabalhador (o *método chamado*) realizar uma tarefa e relatar a ele (i. e., *devolver*) os resultados quando a tarefa for completada. O método do patrão não tem conhecimento sobre *como* o método do trabalhador realiza as tarefas a ele designadas. O trabalhador também pode chamar outros métodos trabalhadores e o patrão não terá conhecimento dessa ocorrência. Logo veremos como essa “ocultação” de detalhes da implementação promove a boa engenharia de *software*. A Fig. 6.1 mostra o método **patrão** se comunicando com vários métodos trabalhadores de uma maneira hierárquica. Observe que **trabalhador1** atua como um método de “patrão” para **trabalhador4** e **trabalhador5**. Os relacionamentos entre métodos também podem ser diferentes da estrutura hierárquica mostrada nessa figura.

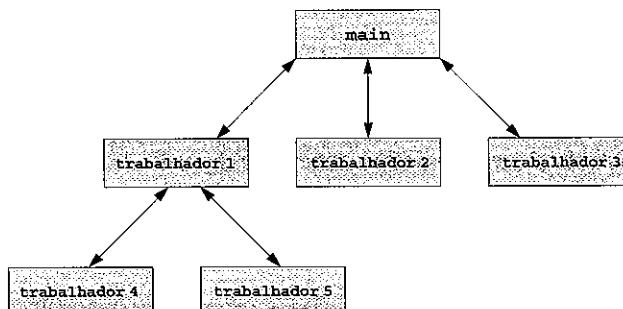


Fig. 6.1 Relacionamento hierárquico método do trabalhador/método do patrão.

6.3 Os métodos da classe Math

Os métodos da classe **Math** permitem realizar certos cálculos matemáticos comuns. Utilizamos vários métodos da classe **Math** aqui para apresentar o conceito de métodos. Ao longo do livro, discutiremos muitos outros métodos das classes da Java API.

Os métodos são chamados escrevendo-se o nome do método, depois um parêntese esquerdo, depois do *argumento* (ou uma lista de argumentos separados por vírgulas) do método, e depois um parêntese direito. Por exemplo, o programador que deseja calcular a raiz quadrada de **900.0** poderia escrever

```
Math.sqrt( 900.0 )
```

Quando essa instrução é executada, ela chama o método **static sqrt** da classe **Math** para calcular a raiz quadrada do número contido entre os parênteses (**900.0**). O número **900.0** é o *argumento* do método **sqrt**. A expressão precedente é avaliada como **30.0**. O método **sqrt** recebe um argumento do tipo **double** e devolve um resultado do tipo **double**. Observe que todos os métodos da classe **Math** são **static**; portanto, eles são invoca-

dos precedendo-se o nome do método com o nome da classe **Math** e um operador ponto (.). Para gerar como saída na janela de comando o valor da chamada de método precedente, o programador poderia escrever

```
System.out.println( Math.sqrt( 900.0 ) );
```

Nesta instrução, o valor que **sqrt** devolve se torna o argumento para o método **println**.



Observação de engenharia de software 6.2

Não é necessário importar a classe Math para utilizar seus métodos. A classe Math faz parte do pacote java.lang que é importado automaticamente pelo compilador.



Erro comum de programação 6.1

Esquecer de invocar um método da classe Math que precede o nome do método pelo nome da classe Math e um operador ponto (.) resulta em um erro de sintaxe.

Os argumentos de um método podem ser constantes, variáveis ou expressões. Se **c1 = 13.0**, **d = 3.0** e **f = 4.0**, então a instrução

```
System.out.println( Math.sqrt( c1 + d * f ) );
```

calcula e imprime a raiz quadrada de $13.0 + 3.0 * 4.0 = 25.0$, a saber 5.0.

Alguns métodos da classe **Math** são resumidos na Fig. 6.2. Na figura, as variáveis **x** e **y** são do tipo **double**. A classe **Math** também define duas constantes matemáticas comumente utilizadas – **Math.PI** e **Math.E**. A constante **Math.PI** (3,14159265358979323846) da classe **Math** é a relação entre a circunferência de um círculo e o seu diâmetro. A constante **Math.E** (2,7182818284590452354) é o valor da base para logaritmos naturais (calculados com o método **static log** da classe **Math**).

6.4 Métodos

Os métodos permitem modularizar um programa. As variáveis declaradas em definições de métodos são *variáveis locais* – só o método que as define sabe que elas existem. A maioria dos métodos tem uma lista de *parâmetros* que fornece o meio de passar informações entre os métodos através de chamadas de métodos. Os parâmetros de um método também são variáveis locais.

Método	Descrição	Exemplo
abs(x)	valor absoluto de x (esse método também tem versões para valores float , int e long)	abs(23.7) é 23.7 abs(0.0) é 0.0 abs(-23.7) é 23.7
ceil(x)	arredonda x para o menor inteiro não menor que x	ceil(9.2) é 10.0 ceil(-9.8) é -9.0
cos(x)	co-seno trigonométrico de x (x em radianos)	cos(0.0) é 1.0
exp(x)	método exponencial e^x	exp(1.0) é 2.71828 exp(2.0) é 7.38906
floor(x)	arredonda x para o maior inteiro não maior que x	floor(9.2) é 9.0 floor(-9.8) é -10.0
log(x)	logaritmo natural de x (base e)	log(2.718282) é 1.0 log(7.389056) é 2.0

Fig. 6.2 Métodos da classe **Math** (parte 1 de 2).

Método	Descrição	Exemplo
<code>max(x, y)</code>	maior valor entre x e y (esse método também tem versões para valores <code>float</code> , <code>int</code> e <code>long</code>)	<code>max(2.3, 12.7)</code> é 12.7 <code>max(-2.3, -12.7)</code> é -2.3
<code>min(x, y)</code>	menor valor entre x e y (esse método também tem versões para valores <code>float</code> , <code>int</code> e <code>long</code>)	<code>min(2.3, 12.7)</code> é 2.3 <code>min(-2.3, -12.7)</code> é -12.7
<code>pow(x, y)</code>	x elevado à potência y (x^y)	<code>pow(2.0, 7.0)</code> é 128.0 <code>pow(9.0, .5)</code> é 3.0
<code>sin(x)</code>	seno trigonométrico de x (x em radianos)	<code>sin(0.0)</code> é 0.0
<code>sqrt(x)</code>	raiz quadrada de x	<code>sqrt(900.0)</code> é 30.0 <code>sqrt(9.0)</code> é 3.0
<code>tan(x)</code>	tangente trigonométrica de x (x em radianos)	<code>tan(0.0)</code> é 0.0

Fig. 6.2 Métodos da classe `Math` (parte 2 de 2).

Há várias motivações para modularizar um programa com métodos. A abordagem de dividir para conquistar deixa o desenvolvimento de programas mais gerenciável. Outra motivação é a *capacidade de reutilização de software* – utilizar métodos existentes como blocos de construção para criar novos programas. Com boas convenções de atribuição de nomes e definição de métodos, você pode criar programas a partir de métodos padronizados em vez de construí-los com código personalizado. Por exemplo, não precisamos definir como converter `Strings` em inteiros e em números de ponto flutuante – Java já nos oferece esses métodos na classe `Integer (parseInt)` e na classe `Double (parseDouble)`. Uma terceira motivação é evitar a repetição de código em um programa. Empacotar o código como um método permite que o código seja executado a partir de diversos pontos de um programa simplesmente chamando-se o método.



Observação de engenharia de software 6.3

Para incentivar a reutilização de software, cada método deve se limitar a executar uma tarefa única e bem-definida e o nome do método deve efetivamente expressar essa tarefa.



Observação de engenharia de software 6.4

Se você não conseguir escolher um nome conciso que expresse o que o método faz, é possível que seu método esteja tentando realizar tarefas demais. Em geral, é melhor dividir esse método em vários métodos menores.

6.5 Definições de métodos

Os programas apresentados até agora consistiam em uma definição de classe que continha pelo menos uma definição de método que chamava métodos da Java API para realizar suas tarefas. Vamos analisar agora como os programadores escrevem seus próprios métodos personalizados. Para simplificar, enquanto não discutirmos mais detalhes de definições de classes, no Capítulo 8, usamos *applets* para todos os programas que contêm duas ou mais definições de métodos.

Considere um *applet* que utiliza um método `square` (invocado a partir do método `init` do *applet*) para calcular os quadrados dos inteiros de 1 a 10 (Fig. 6.3).

Quando o *applet* começa a ser executado, o contêiner de *applets* chama o método `init` do *applet*. A linha 16 declara a referência `outputArea` da classe `JTextArea` e a inicializa com um novo objeto `JTextArea`. Essa `JTextArea` irá exibir os resultados do programa.

Esse programa é o primeiro em que exibimos um componente GUI em um *applet*. A área de exibição na tela para um `JApplet` tem um *painel de conteúdo* ao qual os componentes GUI devem ser anexados, a fim de poderem ser exibidos durante a execução. O painel de conteúdo é um objeto da classe `Container` do pacote `java.awt`. Essa classe foi importada na linha 5 para utilização no *applet*. A linha 19 declara a referência `container` da classe `Container` e atribui a ela o resultado de uma chamada ao método `getContentPane` – um dos muitos métodos que nossa classe `SquareInt` herda da classe `JApplet`. O método `getContentPane` retorna uma referência para o painel de conteúdo do *applet*. O programa usa esta referência para anexar componentes GUI, como uma `JTextArea`, à interface com o usuário do *applet*.

A linha 22 coloca o objeto componente GUI `JTextArea` ao qual `outputArea` faz referência no *applet*. Quando o *applet* é executado, quaisquer componentes GUI anexados a ele são exibidos. O método `add` de `Container` anexa um componente GUI a um contêiner. Por enquanto, podemos anexar somente um componente GUI ao painel de conteúdo do *applet* e esse componente GUI automaticamente ocupará a área inteira de desenho do *applet* na tela (como definido pelas configurações `width` e `height` do *applet*, em pixels, no documento de HTML do *applet*). Mais tarde, discutiremos como inserir vários componentes GUI em um *applet*, mudando o layout do *applet*. O layout controla como o *applet* posiciona componentes GUI em sua área na tela.

A linha 24 declara a variável `int result` para armazenar o resultado de cada cálculo de quadrado. A linha 25 declara a referência para `String output` e a inicializa com o *string* vazio. Este `String` conterá os resultados de se elevar ao quadrado os valores de 1 a 10. As linhas 28 a 37 definem uma estrutura de repetição `for`. Cada iteração do laço calcula o quadrado do valor atual da variável de controle `counter`, armazena o valor em `result` e concatena `result` no final de `output`.

O *applet* invoca (ou chama) seu método `square` na linha 31 com a instrução

```
result = square( counter );
```

O `()` após `square` representa o *operador de chamada de método*, o qual tem precedência alta. Neste ponto, o programa faz uma cópia do valor de `counter` (o *argumento* para a chamada de método) e transfere o controle do programa para a primeira linha do método `square` (definido nas linhas 44 a 48). O método `square` recebe a cópia do valor de `counter` no *parâmetro y*. Então, `square` calcula `y * y`. O método `square` usa uma instrução `return` para devolver (i.e., mandar de volta) o resultado do cálculo para a instrução em `init` que invocou `square`. No método `init`, o valor devolvido é atribuído à variável `result`. As linhas 34 e 35 concatenam "The square of ", o valor de `counter`, " is ", o valor de `result` e um caractere de nova linha ao final de `output`. Esse processo se repete para cada iteração da estrutura de repetição `for`. A linha 39 utiliza o método `setText` para configurar o texto da `outputArea` com o `String output`.

```

1 // Fig. 6.3: Squareintegers.java
2 // Um método de cálculo de quadrados definido pelo programador
3
4 // Pacotes do núcleo de Java
5 import java.awt.Container;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class SquareIntegers extends JApplet {
11
12     // prepara GUI e calcula quadrados de inteiros de 1 a 10
13     public void init()
14     {
15         // JTextArea para exibir resultados
16         JTextArea outputArea = new JTextArea();
17

```

Fig. 6.3 Utilizando o método `square` definido pelo programador (parte 1 de 2).

```

18     // obtém painel de conteúdo do applet (área de exibição do componente GUI)
19     Container container = getContentPane();
20
21     // anexa outputArea ao Container
22     container.add( outputArea );
23
24     int result;    // armazena o resultado da chamada para o método square
25     String output = "";    // String que contém os resultados
26
27     // repete 10 vezes
28     for ( int counter = 1; counter <= 10; counter++ ) {
29
30         // calcula o quadrado de counter e armazena em result
31         result = square( x );
32
33         // acrescenta result ao String output
34         output += "The square of " + counter +
35             " is " + result + "\n";
36
37     } // fim da estrutura for
38
39     outputArea.setText( output );    // coloca resultados em JTextArea
40
41 } // fim do método init
42
43 // definição do método square
44 public int square( int y )
45 {
46     return y * y;
47
48 } // fim do método square
49
50 } // fim da classe SquareIntegers

```

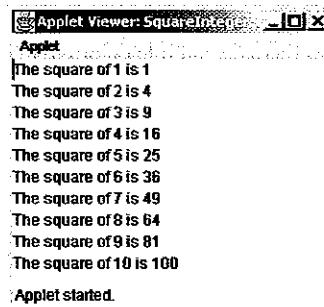


Fig. 6.3 Utilizando o método `square` definido pelo programador (parte 2 de 2).

Observe que declararmos as referências `output`, `outputArea` e `container` e a variável `result` como variáveis locais em `init`, porque elas são utilizadas somente em `init`. As variáveis devem ser declaradas como variáveis de instância só se precisam ser utilizadas em mais de um método da classe ou se seus valores devem ser salvos pelo programa entre as chamadas aos métodos da classe. Além disso, observe que o método `init` chama o método `square` diretamente, sem anteceder o nome do método com um nome de classe e um operador ponto ou um nome de referência e um operador ponto. Cada método em uma classe é capaz de chamar os outros métodos da classe diretamente. Entretanto, existe uma exceção a esta regra. Os métodos `static` de uma classe somente podem chamar outros métodos `static` da classe diretamente. O Capítulo 8 discute os métodos `static` em detalhes.

A definição do método `square` (linha 44) mostra que `square` espera um parâmetro inteiro `y`; `square` usa este nome para manipular o valor que ele recebe. A palavra-chave `int` que precede o nome do método indica que `square` devolve um resultado inteiro. A instrução `return` em `square` passa o resultado do cálculo `y * y` de vol-

ta para o método que chamou. Observe que a definição inteira do método está contida entre as chaves da classe **SquareIntegers**. Todos os métodos devem estar definidos dentro de uma definição de classe.



Boa prática de programação 6.2

Coloque uma linha em branco entre as definições de métodos para separar os métodos e melhorar a legibilidade do programa.



Erro comum de programação 6.2

Definir um método fora das chaves de uma definição de classe é um erro de sintaxe.

O formato genérico de uma definição de método é

```
tipo do valor de retorno nome do método ( lista de parâmetros )
{
    declarações e instruções
}
```

O *nome do método* é qualquer identificador válido. O *tipo do valor de retorno* é o tipo de dados do resultado retornado do método para o chamador. O *tipo do valor de retorno void* indica que um método não retorna um valor. Os métodos podem retornar no máximo um valor.

A *lista de parâmetros* é uma lista separada por vírgulas na qual o método declara o nome e o tipo de cada parâmetro. Deve haver um argumento na chamada de método para cada parâmetro na definição do método. Cada argumento também deve ser compatível com o tipo do parâmetro correspondente na definição do método. Por exemplo, um parâmetro do tipo **double** pode receber valores 7.35, 22 ou -0.03546, mas não "hello" (porque o **String** não pode ser atribuído a uma variável **double**). Se um método não recebe nenhum valor, a *lista de parâmetros* está vazia (isto é, após o nome do método há um conjunto vazio de parênteses). Cada parâmetro na lista de parâmetros de um método deve ser declarado com um tipo de dados; caso contrário, ocorre um erro de sintaxe.

Segundo a primeira linha da definição do método (também conhecida como *cabeçalho do método*), *declarações e instruções* entre chaves formam o *corpo do método*. O corpo do método também é conhecido como *bloco*. As variáveis podem ser declaradas em qualquer bloco e os blocos podem ser aninhados. Um método não pode ser definido dentro de outro método.

Há três maneiras de devolver o controle para a instrução que invocou um método. Se o método não devolve um resultado, o controle é devolvido quando o fluxo de controle do programa atinge a chave direita de fechamento do método ou quando a instrução

```
return;
```

é executada. Se o método devolve um resultado, a instrução

```
return expressão;
```

avalia a *expressão* e depois devolve o valor resultante ao chamador. Quando uma instrução **return** é executada, o controle é devolvido imediatamente para a instrução que invocou o método.

Observe que o exemplo da Fig. 6.3, na verdade, contém duas definições de método – **init** (linhas 13 a 41) e **square** (linhas 44 a 48). Lembre-se de que o contêiner de *applets* chama o método **init** para inicializar o *applet*. Nesse exemplo, o método **init** invoca repetidamente o método **square** para executar um cálculo e depois exibe os resultados na **JTextArea** que está anexada ao painel de conteúdo do *applet*. Quando o *applet* aparece na tela, os resultados são exibidos na **JTextArea**.

Repare na sintaxe utilizada para invocar o método **square** – utilizamos somente o nome do método, seguido pelos argumentos para o método entre parênteses. Os métodos em uma definição de classe têm permissão para invocar outros métodos na mesma definição de classe utilizando essa sintaxe (há uma exceção a essa regra discutida no Capítulo 8). Os métodos na mesma definição de classe são tanto métodos definidos nessa classe como métodos herdados (os métodos de classe que a classe atual estende [**extends**] – **JApplet** na Fig. 6.3). Agora já vimos três maneiras de chamar um método: um nome de método sozinho (como mostrado com **square(x)** nesse exemplo), uma referência a um objeto seguido pelo operador ponto (**.**) e o nome de método (como **g.drawLine(x1, y1, x2, y2)**) e um nome de classe seguido por um nome de método (como **Integer.parseInt(stringToConvert)**). A última sintaxe é usada somente para métodos **static** de uma classe (discutidos em detalhes no Capítulo 8).



Erro comum de programação 6.3

Omitir o tipo do valor de retorno em uma definição de método é um erro de sintaxe.



Erro comum de programação 6.4

Esquecer de devolver um valor de um método que supostamente deve devolver um valor é um erro de sintaxe. Se um tipo do valor de retorno diferente de `void` for especificado, o método deve conter uma instrução `return`.



Erro comum de programação 6.5

Retornar o valor de um método cujo tipo de retorno foi declarado como `void` é um erro de sintaxe.



Erro comum de programação 6.6

Declarar parâmetros de método do mesmo tipo como `float x, y` em vez de `float x, float y` é um erro de sintaxe porque são exigidos tipos diferentes para cada parâmetro na lista de parâmetros.



Erro comum de programação 6.7

Colocar um ponto-e-vírgula após o parêntese direito que envolve a lista de parâmetros de uma definição de método é um erro de sintaxe.



Erro comum de programação 6.8

Redefinir um parâmetro de método no corpo do método é um erro de sintaxe.



Erro comum de programação 6.9

Passar para um método um argumento que não é compatível com o tipo do parâmetro correspondente é um erro de sintaxe.



Erro comum de programação 6.10

Definir um método dentro de outro método é um erro de sintaxe.



Boa prática de programação 6.3

Evite usar os mesmos nomes para variáveis de instância e variáveis locais. Isso ajuda os leitores de seu programa a distinguir variáveis usadas em diferentes partes de uma definição de classe.



Boa prática de programação 6.4

Escolher nomes de métodos significativos e nomes de parâmetros significativos torna os programas mais legíveis e ajuda a evitar a utilização excessiva de comentários.



Observação de engenharia de software 6.5

Um método normalmente não deve ter mais do que uma página. Melhor ainda, normalmente não deve ocupar mais que meia página. Independentemente do comprimento de um método, ele deve realizar bem uma tarefa. Métodos pequenos promovem a capacidade de reutilização do software.



Observação de engenharia de software 6.6

Os programas devem ser escritos como coleções de métodos pequenos. Isso torna mais fácil escrever, depurar, manter e modificar os programas.



Observação de engenharia de software 6.7

Um método que requer um número grande de parâmetros pode estar realizando tarefas demais. Pense em dividir o método em métodos menores que realizam as tarefas separadas. O cabeçalho do método deve caber em uma linha, se possível.



Observação de engenharia de software 6.8

O cabeçalho do método e as suas chamadas devem concordar em número, tipo e ordem de parâmetros e argumentos.

*Dica de teste e depuração 6.1*

Métodos pequenos são mais fáceis de testar, depurar e entender que os grandes.

O *applet* em nosso próximo exemplo (Fig. 6.4) utiliza um método definido pelo programador, denominado **maximum**, para determinar e devolver o maior de três valores de ponto flutuante.

```

1 // Fig. 6.4: Maximum.java
2 // Localizando o maior de três valores do tipo double
3
4 // Pacotes do núcleo de Java
5 import java.awt.Container;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class Maximum extends JApplet {
11
12     // inicializa o applet lendo dados digitados pelo usuário e criando GUI
13     public void init()
14     {
15         // obtém dados digitados pelo usuário
16         String s1 = JOptionPane.showInputDialog(
17             "Enter first floating-point value" );
18         String s2 = JOptionPane.showInputDialog(
19             "Enter second floating-point value" );
20         String s3 = JOptionPane.showInputDialog(
21             "Enter third floating-point value" );
22
23         // converte dados digitados pelo usuário para valores double
24         double number1 = Double.parseDouble( s1 );
25         double number2 = Double.parseDouble( s2 );
26         double number3 = Double.parseDouble( s3 );
27
28         // chama o método maximum para determinar o maior valor
29         double max = maximum( number1, number2, number3 );
30
31         // cria a JTextArea para exibir os resultados
32         JTextArea outputArea = new JTextArea();
33
34         // exibe os números e o valor máximo
35         outputArea.setText( "number1: " + number1 +
36             "\nnumber2: " + number2 + "\nnumber3: " + number3 +
37             "\nmaximum is: " + max );
38
39         // obtém a área de exibição do componente GUI do applet
40         Container container = getContentPane();
41
42         // anexa a outputArea ao Container container
43         container.add( outputArea );
44
45     } // fim do método init
46
47     // o método maximum usa o método max da classe Math
48     // para ajudar a determinar o valor máximo
49     public double maximum( double x, double y, double z )
50     {

```

Fig. 6.4 O método **maximum** definido pelo programador (parte 1 de 2).

```

51     return Math.max( x, Math.max( y, z ) );
52
53 } // fim do método maximum
54
55 } // fim da classe Maximum

```

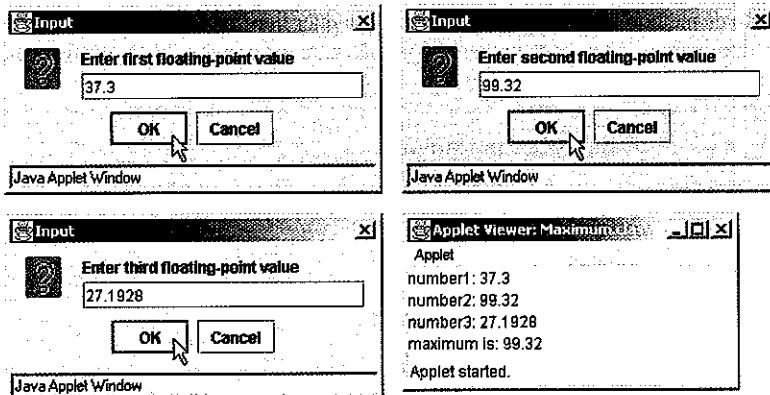


Fig. 6.4 O método `maximum` definido pelo programador (parte 2 de 2).

Os três valores em ponto flutuante são digitados pelo usuário através dos diálogos de entrada (linhas 16 a 21 de `init`). As linhas 24 a 26 utilizam o método `Double.parseDouble` para converter os `Strings` digitados pelo usuário em valores `double`. A linha 29 chama o método `maximum` (definido nas linhas 49 a 53) para determinar o maior valor `double` dentre os três valores `double` passados como argumentos para o método. O método `maximum` devolve o resultado para o método `init` usando uma instrução `return`. O programa atribui o resultado à variável `max`. As linhas 35 a 37 usam a concatenação de `String` para formar um `String` que contém os três valores `double` digitados pelo usuário e o valor `max` e colocam o resultado na `JTextArea outputArea`.

Repare na implementação do método `maximum` (linhas 49 a 53). A primeira linha indica que o método devolve um valor em ponto flutuante `double`, que o nome de método é `maximum` e que o método aceita três parâmetros `double` (`x`, `y` e `z`) para realizar sua tarefa. Além disso, a instrução (linha 51) no corpo do método devolve o maior dos três valores de ponto flutuante, utilizando duas chamadas ao método `Math.max`. Primeiro, a instrução invoca o método `Math.max` com os valores das variáveis `y` e `z` para determinar o maior desses dois valores. Em seguida, a instrução passa o valor da variável `x` e o resultado da primeira chamada a `Math.max` para o método `Math.max`. Por fim, o resultado da segunda chamada a `Math.max` é devolvido para a linha 29 (o ponto em que o método `init` invocou o método `maximum`).

6.6 Promoção de argumentos

Outro recurso importante das definições de métodos é a *coerção de argumentos* (isto é, forçar os argumentos para o tipo apropriado a fim de passá-los para o método). Por exemplo, o programa pode chamar o método `sqrt` da classe `Math` com um argumento inteiro, embora o método espere receber um argumento `double`. Por exemplo, a instrução

```
System.out.println( Math.sqrt( 4 ) );
```

avalia corretamente `Math.sqrt(4)` e imprime o valor 2. A lista de parâmetros da definição do método faz com que Java converta o valor inteiro 4 para o valor `double` 4.0 antes de passar o valor para `sqrt`. Em alguns casos, tentar essas conversões leva a erros de compilação se as *regras de promoção* de Java não forem satisfeitas. As regras de promoção especificam como os tipos podem ser convertidos em outros tipos sem perder dados. Em nosso exemplo `sqrt` acima, `int` é convertido em `double` sem alterar seu valor. Entretanto, converter um `double` em `int`

trunca a parte fracionária do valor **double**. Converter tipos inteiros grandes para tipos inteiros pequenos (por exemplo, **long** para **int**) também pode resultar em valores alterados.

As regras de promoção se aplicam a expressões que contém valores de dois ou mais tipos de dados (também conhecidas como *expressões de tipo misto*) e aos valores de tipos de dados primitivos passados como argumentos para métodos. O tipo de cada valor em uma expressão de tipo misto é promovido para “o mais alto” tipo na expressão (na realidade, a expressão usa uma cópia temporária de cada valor; os valores originais permanecem inalterados). O tipo de um argumento de método pode ser promovido para qualquer tipo “mais alto”. A Fig. 6.5 lista os tipos de dados primitivos e os tipos para os quais cada um tem permissão para ser promovido automaticamente.

Tipo	Promoções permitidas
double	Nenhuma
float	double
long	float ou double
int	long, float ou double
char	int, long, float ou double
short	int, long, float ou double
byte	short, int, long, float ou double
boolean	Nenhuma (valores boolean não são considerados números em Java)

Fig. 6.5 Promoções permitidas para tipos de dados primitivos.

Converter valores para tipos inferiores pode resultar em valores diferentes. Portanto, nos casos em que as informações podem ser perdidas devido à conversão, o compilador Java exige que o programador utilize um operador de coerção para forçar que a conversão ocorra. Para invocar nosso método **square**, que utiliza um parâmetro inteiro com a variável **double** **y** (Fig. 6.3), escrevemos a chamada de método como **square((int) y)**. Essa chamada de método faz explicitamente a coerção (conversão) do valor de **y** em um inteiro para utilização no método **square**. Portanto, se o valor de **y** é **4.5**, o método **square** retorna **16**, não **20.25**.



Erro comum de programação 6.11

Converter um valor de um tipo de dados primitivo para outro tipo de dados primitivo pode alterar o valor se o novo tipo de dados não é uma promoção permitida (por exemplo, **double** para **int**). Além disso, converter qualquer valor integral para um valor de ponto flutuante e de volta para um valor integral pode introduzir erros de arredondamento no resultado.

6.7 Pacotes da Java API

Como vimos, Java contém muitas classes predefinidas que são agrupadas por diretórios em categorias de classes relacionadas, denominadas *pacotes*. Em geral, referimo-nos a esses pacotes como a Java API (*Java applications programming interface* – interface de programas aplicativos em Java), ou a *biblioteca de classes Java*.

Por todo o texto, as instruções **import** são utilizadas para especificar a localização das classes necessárias para compilar um programa Java. Por exemplo, o programa usa a instrução

```
import javax.swing.JApplet;
```

para dizer ao compilador que carregue a classe **JApplet** do pacote **javax.swing**. Um dos grandes pontos fortes de Java é o grande número de classes nos pacotes da Java API que os programadores podem reutilizar em vez de “reinventar a roda”. Exercitamos um grande número dessas classes neste livro. A Fig. 6.6 lista um subconjunto dos muitos pacotes da Java API e fornece uma breve descrição de cada pacote. Usamos as classes desses pacotes e outros ao longo do livro. Fornecemos essa tabela para começar a lhe apresentar a variedade de componentes reutilizáveis disponíveis na Java API. Ao aprender Java, você deve dedicar algum tempo para ler as descrições dos pacotes e as classes na documentação da Java API (java.sun.com/j2se/1.3/docs/api).

Pacote	Descrição
<code>java.applet</code>	<i>The Java Applet Package.</i> Esse pacote contém a classe <code>Applet</code> e várias interfaces que permitem a criação de <i>applets</i> , interação de <i>applets</i> com o navegador e execução de clipes de áudio. Em Java 2, utiliza-se a classe <code>javax.swing.JApplet</code> para definir um <i>applet</i> que utiliza <i>componentes GUI Swing</i> .
<code>java.awt</code>	<i>The Java Abstract Windowing Toolkit Package.</i> Esse pacote contém as classes e interfaces exigidas para criar e manipular interfaces gráficas com o usuário em Java 1.0 e 1.1. Em Java 2, essas classes ainda podem ser utilizadas, mas os <i>componentes GUI Swing</i> dos pacotes <code>javax.swing</code> são freqüentemente utilizados em seu lugar.
<code>java.awt.event</code>	<i>The Java Abstract Windowing Toolkit Event Package</i> Contém as classes e interfaces que permitem o tratamento de eventos para os componentes GUI nos pacotes <code>java.awt</code> e <code>javax.swing</code> .
<code>java.io</code>	<i>The Java Input/Output Package.</i> Contém as classes que permitem aos programas fazer entrada e saída de dados (veja o Capítulo 16).
<code>java.lang</code>	<i>The Java Language Package.</i> Contém as classes e interfaces exigidas por muitos programas Java (muitas são discutidos ao longo desse texto) e é automaticamente importado pelo compilador para todos os programas.
<code>java.net</code>	<i>The Java Networking Package.</i> Contém as classes que permitem aos programas comunicar-se através das redes (veja o Capítulo 17).
<code>java.text</code>	<i>The Java Text Package.</i> Contém as classes e interfaces que permitem a um programa Java manipular números, datas, caracteres e <i>strings</i> . Fornece muitos dos recursos de internacionalização de Java, i. e., recursos que permitem a um programa ser personalizado para um local específico (por exemplo, o <i>applet</i> pode exibir <i>strings</i> em idiomas diferentes, com base no país do usuário).
<code>java.util</code>	<i>The Java Utilities Package.</i> Contém classes de utilitários e interfaces como: manipulações de data e hora, recursos para processamento de números aleatórios (<code>Random</code>), armazenamento e processamento de grandes quantidades de dados, quebra de <i>strings</i> em pedaços menores chamados <i>tokens</i> (<code> StringTokenizer</code>) e outros recursos (veja o Capítulo 19, o Capítulo 20 e o Capítulo 21).
<code>javax.swing</code>	<i>The Java Swing GUI Components Package.</i> Contém classes e interfaces para os componentes GUI Swing de Java que fornecem suporte para GUIs portáteis.
<code>javax.swing.event</code>	<i>The Java Swing Event Package.</i> Contém classes e interfaces que permitem tratamento de eventos para componentes GUI no pacote <code>javax.swing</code> .

Fig. 6.6 Os pacotes da Java API.

O conjunto de pacotes disponível no Java 2 Software Development Kit (J2SDK) é bastante grande. Além dos pacotes resumidos na Fig. 6.6, o J2SDK contém pacotes para gráficos complexos, interfaces gráficas avançadas com o usuário, impressão, redes avançadas, segurança, processamento de bancos de dados, multimídia, acessibilidade (para pessoas com deficiências) e muitas outras funções. Para uma visão geral dos pacotes no J2SDK versão 1.3, visite a página

java.sun.com/j2se/1.3/docs/api/overview-summary.html

Além disso, muitos outros pacotes estão disponíveis para baixar no endereço java.sun.com.

6.8 Geração de números aleatórios

Agora faremos um breve e, esperamos, divertido desvio para abordar uma conhecida aplicação de programação, a saber, a simulação de jogos. Nesta seção e na próxima, desenvolveremos um programa para jogar bem-estruturado que inclui múltiplos métodos. O programa utiliza a maioria das estruturas de controle que estudamos até este ponto do livro e apresenta diversos conceitos novos.

Há algo na atmosfera de um cassino que revigora as pessoas – dos grandes apostadores nas mesas de feltro e mogno dos jogos de dados aos pequenos apostadores nos caça-níqueis. Trata-se do *elemento sorte*, a possibilidade de que a sorte converta um punhado de dinheiro em uma montanha de riquezas. O elemento sorte pode ser introduzido através do método `random` da classe `Math`. [Nota: Java também oferece uma classe `Random` no pacote `java.util`. A classe `Random` é abordada no Capítulo 20.]

Considere a seguinte instrução:

```
double randomValue = Math.random();
```

O método `random` da classe `Math` gera um valor `double` de 0.0 até, mas não incluindo, 1.0. Se `random` realmente produz valores de forma aleatória, então cada valor de 0.0 até, mas não incluindo, 1.0 tem uma *chance* (ou *probabilidade*) igual de ser escolhido toda vez que `random` é chamado. Observe que os valores devolvidos por `random` são na verdade *números pseudo-aleatórios* – uma sequência de valores produzida por um cálculo matemático complexo. Este cálculo usa a hora do dia atual para *semejar* o gerador de números aleatórios, de modo que cada execução de um programa dá origem a uma sequência diferente de valores aleatórios.

O intervalo de valores produzido diretamente pelo método `random` muitas vezes é diferente do intervalo de valores necessário em um aplicativo Java em particular. Por exemplo, o programa que simula o lançamento de uma moeda talvez exija somente 0 para “cara” e 1 para “coroa”. O programa que simula o lançamento de um dado de seis faces exige inteiros aleatórios no intervalo de 1 a 6. O programa que prevê aleatoriamente o próximo tipo de nave espacial (uma entre quatro possibilidades) que voará no horizonte de um *video game* exigiria inteiros aleatórios no intervalo de 1 a 4.

Para demonstrar `random`, vamos desenvolver um programa que simula 20 lançamentos de um dado de seis faces e exibe o valor de cada lançamento. Utilizamos o operador de multiplicação (*) em conjunto com o método `random`, como segue, para produzir inteiros no intervalo de 0 a 5:

```
(int) ( Math.random() * 6 )
```

Essa manipulação chama-se *escalonar* o intervalo de valores produzido pelo método `random` de `Math`. O número 6 na expressão precedente chama-se *fator de escala*. O operador de coerção de inteiros trunca a parte em ponto flutuante (a parte depois da casa decimal) de cada valor produzido pela expressão. Assim, *deslocamos* o intervalo de números produzido adicionando 1 ao nosso resultado anterior, como em

```
1 + (int) ( Math.random() * 6 )
```

A Fig. 6.7 confirma que os resultados do cálculo precedente são inteiros e estão no intervalo de 1 a 6.

```

1 // Fig. 6.7: RandomIntegers.java
2 // Inteiros aleatórios deslocados e escalonados
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 public class RandomIntegers {
8
9     // método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        int value;
13        String output = "";
14

```

Fig. 6.7 Inteiros aleatórios deslocados e escalonados (parte 1 de 2).

```

15     // repete 20 vezes
16     for ( int counter = 1; counter <= 20; counter++ ) {
17
18         // escolhe números aleatórios entre 1 e 6
19         value = 1 + ( int ) ( Math.random() * 6 );
20
21         output += value + " ";    // acrescenta o valor à saída
22
23         // se counter é divisível por 5,
24         // acrescenta nova linha ao String output
25         if ( counter % 5 == 0 )
26             output += "\n";
27
28     } // fim da estrutura for
29
30     JOptionPane.showMessageDialog( null, output,
31         "20 Random Numbers from 1 to 6",
32         JOptionPane.INFORMATION_MESSAGE );
33
34     System.exit( 0 );    // termina o aplicativo
35
36 } // fim do método main
37
38 } // fim da classe RandomIntegers

```

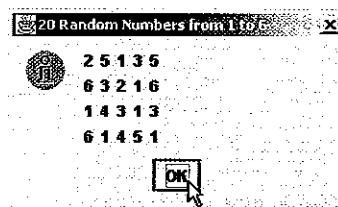


Fig. 6.7 Inteiros aleatórios deslocados e escalonados (parte 2 de 2).

Para mostrar que esses números ocorrem com probabilidade aproximadamente igual, vamos simular 6000 lançamentos de um dado com o programa da Fig. 6.8. Cada inteiro de 1 a 6 deve aparecer aproximadamente 1000 vezes.

```

1 // Fig. 6.8: Rolldie.java
2 // Rola um dado de seis faces 6000 vezes
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class RollDie {
8
9     // método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        int frequency1 = 0, frequency2 = 0, frequency3 = 0,
13            frequency4 = 0, frequency5 = 0, frequency6 = 0, face;
14
15        // resume os resultados
16        for ( int roll = 1; roll <= 6000; roll++ ) {
17            face = 1 + ( int ) ( Math.random() * 6 );

```

Fig. 6.8 Lançando um dado de seis faces 6000 vezes (parte 1 de 2).

```

18
19      // determina valor do lançamento e incrementa contador correspondente
20      switch ( face ) {
21
22          case 1:
23              ++frequency1;
24              break;
25
26          case 2:
27              ++frequency2;
28              break;
29
30          case 3:
31              ++frequency3;
32              break;
33
34          case 4:
35              ++frequency4;
36              break;
37
38          case 5:
39              ++frequency5;
40              break;
41
42          case 6:
43              ++frequency6;
44              break;
45
46      } // fim da estrutura switch .
47
48  } // fim da estrutura for
49
50 JTextArea outputArea = new JTextArea();
51
52 outputArea.setText( "Face\tFrequency" +
53     "\n1\t" + frequency1 + "\n2\t" + frequency2 +
54     "\n3\t" + frequency3 + "\n4\t" + frequency4 +
55     "\n5\t" + frequency5 + "\n6\t" + frequency6 );
56
57 JOptionPane.showMessageDialog( null, outputArea,
58     "Rolling a Dice 6000 Times",
59     JOptionPane.INFORMATION_MESSAGE );
60
61 System.exit( 0 );    // termina o aplicativo
62
63 } // fim do método main
64
65 } // fim da classe RollDie

```

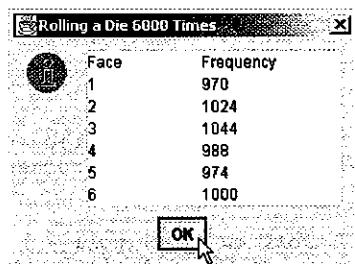


Fig. 6.8 Lançando um dado de seis faces 6000 vezes (parte 2 de 2).

Como mostra a saída do programa, escalar e deslocar os valores produzidos pelo método `random` permite simular realisticamente o lançamento de um dado de seis faces. Observe que utilizamos estruturas de controle aninhadas no programa para determinar o número de vezes que cada face do dado ocorreu. O laço `for` nas linhas 16 a 48 é repetido 6000 vezes. A cada iteração do laço, a linha 17 produz um valor de 1 a 6. A estrutura aninhada `switch` nas linhas 20 a 46 utiliza como sua expressão de controle o valor `face` que foi escolhido aleatoriamente. Com base no valor de `face`, a estrutura `switch` incrementa uma das seis variáveis contadoras durante cada iteração do laço. Observe que a estrutura `switch` não tem nenhum caso `default`. Quando estudarmos *arrays* no Capítulo 7, mostraremos como substituir toda a estrutura `switch` nesse programa por uma única instrução. Execute o programa várias vezes e observe os resultados. Observe que o programa produz resultados diferentes cada vez que o programa é executado.

Os valores produzidos diretamente por `random` estão sempre no intervalo

```
0.0 ≤ Math.random() < 1.0
```

Anteriormente, demonstramos como escrever uma única instrução para simular a rolagem de um dado de seis lados com a instrução

```
face = 1 + (int) ( Math.random() * 6 );
```

a qual sempre atribui (aleatoriamente) um inteiro no intervalo $1 \leq \text{face} \leq 6$ à variável `face`. Observe que a amplitude desse intervalo (isto é, o número de inteiros consecutivos no intervalo) é 6 e o número inicial no intervalo é 1. Examinando a instrução precedente, vemos que a amplitude do intervalo é determinada pelo número utilizado para escalar `random` com o operador de multiplicação (isto é, 6) e o número inicial do intervalo é igual ao número (isto é) adicionado a `(int) (Math.random() * 6)`. Podemos generalizar esse resultado como:

```
n = a + (int) ( Math.random() * b );
```

onde `a` é o *valor de deslocamento* (que é igual ao primeiro número no intervalo desejado de inteiros consecutivos) e `b` é o *fator de escala* (que é igual à amplitude do intervalo desejado de inteiros consecutivos). Nos exercícios, veremos que é possível escolher inteiros aleatoriamente a partir de conjuntos de valores que não sejam intervalos de inteiros consecutivos.

6.9 Exemplo: um jogo de azar

Um dos jogos de azar mais populares nos EUA é um jogo de dados conhecido como “*craps*”, que é jogado em cassinos e nas ruas de todo o mundo. As regras do jogo são simples e diretas:

*O jogador lança dois dados. Cada dado tem seis faces. Essas faces contêm 1, 2, 3, 4, 5 e 6 pontos, respectivamente. Depois que os dados param de rolar, calcula-se a soma dos pontos nas faces viradas para cima. Se a soma for 7 ou 11 no primeiro lance, o jogador ganha. Se a soma for 2, 3 ou 12 no primeiro lance (chamado de “*craps*”), o jogador perde (isto é, a “casa” ganha). Se a soma for 4, 5, 6, 8, 9 ou 10 no primeiro lance, essa soma torna-se a “pontuação” do jogador. Para ganhar, você deve continuar a rolar os dados até “fazer sua pontuação” (isto é, obter um valor igual à sua pontuação). O jogador perde se obtiver um 7 antes de fazer a pontuação.*

O applet na Fig. 6.9 simula o jogo de *craps*.

```

1 // Fig. 6.9: Craps.java
2 // Craps
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class Craps extends JApplet implements ActionListener {
```

Fig. 6.9 Programa que simula o jogo *craps* (parte 1 de 4).

```

12
13 // variáveis constantes para status do jogo
14 final int WON = 0, LOST = 1, CONTINUE = 2;
15
16 // outras variáveis utilizadas
17 boolean firstRoll = true; // true se for a primeira jogada
18 int sumOfDice = 0; // soma dos dados
19 int myPoint = 0; // pontos se não ganhar/perder na 1a. jogada
20 int gameStatus = CONTINUE; // o jogo ainda não acabou
21
22 // componentes da interface gráfica com o usuário
23 JLabel die1Label, die2Label, sumLabel, pointLabel;
24 JTextField die1Field, die2Field, sumField, pointField;
25 JButton rollButton;
26
27 // configura os componentes GUI
28 public void init()
29 {
30     // obtém painel de conteúdo e muda o Leiaute
31     // para um FlowLayout
32     Container container = getContentPane();
33     container.setLayout( new FlowLayout() );
34
35     // cria campo de rótulo e texto para o dado 1
36     die1Label = new JLabel( "Die 1" );
37     container.add( die1Label );
38     die1Field = new JTextField( 10 );
39     die1Field.setEditable( false );
40     container.add( die1Field );
41
42     // cria campo de rótulo e texto para o dado 2
43     die2Label = new JLabel( "Die 2" );
44     container.add( die2Label );
45     die2Field = new JTextField( 10 );
46     die2Field.setEditable( false );
47     container.add( die2Field );
48
49     // cria campo de rótulo e texto para a soma
50     sumLabel = new JLabel( "Sum is" );
51     container.add( sumLabel );
52     sumField = new JTextField( 10 );
53     sumField.setEditable( false );
54     container.add( sumField );
55
56     // cria campo de rótulo e texto para point
57     pointLabel = new JLabel( "Point is" );
58     container.add( pointLabel );
59     pointField = new JTextField( 10 );
60     pointField.setEditable( false );
61     container.add( pointField );
62
63     // cria botão em que o usuário clica para lançar os dados
64     rollButton = new JButton( "Roll Dice" );
65     rollButton.addActionListener( this );
66     container.add( rollButton );
67 }
68
69 // processa um lançamento dos dados
70 public void actionPerformed( ActionEvent actionEvent )
71 {

```

Fig. 6.9 Programa que simula o jogo *craps* (parte 2 de 4).

```

72     // primeiro lançamento dos dados
73     if ( firstRoll ) {
74         sumOfDice = rollDice();      // lança os dados
75
76     switch ( sumOfDice ) {
77
78         // ganha na primeira jogada
79         case 7: case 11:
80             gameStatus = WON;
81             pointField.setText( "" ); // limpa campo de texto da pontuação
82             break;
83
84         // perde na primeira jogada
85         case 2: case 3: case 12:
86             gameStatus = LOST;
87             pointField.setText( "" ); // limpa campo de texto da pontuação
88             break;
89
90         // memoriza os pontos obtidos
91     default:
92         gameStatus = CONTINUE;
93         myPoint = sumOfDice;
94         pointField.setText( Integer.toString( myPoint ) );
95         firstRoll = false;
96         break;
97
98     } // fim da estrutura switch
99
100    } // fim do corpo da estrutura if
101
102    // lançamentos subseqüentes dos dados
103    else {
104        sumOfDice = rollDice();      // lança os dados
105
106        // determina estado do jogo
107        if ( sumOfDice == myPoint ) // vitória fazendo o ponto
108            gameStatus = WON;
109        else
110            if ( sumOfDice == 7 )     // derrota por jogar 7
111                gameStatus = LOST;
112    }
113
114    // exibe mensagem indicando estado do jogo
115    displayMessage();
116
117} // fim do método actionPerformed
118
119// lança os dados, calcula soma e exibe resultados
120public int rollDice()
121{
122    int die1, die2, sum;
123
124    // escolhe valores aleatórios para os dados
125    die1 = 1 + ( int ) ( Math.random() * 6 );
126    die2 = 1 + ( int ) ( Math.random() * 6 );
127
128    sum = die1 + die2;    // soma os valores dos dados
129
130    // exibe resultados
131    die1Field.setText( Integer.toString( die1 ) );
132    die2Field.setText( Integer.toString( die2 ) );

```

Fig. 6.9 Programa que simula o jogo *craps* (parte 3 de 4).

```

133     sumField.setText( Integer.toString( sum ) );
134
135     return sum; // retorna soma dos dados
136
137 } // fim do método rollDice
138
139 // determina o estado do jogo e
140 // exibe mensagem apropriada na barra de estado
141 public void displayMessage()
142 {
143     // jogo deve continuar
144     if ( gameStatus == CONTINUE )
145         showStatus( "Roll again." );
146
147     // jogo ganho ou perdido
148     else {
149
150         if ( gameStatus == WON )
151             showStatus( "Player wins. " +
152                         "Click Roll Dice to play again." );
153         else
154             showStatus( "Player loses. " +
155                         "Click Roll Dice to play again." );
156
157         // próxima jogada é a primeira de um novo jogo
158         firstRoll = true;
159     }
160
161 } // fim do método displayMessage
162
163 } // fim da classe Craps

```

Um objeto JLabel Um objeto JTextField Um objeto JButton

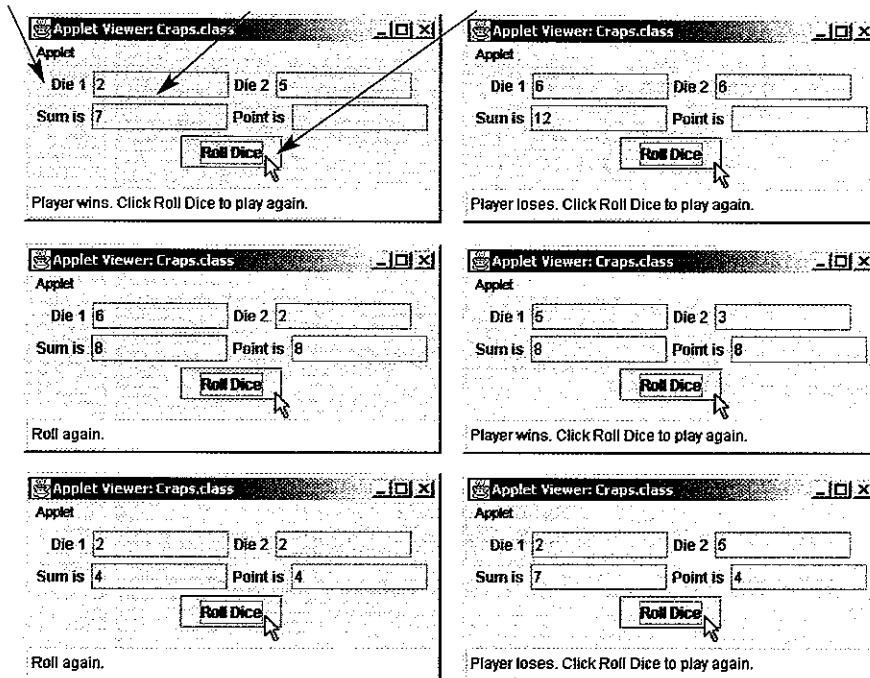


Fig. 6.9 Programa que simula o jogo *craps* (parte 4 de 4).

Repare que o jogador deve lançar dois dados na primeira e em todas as jogadas subsequentes. Ao executar o *applet*, clique no botão **Roll Dice** para jogar. O canto inferior esquerdo da janela do **appletviewer** exibe o resultado de cada jogada. As capturas de tela mostram quatro execuções separadas do *applet* (uma vitória e uma derrota na primeira jogada e uma vitória e uma derrota depois da primeira jogada).

Até agora, a maioria das interações do usuário com nossos programas foram por meio de um diálogo de entrada (em que o usuário podia digitar um valor de entrada para o programa) ou um diálogo de mensagem (em que uma mensagem era exibida para o usuário e este podia clicar em **OK** para fechar o diálogo). Embora esses diálogos sejam maneiras válidas de receber entrada de um usuário e exibir a saída em um programa Java, suas capacidades são relativamente limitadas – o diálogo pode obter somente um valor por vez do usuário e pode exibir somente uma mensagem. É muito mais comum receber múltiplas entradas do usuário de uma vez (como a entrada do nome e de informações de endereço do usuário) ou exibir muitos pedaços de dados de uma vez (como os valores dos dados, a soma dos dados e a pontuação, nesse exemplo). Para iniciar nossa introdução a interfaces de usuário mais elaboradas, esse programa ilustra dois novos conceitos de interface gráfica com o usuário – anexar vários componentes GUI a um *applet* e o tratamento de eventos de GUI. Discutimos cada um destes novos aspectos à medida que são encontrados no programa.

As instruções **import** nas linhas 5 a 9 permitem que o compilador carregue as classes utilizadas nesse *applet*. A linha 5 especifica que o programa usa classes do pacote **java.awt** (especificamente, as classes **Container** e **FlowLayout**). A linha 6 especifica que o programa utiliza classes do pacote **java.awt.event**. Esse pacote contém muitos tipos de dados que permitem processar as interações de um usuário com a GUI de um programa. Nesse programa, usamos os tipos de dados **ActionListener** e **ActionEvent** do pacote **java.awt.event**. A linha 9 especifica que o programa utiliza classes do pacote **javax.swing** (especificamente, as classes **JApplet**, **JLabel**, **JTextField** e **JButton**).

Como declarado anteriormente, todo programa Java baseia-se em pelo menos uma definição de classe que estende e melhora uma definição de classe existente através de herança. Lembre-se de que os *applets* herdam da classe **JApplet**. A linha 11 indica que a classe **Craps** herda de **JApplet** e implementa **ActionListener**. Uma classe pode herdar atributos e comportamentos existentes (dados e métodos) de outra classe especificada à direita da palavra-chave **extends** na definição de classe. Além disso, a classe pode implementar uma ou mais *interfaces*. A interface especifica um ou mais comportamentos (isto é, métodos) que você deve definir em sua definição de classe. Implementar a interface **ActionListener** nos força a *definir um método* com a primeira linha

```
public void actionPerformed( ActionEvent actionEvent )
```

em nossa classe **Craps**. A tarefa desse método é processar uma interação do usuário com o **JButton** (chamado de **Roll Dice** na interface com o usuário). Quando o usuário pressiona o botão, esse método será chamado automaticamente em resposta à interação do usuário. Esse processo chama-se *tratamento de evento*. O *evento* é a interação do usuário (i. e., pressionar o botão). O *tratador de evento* é o método **actionPerformed**. Discutiremos os detalhes dessa interação e o método **actionPerformed** em breve. O Capítulo 9 discute as interfaces em detalhes. Por enquanto, quando você desenvolver seus próprios *applets* que têm interface gráfica com o usuário, imite os recursos que suportam o tratamento de eventos dos componentes GUI que apresentamos.

O jogo é razoavelmente complicado. O jogador pode ganhar ou perder na primeira jogada ou pode ganhar ou pode perder em qualquer jogada. A linha 14 cria variáveis que definem os três estados do jogo de dados *craps* – jogo ganho (**WON**), jogo perdido (**LOST**) ou continua a lançar os dados (**CONTINUE**). A palavra-chave **final** no início da declaração indica que esses são *valores constantes*. Quando o programa declara uma variável final, o programa deve inicializar a variável antes de usar a variável e não pode modificar a variável dali em diante. Se a variável for uma variável de instância, esta inicialização normalmente acontece na declaração da variável. A inicialização também pode ocorrer em um método especial de uma classe chamado de *construtor* (discutido no Capítulo 8). As constantes geralmente se chamam *constantes identificadas* ou *variáveis apenas de leitura (read-only)*. Fornecemos mais detalhes sobre a palavra-chave **final** no Capítulo 7 e no Capítulo 8.

Erro comum de programação 6.12



Depois de declarar e inicializar uma variável **final**, tentar atribuir outro valor a essa variável é um erro de sintaxe.

Boa prática de programação 6.5



Utilize somente letras maiúsculas (com caracteres de sublinhado entre as palavras) nos nomes de variáveis **final**. Esse formato destaca essas constantes em um programa.



Boa prática de programação 6.6

*Utilizar variáveis **final** com nomes significativos em vez de constantes inteiras (como 2) torna os programas mais legíveis.*

As linhas 17 a 20 declaram diversas variáveis de instância que são utilizadas ao longo do applet **Craps**. A variável **firstRoll** é uma variável **boolean** que indica se o próximo lançamento dos dados é o primeiro lançamento no jogo atual. A variável **sumOfDice** guarda a soma dos dados na última jogada. A variável **myPoint** armazena a “pontuação” se o jogador não ganhar nem perder na primeira jogada. A variável **gameStatus** monitora o estado atual do jogo (**WON**, **LOST** ou **CONTINUE**).

As linhas 23 a 25 declaram referências para os componentes GUI utilizados na interface gráfica com o usuário desse applet. As referências **die1Label**, **die2Label**, **sumLabel** e **pointLabel** fazem referência aos *objetos JLabel*. O **JLabel** contém um *string* de caracteres a ser exibido na tela. Normalmente, ele indica o propósito de um outro componente GUI na tela. Nas capturas de tela da Fig. 6.9, os objetos **JLabel** são o texto à esquerda de cada retângulo nas duas primeiras linhas da interface com o usuário. As referências **Die1Field**, **Die2Field**, **sumField** e **pointField** fazem referência aos *objetos JTextField*. Os **JTextFields** são utilizados para obter uma única linha de informações do usuário pelo teclado ou para exibir informações na tela. Os objetos **JTextField** referem-se aos retângulos à direita de cada **JLabel** nas duas primeiras linhas da interface de usuário. A referência **rollButton** se refere a um *objeto JButton*. Quando o usuário pressiona o **JButton**, o programa normalmente responde executando uma tarefa (lançando o dado, neste exemplo). O objeto **JButton** é o retângulo que contém as palavras **Roll Dice** na parte inferior da interface de usuário mostrada na Fig. 6.9. Vimos **JButtons** em programas anteriores – todo diálogo de mensagem e todo diálogo de entrada continham um botão **OK** para fechar o diálogo de mensagem ou enviar a entrada do usuário para o programa. Também vimos **JTextFields** em programas anteriores – todo diálogo de entrada contém um **JTextField** no qual o usuário digita um valor de entrada.

O método **init** (linhas 28 a 67) cria os objetos componentes GUI e os anexa à interface com o usuário. A linha 32 declara a referência **Container container** e atribui a ela o resultado de uma chamada ao método **getContentPane**. Lembre-se, o método **getContentPane** devolve uma referência para o painel de conteúdo do applet que pode ser utilizada para anexar componentes GUI à interface com o usuário do *applet*.

A linha 33 utiliza o método **setLayout** de **Container** para especificar o gerenciador de layout para a interface de usuário do *applet*. Os gerenciadores de layout organizam os componentes GUI em um **Container** para fins de apresentação. Os gerenciadores de layout determinam a posição e o tamanho de cada componente GUI anexado ao contêiner, processando, assim, a maior parte dos detalhes de layout e permitindo que o programador se concentre na aparência e no comportamento básicos dos programas.

FlowLayout é o gerenciador de layout mais simples. Os componentes GUI são posicionados da esquerda para a direita, na ordem em que são anexados ao **Container** (o painel de conteúdo do *applet* neste exemplo) com o método **add**. Quando o gerenciador de layout encontra o limite do contêiner, ele comeceira uma nova linha de componentes e continua dispondo os componentes naquela linha.. A linha 33 cria um novo objeto da classe **FlowLayout** e o passa como argumento para o método **setLayout**. Normalmente, o layout é configurado antes de qualquer componente GUI ser adicionado a um **Container**.



Erro comum de programação 6.13

Se um Container não é suficientemente grande para exibir o componente GUI a ele anexado, alguns ou todos os componentes GUI simplesmente não serão exibidos.

[Nota: cada **Container** pode ter somente um gerenciador de layout de cada vez. **Containers** separados no mesmo programa podem ter gerenciadores de layout diferentes. A maioria dos ambientes de programação Java fornece ferramentas de projeto GUI que ajudam o programador a projetar graficamente uma GUI; assim, as ferramentas escrevem código Java para criar a GUI. Algumas dessas ferramentas de projetar GUIs também permitem que o programador utilize gerenciadores de layout. Os Capítulos 12 e 13 discutem vários gerenciadores de layout que permitem um controle mais preciso sobre o layout dos componentes GUI.]

Cada uma das linhas 36 a 40, 43 a 57, 50 a 54 e 57 a 61 cria um par **JLabel** e **JTextField** e o anexa à interface de usuário. Como esses conjuntos de linhas são muito semelhantes, concentrarmo-nos nas linhas 36 a 40. A linha 36 cria um novo objeto **JLabel**, inicializa-o com o *string* “**Die 1**” e atribui o objeto à referência **die1Label**. Es-

se procedimento rotula o `JTextField` correspondente na interface com o usuário (chamado `die1Field`) de modo que o usuário possa determinar a finalidade do valor exibido em `die1Field`. A linha 37 anexa o `JLabel` a que `die1Label` faz referência ao painel de conteúdo do *applet*. A linha 38 cria um novo objeto `JTextField`, inicializa-o para ter uma largura de 10 caracteres e atribui o objeto à referência `die1Field`. Esse `JTextField` exibe o valor do primeiro dado depois de cada lançamento dos dados. A linha 39 utiliza o método `setEditable` de `JTextField` com o argumento `false` para indicar que o usuário não deve ser capaz de digitar no `JTextField`. Este ajuste torna o `JTextField` *não-editável* e faz com que ele seja exibido com um fundo acinzentado por *default*. Um `JTextField` editável tem um fundo branco (como visto em diálogos de entrada). A linha 32 anexa o `JTextField` a que `die1Field` faz referência ao painel de conteúdo do *applet*.

A linha 64 cria um novo objeto `JButton`, inicializa-o com o string "`Roll Dice`" (esse string aparecerá no botão) e atribui o objeto à referência `rollButton`.

A linha 65 especifica que este *applet this* deve *ouvir* os eventos do `rollButton`. A palavra-chave `this` permite ao *applet* fazer referência a si próprio (discutiremos `this` em detalhes no Capítulo 8). Quando o usuário interage com um componente GUI, um *evento* é enviado para o *applet*. Eventos GUI são mensagens (chamadas de métodos) que indicam que o usuário do programa interagiu com um dos componentes GUI do programa. Por exemplo, quando você pressiona `rollButton` nesse programa, uma mensagem indicando o evento que ocorreu é enviada para o *applet* para notificar o *applet* de que você pressionou o botão. Para um `JButton`, a mensagem indica para o *applet* que *uma ação foi realizada* sobre o `JButton` pelo usuário e automaticamente chama o método `actionPerformed` para processar a interação do usuário.

Esse estilo de programação é conhecido como *programação baseada em eventos* – o usuário interage com um componente GUI, o programa é notificado do evento e o programa processa o evento. A interação do usuário com a GUI “dirige” o programa. Os métodos que são chamados quando um evento ocorre também são conhecidos como *métodos de tratamento de eventos*. Quando um evento GUI ocorre em um programa, Java cria um objeto que contém informações sobre o evento que ocorreu e *chama* um método apropriado de tratamento de evento. Antes que qualquer evento possa ser processado, cada componente GUI deve conhecer qual objeto no programa define o método de tratamento de evento que será chamado quando um evento ocorrer. Na linha 65, o método `addActionListener` de `JButton` é utilizado para dizer a `rollButton` que o *applet (this)* pode *ouvir eventos de ação* e define o método `actionPerformed`. Este procedimento chama-se *registrar o tratador de eventos* no componente GUI (também gostamos de chamá-la de *linha de início de escuta*, porque o *applet* agora está ouvindo eventos do botão). Para responder a um evento de ação, devemos definir uma classe que implemente `ActionListener` (isso exige que a classe também defina o método `actionPerformed`) e devemos registrar o tratador do evento no componente GUI. Por fim, a última linha em `init` anexa o `JButton` a que `roll` faz referência ao painel de conteúdo do *applet*, completando, assim, a interface com o usuário.

O método `actionPerformed` (linhas 70 a 117) é um dos vários métodos que processam interações entre o usuário e componentes GUI. A primeira linha do método indica que `actionPerformed` é um método `public` que não retorna nada (`void`) quando ele completa sua tarefa. O método `actionPerformed` recebe um argumento – um `ActionEvent` – quando ele é chamado em resposta a uma ação realizada sobre um componente GUI pelo usuário (neste caso, pressionar o `JButton`). O argumento `ActionEvent` contém informações sobre a ação que ocorreu.

Definimos o método `rollDice` (linhas 120 a 137) para lançar os dados e calcular e exibir sua soma. O método `rollDice` é definido uma vez, mas é chamado de dois lugares no programa (linhas 74 e 104). O método `rollDice` não recebe nenhum argumento, por isso tem uma lista de parâmetros vazia. O método `rollDice` devolve a soma dos dois dados, depois um tipo `int` é indicado no cabeçalho do método para o valor devolvido.

O usuário clica em `Roll Dice` para lançar os dados. Essa ação invoca o método `actionPerformed` (linha 70) do *applet*. O método `actionPerformed` verifica a variável `boolean firstRoll` (linha 73) para determinar se ela é `true` ou `false`. Se for `true`, essa é a primeira jogada do jogo. A linha 74 chama `rollDice`, que seleciona dois valores aleatórios entre 1 e 6, exibe o valor do primeiro dado, o valor do segundo dado e a soma dos dados nos primeiros três `JtextFields`, respectivamente, e devolve a soma dos dados. Observe que os valores inteiros são convertidos para `Strings` com o método `static Integer.toString` porque `JtextFields` podem exibir somente `Strings`. Depois da primeira jogada, a estrutura `switch` aninhada na linha 76 em `actionPerformed` determina se o jogo foi ganho ou perdido ou se o jogo deve continuar com outra jogada. Após a primeira jogada, se o jogo não tiver acabado, `sumOfDice` é salvo em `myPoint` e exibido em `pointField`.

A linha 115 chama `displayMessage` (definido nas linhas 141 a 161) para exibir o estado atual do jogo. A estrutura `if/else` na linha 144 utiliza o método `showStatus` do *applet* para exibir um `String` na barra de estado do contêiner do *applet*. A linha 145 exibe

`Roll again.`

se `gameStatus` for igual a `CONTINUE`. As linhas 150 a 151 exibem

`Player wins. Click Roll Dice to play again.`

se `gameStatus` é igual a `WON`. As linhas 153 a 154 exibem

`Player loses. Click Roll Dice to play again.`

se `gameStatus` é igual a `LOST`. O método `showStatus` recebe um argumento `String` e o exibe na barra de estado do contêiner de *applets*. Se o jogo terminou (i.e., foi ganho ou perdido), a linha 158 configura `firstRoll` como `true` para indicar que o próximo lançamento dos dados é a primeira jogada do próximo jogo.

O programa espera o usuário clicar no botão `Roll Dice` novamente. Toda vez que o usuário pressiona `Roll Dice`, o método `actionPerformed` invoca o método `rollDice` para produzir uma nova `sumOfDice`. Se a jogada atual é uma continuação de um jogo incompleto, o código nas linhas 103 a 112 é executado. Na linha 107, se `sumOfDice` coincidir com `myPoint`, a linha 108 configura `gameStatus` como `WON`, e o jogo está terminado. Na linha 110, se `sumOfDice` for igual a 7, a linha 111 configura `gameStatus` como `LOST` e o jogo está terminado. Quando o jogo termina, `displayMessage` exibe uma mensagem apropriada e o usuário pode clicar no botão `Roll Dice` para iniciar um novo jogo. Ao longo do programa, os quatro `JTextFields` são atualizados com os novos valores dos dados e a soma em cada jogada, e o `pointField` é atualizado toda vez que um novo jogo é iniciado.

Observe a utilização interessante dos vários mecanismos de controle de programa já discutidos. O programa de jogo de dados utiliza quatro métodos – `init`, `actionPerformed`, `rollDice` e `displayMessage` – e as estruturas aninhadas `switch`, `if/else` e `if`. Observe também o uso de múltiplos rótulos `case` na estrutura `switch` para executar as mesmas instruções (linhas 79 e 85). Note, também, que o mecanismo de tratamento de eventos age como uma forma de controle de programa. Neste programa, o tratamento de eventos permite usar repetição controlada pelo usuário – cada vez que o usuário clica em `Roll Dice`, o programa lança os dados novamente. Nos exercícios, investigamos várias características interessantes da execução do jogo de *craps*.

6.10 Duração dos identificadores

Os Capítulos 2 a 5 utilizam identificadores para nomes de variáveis e de referências. Os atributos de variáveis e de referências incluem nome, tipo, tamanho e valor. Nós também utilizamos identificadores como nomes para classes e métodos definidos pelo usuário. Na verdade, cada identificador em um programa tem outros atributos, incluindo a *duração* e o *escopo*.

A *duração* de um identificador (também chamada de *tempo de vida*) é o período durante o qual esse identificador existe na memória. Alguns identificadores existem por curtos períodos de tempo e outros existem durante toda a execução de um programa.

O *escopo* de um identificador é onde o identificador pode ser usado em um programa. Alguns identificadores podem ser usados ao longo de todo o programa, enquanto outros podem ser usados apenas em partes limitadas do programa. Esta seção discute a duração dos identificadores. A Seção 6.11 discute o escopo dos identificadores.

Os identificadores que representam variáveis locais em um método (isto é, parâmetros e variáveis declaradas no corpo do método) têm *duração automática*. As variáveis de duração automática são criadas quando o controle de um programa atinge sua declaração; elas existem enquanto o bloco em que estão declaradas está ativo e são destruídas quando o bloco em que são declaradas termina. Continuaremos a nos referir às variáveis de duração automática como *variáveis locais*.

Dica de desempenho 6.2



A duração automática é uma maneira de economizar memória, porque as variáveis de duração automática são criadas quando o controle do programa atinge sua declaração e são destruídas quando o controle do programa sai do bloco em que são declaradas.

As variáveis de instância de uma classe são automaticamente inicializadas pelo compilador se o programador não fornecer valores iniciais explícitos. As variáveis dos tipos de dados primitivos são inicializadas com zero, exceto as variáveis **boolean**, que são inicializadas com **false**. As referências são inicializadas com **null**. Diferentemente das variáveis de instância de uma classe, as variáveis automáticas devem ser inicializadas pelo programador antes de poderem ser utilizadas.

Dica de teste e depuração 6.2



Se uma variável automática não é inicializada antes de ser utilizada em um método, o compilador emite uma mensagem de erro.

Java também tem identificadores de *duração estática*. As variáveis e referências de duração estática existem a partir do ponto em que a classe que as define é carregada na memória para execução até o término do programa. O espaço para seu armazenamento é alocado e inicializado quando suas classes são carregadas na memória. Embora os nomes de variáveis e referências de duração estática existam quando suas classes são carregadas na memória, esses identificadores não podem, necessariamente, ser utilizados em todo o programa. Duração (o tempo de vida de um identificador) e escopo (onde um nome pode ser utilizado) são questões separadas, como mostrado na Seção 6.11.

Observação de engenharia de software 6.9



A duração automática é um exemplo do princípio do privilégio mínimo. Este princípio afirma que cada componente de um sistema deve ter direitos e privilégios suficientes para executar a tarefa a ele designada, mas nenhum direito ou privilégio adicional. Esta restrição ajuda a evitar a ocorrência de erros acidentais e/ou maliciosos nos sistemas. Por que manter variáveis armazenadas e acessíveis na memória quando elas não são necessárias?

6.11 Regras de escopo

O *escopo* de um identificador para uma variável, referência ou método é a parte do programa que pode fazer referência ao identificador. Uma variável ou referência local declarada em um bloco pode ser utilizada somente nesse bloco ou em blocos aninhados dentro desse bloco. Os escopos para um identificador são o *escopo de classe* e *escopo de bloco*. Há também um escopo especial para rótulos utilizado com as instruções **break** e **continue** (apresentadas no Capítulo 5). O rótulo é visível somente no corpo da estrutura de repetição que vem logo após o rótulo.

Os métodos e as variáveis de instância de uma classe têm *escopo de classe*. O escopo de classe inicia na chave esquerda de abertura, {, da definição de classe e termina na chave direita de fechamento, }, da definição de classe. O escopo de classe permite que os métodos de uma classe invoquem diretamente todos os métodos definidos nessa mesma classe ou herdados por essa classe (como os métodos herdados por nossos *applets* da classe **JApplet**) e acessem diretamente todas as variáveis de instância definidas na classe. No Capítulo 8, veremos que os métodos **static** são uma exceção a essa regra. Em certo sentido, todas as variáveis de instância e os métodos de uma classe são *globais* para os métodos da classe em que são definidos (isto é, os métodos podem modificar as variáveis de instância diretamente e invocar outros métodos da classe). [Nota: uma das razões por que utilizamos principalmente *applets* neste capítulo é simplificar nossas discussões. Não apresentamos até agora um verdadeiro aplicativo com janelas em que os métodos da classe de nosso aplicativo tenham acesso a todos os outros métodos da classe e às variáveis de instância da classe.]

Os identificadores declarados dentro de um bloco têm *escopo de bloco*. O escopo de bloco inicia na declaração do identificador e termina na chave direita de fechamento () do bloco. As variáveis locais de um método têm escopo de bloco, da mesma forma que os parâmetros do método, que também são variáveis locais do método. Qualquer bloco pode conter declarações de variáveis ou de referências. Quando os blocos são aninhados no corpo de um método e um identificador declarado em um bloco externo tem o mesmo nome que um identificador declarado em um bloco interno, o compilador gera um erro de sintaxe dizendo que a variável já está definida. Se uma variável local em um método tem o mesmo nome que uma variável de instância, a variável de instância fica “escondida” até que o bloco termine a execução. No Capítulo 8, discutimos como acessar essas variáveis de instância “escondidas”.

Erro comum de programação 6.14



Utilizar accidentalmente, para um identificador em um bloco interno de um método, o mesmo nome que aquele utilizado para um identificador em um bloco externo do mesmo método resulta em um erro de sintaxe do compilador.



Boa prática de programação 6.7

Evite nomes de variáveis locais que ocultam nomes de variáveis de instância. Isso pode ser feito evitando-se o uso de identificadores duplicados em uma classe.

O applet da Fig. 6.10 demonstra questões de escopo com variáveis de instância e variáveis locais.

Esse exemplo utiliza o método **start** do applet (linhas 27 a 40) pela primeira vez. Lembre-se de que, quando um contêiner de *applets* carrega um *applet*, o contêiner cria primeiro uma instância da classe do *applet*. Ele então chama os métodos **init**, **start** e **paint** do *applet*. O método **start** sempre é definido com a linha mostrada na linha 27 como sua primeira linha.

```

1 // Fig. 6.10: Scoping.java
2 // Um exemplo de escopo
3
4 // Pacotes do núcleo de Java
5 import java.awt.Container;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class Scoping extends JApplet {
11     JTextArea outputArea;
12
13     // variável de instância acessível a todos os métodos desta classe
14     int x = 1;
15
16     // configura a GUI do applet
17     public void init()
18     {
19         outputArea = new JTextArea();
20         Container container = getContentPane();
21         container.add( outputArea );
22
23     } // fim do método init
24
25     // método start chamado após o término de init;
26     // start chama os métodos useLocal e useInstance
27     public void start()
28     {
29         int x = 5; // variável local para o método start
30
31         outputArea.append( "local x in start is " + x );
32
33         useLocal();      // useLocal tem x local
34         useInstance();   // useInstance utiliza a variável de instância x
35         useLocal();      // useLocal reinicializa x local
36         useInstance();   // a variável de instância x retém seu valor
37
38         outputArea.append( "\n\nlocal x in start is " + x );
39
40     } // fim do método start
41
42     // useLocal reinicializa a variável local x durante cada chamada
43     public void useLocal()
44     {
45         int x = 25; // inicializado toda vez que useLocal é chamado
46
47         outputArea.append( "\n\nlocal x in useLocal is " + x +
48             " after entering useLocal" );

```

Fig. 6.10 Um exemplo de escopo (parte 1 de 2).

```

49      +++;  

50      outputArea.append( "\nlocal x in useLocal is " + x +  

51          " before exiting useLocal" );  

52  

53  } // fim do método useLocal  

54  

55 // useInstance modifica a variável de instância x durante cada chamada  

56 public void useInstance()  

57 {  

58     outputArea.append( "\n\ninstance variable x is " + x +  

59                     " on entering useInstance" );  

60     x *= 10;  

61     outputArea.append( "\ninstance variable x is " + x +  

62                     " on exiting useInstance" );  

63  

64 } // fim do método useInstance  

65  

66 } // fim da classe Scoping

```

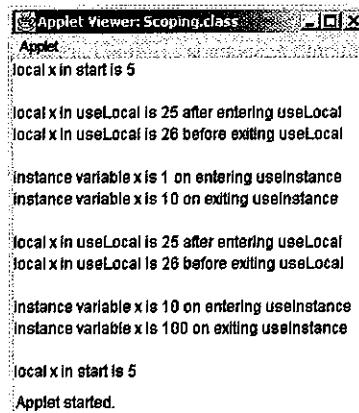


Fig. 6.10 Um exemplo de escopo (parte 2 de 2).

A linha 14 declara a variável de instância **x** e a inicializa com 1. Essa variável de instância fica oculta em qualquer bloco (ou método) que declare uma variável chamada **x**. O método **start** declara a variável local **x** (linha 29) e a inicializa com 5. O valor dessa variável é exibido na **JTextArea outputArea** para mostrar que a variável de instância **x** fica oculta em **start**. O programa define dois outros métodos – **useLocal** (linhas 43 a 53) e **useInstance** (linhas 56 a 64) – cada um deles não aceita nenhum argumento e não devolve resultados. O método **start** chama cada um desses métodos duas vezes. O método **useLocal** define a variável local (automática) **x** (linha 45) e inicializa **x** com 25, exibe o valor de **x** em **outputArea**, incrementa **x** e exibe novamente o valor de **x**. Quando **useLocal** é chamado novamente (linha 35), ele recria a variável local **x** e a inicializa com 25. O método **useInstance** não declara nenhuma variável. Portanto, quando faz referência à variável **x**, utiliza-se a variável de instância **x**. Quando **useInstance** é chamado (linha 34), ele exibe a variável de instância **x** em **outputArea**, multiplica a variável de instância **x** por 10 e exibe a variável de instância **x** novamente antes de retornar. Da próxima vez que o método **useInstance** é chamado (linha 36), a variável de instância tem seu valor modificado, 10. Por fim, o programa exibe a variável local **x** em **start** novamente para mostrar que nenhuma das chamadas de método modificou o valor de **x**, porque todos os métodos faziam referência às variáveis em outros escopos.

6.12 Recursão

Os programas que discutimos até agora são geralmente estruturados como métodos que chamam uns aos outros de uma maneira disciplinada e hierárquica. Para alguns problemas, no entanto, é útil fazer os métodos chamarem a si

mesmos. O *método recursivo* é um método que chama a si próprio diretamente ou indiretamente, através de outro método. A recursão é um tópico importante discutido demoradamente nos cursos de ciência da computação de nível superior. Nesta seção e na próxima, serão apresentados exemplos simples de recursão. Este livro contém um extenso tratamento de recursão. A Fig. 6.15 (no final da Seção 6.14) resume os exemplos e exercícios de recursão deste livro.

Primeiro, analisamos a recursão de maneira conceitual. Depois examinamos vários programas que contêm métodos recursivos. As abordagens da solução de problemas com recursão têm diversos elementos em comum. O método recursivo é chamado para resolver um problema. O método realmente sabe como resolver somente o(s) caso(s) mais simples, ou o(s) caso(s) *básico(s)*. Se o método é chamado com um caso básico, o método devolve um resultado. Se o método é chamado com um problema mais complexo, o método divide o problema em dois pedaços conceituais: um pedaço que o método sabe como resolver (caso básico) e um pedaço que o método não sabe como resolver. Para tornar a recursão factível, o segundo pedaço deve se assemelhar ao problema original, mas ser uma versão um pouco mais simples ou um pouco menor do problema original. Como esse novo problema se parece com o problema original, o método invoca (chama) uma nova cópia de si mesmo para ir trabalhar no problema menor; este procedimento, conhecido como *chamada recursiva*, também é chamado de *etapa de recursão*. A etapa de recursão também inclui normalmente a palavra-chave **return**, porque seu resultado será combinado com a parte do problema que o método sabia como resolver para formar um resultado que será passado de volta para o chamador original.

A etapa de recursão é executada enquanto a chamada original para o método ainda está ativa (isto é, enquanto sua execução ainda não terminou). A etapa de recursão pode resultar em muitas outras chamadas recursivas, à medida que o método divide cada subproblema novo em dois pedaços conceituais. Para que a recursão termine em algum momento, toda vez que o método chama a si próprio com uma versão um pouco mais simples do problema original, a seqüência de problemas cada vez menores deve convergir para o caso básico. Nesse ponto, o método reconhece o caso básico, devolve um resultado para cópia anterior do método e uma seqüência de devoluções segue para cima até a chamada de método original por fim devolver o resultado final para o chamador. Este processo soa exótico quando comparado à solução convencional de problemas que executamos até este ponto. Como um exemplo desses conceitos em operação, vamos escrever um programa recursivo para realizar um cálculo matemático popular.

O fatorial de um inteiro n não-negativo, escrito $n!$ (e pronunciado como “fatorial de n ”), é o produto

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

com $1! = 1$ e $0! = 1$ definido como 1. Por exemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que é igual a 120.

O fatorial de um inteiro, **number**, maior que ou igual a 0, pode ser calculado *iterativamente* (não recursivamente) utilizando a estrutura **for** como segue:

```
factorial = 1;
for ( int counter = number; counter >= 1; counter-- )
    factorial *= counter;
```

Chega-se a uma definição recursiva do método fatorial observando-se o seguinte relacionamento:

$$n! = n \cdot (n - 1)!$$

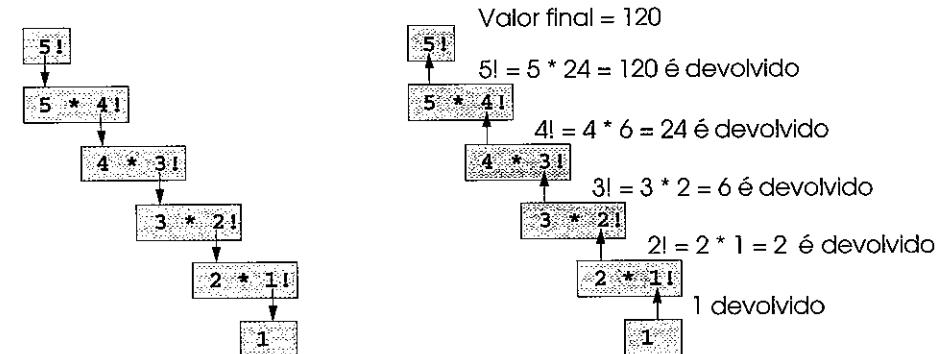
Por exemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, como mostrado a seguir:

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\ 5! &= 5 \cdot (4!) \end{aligned}$$

A avaliação de $5!$ prosseguiria como mostrado na Fig. 6.11. A Fig. 6.11 (a) mostra como a sucessão de chamadas recursivas prossegue até que $1!$ seja avaliado como 1, o que termina a recursão. A Fig. 6.11 (b) mostra os valores devolvidos para cada chamada recursiva para seu chamador até que o valor final seja calculado e devolvido.

A Fig. 6.12 utiliza recursão para calcular e imprimir os fatoriais dos inteiros de 0 a 10 (a escolha do tipo de dados **long** será explicada em breve). O método recursivo **factorial** (linhas 29 a 39) primeiro testa para ver se uma condição de término é verdadeira. Se **number** é menor que ou igual a 1 (o caso básico), **factorial** devolve 1, nenhuma recursão adicional é necessária e o método retorna. Se **number** for maior que 1, a linha 37 expressa o problema como o produto de **number** por uma chamada recursiva a **factorial** que avalia o fatorial de **num-**

ber - 1. Observe que `factorial(number - 1)` é um problema ligeiramente mais simples que o cálculo original `factorial(number)`.



(a) Seqüência de chamadas recursivas. (b) Valores devolvidos de cada chamada recursiva.

Fig. 6.11 Avaliação recursiva de 5!.

```

1 // Fig. 6.12: FactorialTest.java
2 // Método recursivo para cálculo de fatorial
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class FactorialTest extends JApplet {
11     JTextArea outputArea;
12
13     // inicializa o applet, criando a GUI e calculando os fatoriais
14     public void init()
15     {
16         outputArea = new JTextArea();
17
18         Container container = getContentPane();
19         container.add( outputArea );
20
21         // calcula os fatoriais de 0 a 10
22         for ( long counter = 0; counter <= 10; counter++ )
23             outputArea.append( counter + "!" + =
24                 factorial( counter ) + "\n" );
25
26     } // fim do método init
27
28     // Definição recursiva do método factorial
29     public long factorial( long number )
30     {

```

Fig. 6.12 Calculando fatoriais com um método recursivo (parte 1 de 2).

```

31     // caso básico
32     if ( number <= 1 )
33         return 1;
34
35     // etapa de recursão
36     else
37         return number * factorial( number - 1 );
38
39 } // fim do método factorial
40
41 } // fim da classe FactorialTest

```

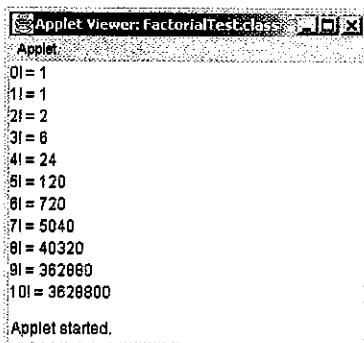


Fig. 6.12 Calculando fatoriais com um método recursivo (parte 2 de 2).

O método **factorial** (linha 29) recebe um parâmetro do tipo **long** e devolve um resultado do tipo **long**. Como pode ser visto na Fig. 6.12, os valores de fatoriais tornam-se grandes rapidamente. Escolhemos o tipo de dados **long** de modo que o programa possa calcular fatoriais maiores que $20!$. Infelizmente, o método **factorial** produz valores grandes tão rapidamente que mesmo esse **long** não nos ajuda a imprimir muitos valores de fatoriais antes que os valores excedam o tamanho que pode ser armazenado em uma variável **long**.

Nos exercícios, exploramos o fato de que **float** e **double** podem acabar sendo necessários para os usuários que desejam calcular fatoriais de números grandes. Essa situação aponta para uma fraqueza na maioria das linguagens de programação, ou seja, as linguagens não são facilmente estendidas para tratar os requisitos específicos de vários aplicativos. Como veremos no Capítulo 9, Java é uma linguagem extensível que permite criar inteiros arbitrariamente grandes se desejarmos. De fato, o pacote **java.math** fornece duas classes – **BigInteger** e **BigDecimal** – explicitamente para cálculos matemáticos de precisão arbitrária que não podem ser representados com os tipos de dados primitivos de Java.



Erro comum de programação 6.15

Omitir o caso básico ou escrever a etapa de recursão incorretamente, de modo que ela não convirja para o caso básico, causará recursão infinita, esgotando a memória em algum momento. Esse erro é análogo ao problema de um laço infinito em uma solução iterativa (não-recursiva).

6.13 Exemplo que utiliza recursão: a série de Fibonacci

A série de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inicia com 0 e 1 e tem a propriedade de que cada número de Fibonacci subsequente é a soma dos dois números que o precedem.

A série ocorre na natureza e, em particular, descreve uma forma de espiral. A relação entre os números sucessivos de Fibonacci converge para um valor constante de 1,618.... Esse número também ocorre na natureza e foi chamado de *relação áurea* ou *média áurea*. Os seres humanos tendem a achar a média áurea esteticamente agradável. Os arquitetos freqüentemente projetam janelas, salas e edifícios cuja relação entre comprimento e largura é a rela-

ção áurea. Os cartões-postais freqüentemente são projetados com uma relação entre comprimento/largura igual à relação áurea.

A série de Fibonacci pode ser definida recursivamente como segue:

```
fibonacci (0) = 0
fibonacci (1) = 1
fibonacci (n) = fibonacci (n - 1) + fibonacci (n - 2)
```

Observe que há dois casos básicos para o cálculo de Fibonacci – fibonacci (0) é definido como 0 e fibonacci (1) é definido como 1. O *applet* da Fig. 6.13 calcula recursivamente o *i*-ésimo número de Fibonacci, utilizando o método **fibonacci** (linhas 68 a 78). O *applet* permite que o usuário digite um inteiro em um **JTextField**. O valor digitado indica o *i*-ésimo número de Fibonacci a calcular. Quando o usuário pressiona a tecla *Enter*, o método **actionPerformed** é executado em resposta ao evento de interface com o usuário e chama o método **fibonacci** para calcular o número de Fibonacci especificado. Observe que os números de Fibonacci tendem a se tornar grandes rapidamente. Portanto, utilizamos tipo de dados **long** para o tipo de parâmetro e o tipo do valor devolvido no método **fibonacci**. Na Fig. 6.13, as capturas de tela mostram os resultados do cálculo de vários números de Fibonacci.

Mais uma vez, o método **init** desse *applet* cria os componentes GUI e os anexa ao painel de conteúdo do *applet*. O gerenciador de layout para o painel de conteúdo é configurado como **FlowLayout** na linha 22.

O tratamento de evento nesse exemplo é semelhante ao tratamento de evento do *applet Craps* na Fig. 6.9. A linha 34 especifica que o *applet this* deve ouvir eventos do **JTextField numberField**. Lembre-se, a palavra-chave **this** permite que o *applet* faça referência a si próprio. Então, na linha 34, o *applet* está dizendo a **numberField** que o *applet* deve ser notificado (com uma chamada ao método **actionPerformed** do *applet*) quando um evento de ação ocorre em **numberField**. Nesse exemplo, o usuário pressiona a tecla *Enter* quando digita no **JTextField numberField** para gerar o evento de ação. Uma mensagem é então enviada para o *applet* (isto é, um método – **actionPerformed** – é chamado para o *applet*) indicando que o usuário do programa interagiu com um dos componentes GUI do programa (**numberField**). Lembre-se de que a instrução para registrar o *applet* como ouvinte de **numberField** só compilará se a classe do *applet* também implementar **ActionListener** (linha 12).

```

1 // Fig. 6.13: FibonacciTest.java
2 // Método recursivo fibonacci
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class FibonacciTest extends JApplet
12     implements ActionListener {
13
14     JLabel numberLabel, resultLabel;
15     JTextField numberField, resultField;
16
17     // configura a GUI do applet
18     public void init()
19     {
20         // obtém painel de conteúdo e configura seu layout para FlowLayout
21         Container container = getContentPane();
22         container.setLayout( new FlowLayout() );
23
24         // cria numberLabel e o anexa ao painel de conteúdo
25         numberLabel =
26             new JLabel( "Enter an integer and press Enter" );
27         container.add( numberLabel );

```

Fig. 6.13 Gerando números de Fibonacci recursivamente (parte 1 de 3).

```

28
29     // cria numberField e o anexa ao painel de conteúdo
30     numberField = new JTextField( 10 );
31     container.add( numberField );
32
33     // registra este applet como um ActionListener de numberField
34     numberField.addActionListener( this );
35
36     // cria resultLabel e o anexa ao painel de conteúdo
37     resultLabel = new JLabel( "Fibonacci value is" );
38     container.add( resultLabel );
39
40     // cria numberField, torna-o não-editável
41     // e o anexa ao painel de conteúdo
42     resultField = new JTextField( 15 );
43     resultField.setEditable( false );
44     container.add( resultField );
45
46 } // fim do método init
47
48 // obtém dados de entrada do usuário e chama o método fibonacci
49 public void actionPerformed( ActionEvent e )
50 {
51     long number, fibonacciValue;
52
53     // obtém dados digitados pelo usuário e converte para long
54     number = Long.parseLong( numberField.getText() );
55
56     showStatus( "Calculating ..." );
57
58     // calcula valor de Fibonacci para o número digitado pelo usuário
59     fibonacciValue = fibonacci( number );
60
61     // indica fim do processamento e exibe resultado
62     showStatus( "Done." );
63     resultField.setText( Long.toString( fibonacciValue ) );
64
65 } // fim do método actionPerformed
66
67 // Definição recursiva do método fibonacci
68 public long fibonacci( long n )
69 {
70     // caso básico
71     if ( n == 0 || n == 1 )
72         return n;
73
74     // etapa de recursão
75     else
76         return fibonacci( n - 1 ) + fibonacci( n - 2 );
77
78 } // fim do método fibonacci
79
80 } // fim da classe FibonacciTest

```

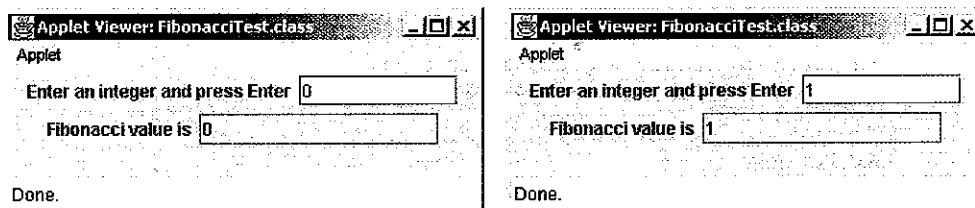


Fig. 6.13 Gerando números de Fibonacci recursivamente (parte 2 de 3).

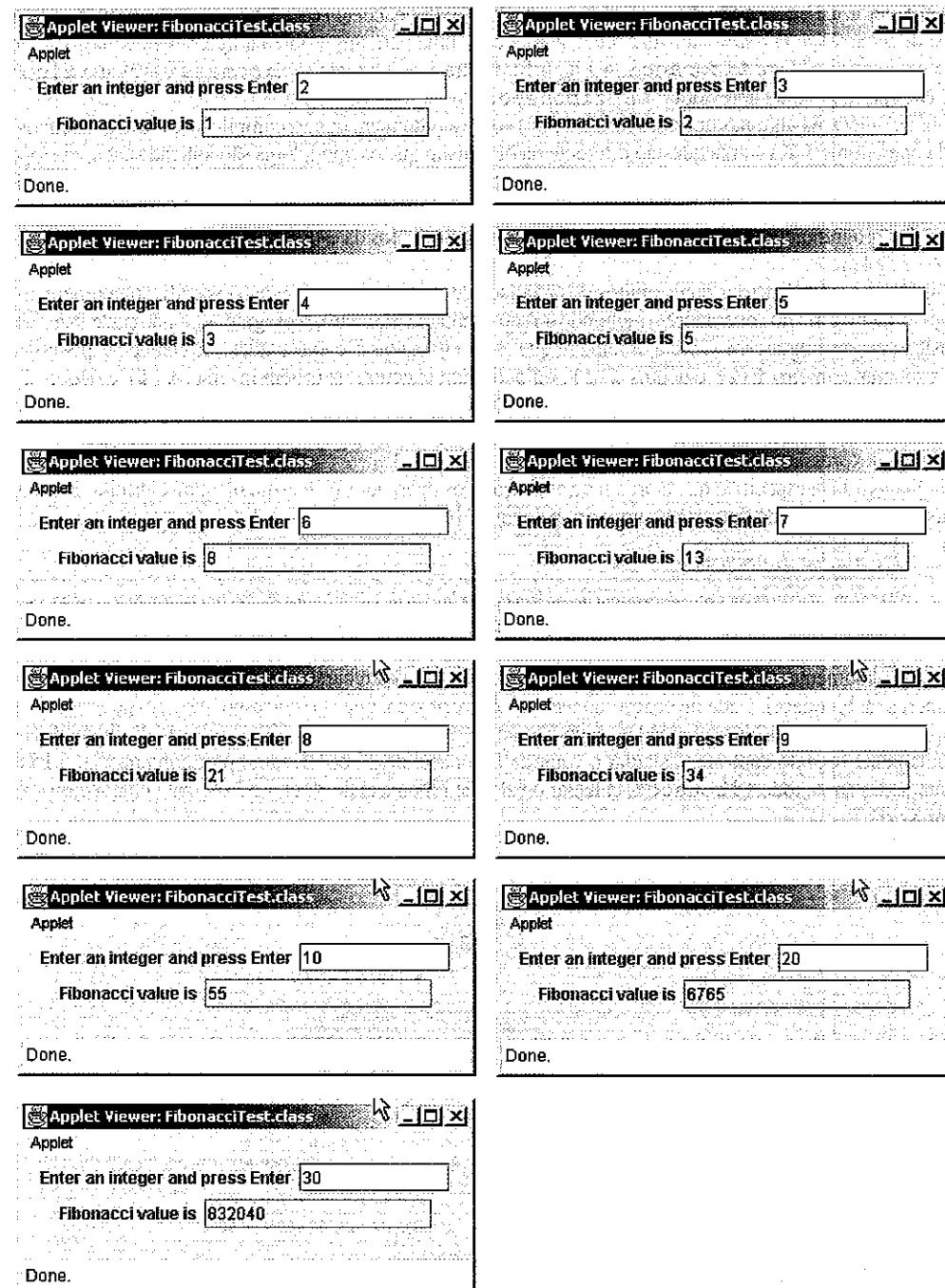


Fig. 6.13 Gerando números de Fibonacci recursivamente (parte 3 de 3).

A chamada para `fibonacci` (linha 59) a partir de `actionPerformed` não é uma chamada recursiva, mas todas as chamadas subsequentes a `fibonacci` executadas a partir do corpo de `fibonacci` são recursivas. Cada

vez que `fibonacci` é invocado, ele imediatamente testa quanto ao caso básico – `n` igual a 0 ou 1. Se essa condição for verdadeira, `fibonacci` devolve `n` (`fibonacci(0)` é 0 e `fibonacci(1)` é 1). Curiosamente, se `n` for maior que 1, a etapa de recursão gera *duas* chamadas recursivas, e cada uma para um problema ligeiramente mais simples do que a chamada original para `fibonacci`. A Fig. 6.14 mostra como o método `fibonacci` avaliaria `fibonacci(3)`. Na figura, `f` é uma abreviatura para `fibonacci`.

A Fig. 6.14 levanta algumas questões interessantes sobre a ordem em que os compiladores Java avaliam os operandos de operadores. Essa é uma questão diferente da ordem em que os operadores são aplicados aos seus operandos, que é a ordem ditada pelas regras de precedência de operadores. A Fig. 6.14 mostra que, durante a avaliação de `f(3)`, serão feitas duas chamadas recursivas, a saber `f(2)` e `f(1)`. Mas em que ordem são feitas essas chamadas? A maioria dos programadores assume que os operandos serão avaliados da esquerda para a direita. Em Java essa suposição é verdadeira.

As linguagens C e C++ (nas quais muitos dos recursos de Java são baseados) não especificam a ordem em que os operandos da maioria dos operadores (incluindo `+`) são avaliados. Portanto, o programador não pode fazer nenhuma suposição nessas linguagens sobre a ordem em que essas chamadas são executadas. As chamadas poderiam, de fato, ser executar primeiro `f(2)` e depois `f(1)`, ou poderiam executar na ordem inversa: `f(1)` e depois `f(2)`. Nesse programa e na maioria dos outros programas, descobre-se que o resultado final seria o mesmo para os dois casos. Mas, em alguns programas, a avaliação de um operando pode ter *efeitos colaterais* que poderiam afetar o resultado final da expressão.

A linguagem Java especifica que a ordem de avaliação dos operandos é da esquerda para a direita. Portanto, as chamadas do método são, de fato, `f(2)` primeiro e depois `f(1)`.



Boa prática de programação 6.8

Não escreva expressões que dependam da ordem de avaliação dos operandos de um operador. O uso de tais expressões freqüentemente resulta em programas que são difíceis de ler, depurar, modificar e manter.

É um bom momento para se dar um conselho sobre os programas recursivos como o que utilizamos aqui para gerar números de Fibonacci. Cada invocação do método `fibonacci` que não corresponde a um dos casos básicos (i.e., 0 ou 1) resulta em mais duas chamadas recursivas ao método `fibonacci`. Esse conjunto de chamadas recursivas rapidamente foge do controle. Calcular o valor de Fibonacci de 20 utilizando o programa na Fig. 6.13 exige 21.891 chamadas ao método `fibonacci`; calcular o valor de Fibonacci de 30 exige 2.692.537 chamadas ao método `fibonacci`.

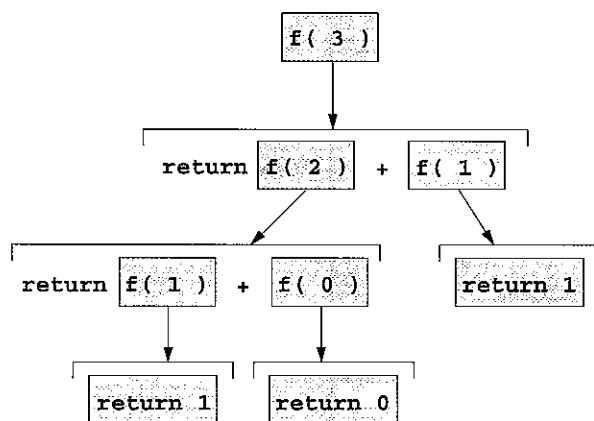


Fig. 6.14 Conjunto de chamadas recursivas para o método `fibonacci` (`f` neste diagrama).

À medida que tentar calcular valores de Fibonacci maiores, você notará que cada número de Fibonacci consecutivo que você solicita que o *applet* calcule resulta em um aumento substancial no tempo de cálculo e no número de chamadas ao método `fibonacci`. Por exemplo, o valor de Fibonacci de 31 exige 4.356.617 chamadas e o va-

lor de Fibonacci de 32 exige 7.049.155 chamadas. Como você pode ver, o número de chamadas para Fibonacci está aumentando rapidamente – 1.664.080 chamadas adicionais entre os valores de Fibonacci de 30 e 31 e 2.692.538 chamadas adicionais entre valores de Fibonacci de 31 e 32. Essa diferença no número de chamadas feitas entre os valores de Fibonacci de 31 e 32 é mais que 1,5 vez o número de chamadas para os valores de Fibonacci entre 30 e 31. Problemas dessa natureza humilham até mesmo os computadores mais poderosos do mundo! No campo da teoria da complexidade, os cientistas de computação estudam como os algoritmos têm de trabalhar duro para completar suas tarefas. As questões de complexidade são discutidas em detalhes nas disciplinas do currículo de ciência da computação de nível superior, geralmente chamadas de “Algoritmos”.

Dica de desempenho 6.3



Evite programas recursivos no estilo de Fibonacci, que resultam em uma “explosão” exponencial de chamadas.

Dica de teste e depuração 6.3



Tente melhorar o programa Fibonacci da Fig. 6.13 de modo que ele calcule a quantidade aproximada de tempo necessária para executar o cálculo. Para este fim, chame o método `static getCurrentTimeMillis` da classe `System`, o qual não recebe argumentos e devolve a hora atual do computador em milissegundos. Chame este método duas vezes – uma antes da chamada para `fibonacci` e uma depois da chamada para `fibonacci`. Salve cada um destes valores e calcule a diferença entre as horas para determinar quantos milissegundos foram necessários para executar o cálculo. Exiba este resultado.

6.14 Recursão versus iteração

Nas seções anteriores, estudamos dois métodos que podem ser facilmente implementados recursiva ou iterativamente. Nesta seção, comparamos as duas abordagens e discutimos por que o programador talvez escolha uma abordagem em vez da outra em uma situação particular.

Tanto a iteração como a recursão se baseiam em uma estrutura de controle: a iteração utiliza uma estrutura de repetição (como `for`, `while` ou `do/while`); a recursão utiliza uma estrutura de seleção (como `if`, `if/else` ou `switch`). Tanto a iteração quanto a recursão envolvem repetição: a iteração utiliza explicitamente uma estrutura de repetição; a recursão consegue repetição através de chamadas para métodos repetidas. A iteração e a recursão envolvem um teste de terminação: a iteração termina quando a condição de continuação do laço falha; a recursão termina quando um caso básico é reconhecido. A iteração com repetição controlada por contador e a recursão se aproximam gradualmente do término: a iteração continua modificando um contador até que o contador assuma um valor que faz a condição de continuação do laço falhar; a recursão continua produzindo versões mais simples do problema original até que o caso básico seja alcançado. Tanto a iteração como a repetição podem ocorrer infinitamente: um laço infinito ocorre com a iteração se o teste de continuação do laço nunca se tornar falso; a recursão infinita ocorre se a etapa de recursão não reduz o problema a cada vez, de maneira que converja para o caso básico.

A recursão tem muitos pontos negativos. Ela invoca repetidamente o mecanismo e, consequentemente, a sobre-carga das chamadas de método. Isso pode ser caro tanto em termos de tempo de processador como em espaço de memória. Cada chamada recursiva faz com que outra cópia do método (na realidade, somente as variáveis do método) seja criada; esse conjunto de cópias pode consumir espaço de memória considerável. A iteração normalmente ocorre dentro de um método, de modo que a sobre-carga das chamadas de método repetidas e a atribuição extra de memória são evitadas. Por que, então, escolher a recursão?

Observação de engenharia de software 6.10



Qualquer problema que possa ser resolvido recursivamente também pode ser resolvido iterativamente (não recursivamente). Em geral, uma abordagem recursiva é escolhida preferencialmente a uma abordagem iterativa quando a abordagem recursiva espelha o problema mais naturalmente e resulta em um programa que é mais fácil de entender e depurar. Freqüentemente, uma abordagem recursiva pode ser implementada com poucas linhas de código e uma abordagem iterativa correspondente pode exigir quantidades maiores de código. Outra razão para escolher uma solução recursiva é que uma solução iterativa pode não ser visível.

Dica de desempenho 6.4



Evite utilizar recursão em situações em que o desempenho é importante. As chamadas recursivas levam tempo e consomem memória adicional.



Erro comum de programação 6.16

Ter accidentalmente um método não-recursivo chamando a si próprio, seja direta ou indiretamente (por outro método), pode causar recursão infinita.

A maioria dos livros sobre programação apresenta a recursão muito mais tarde do que fizemos aqui. Acreditamos que a recursão seja um tópico suficientemente rico e complexo para que seja melhor apresentá-lo mais cedo e espalhar os exemplos dele no restante do texto. A Fig. 6.15 resume os exemplos e os exercícios de recursão no texto.

Capítulo	Exemplos e exercícios com recursão
6	Método factorial Método de Fibonacci Máximo divisor comum Soma de dois inteiros Multiplicação de dois inteiros Elevar um inteiro a uma potência inteira Torres de Hanói Visualizando a recursão
7	Soma dos elementos de um <i>array</i> Impressão de um <i>array</i> Impressão de um <i>array</i> de trás para frente Verificando se um <i>string</i> é um palíndromo Valor mínimo em um <i>array</i> Classificação por seleção Oito Rainhas Pesquisa linear Pesquisa binária Classificação pelo método <i>Quicksort</i> Percorrendo um labirinto
10	Imprimindo de trás para frente um <i>string</i> lido do teclado
19	Inserção em lista encadeada Exclusão em lista encadeada Pesquisa em uma lista encadeada Impressão de uma lista encadeada de trás para frente Inserção em árvore binária Percorrendo uma árvore binária na pré-ordem Percorrendo uma árvore binária na ordem Percorrendo uma árvore binária na pós-ordem

Fig. 6.15 Resumo de exemplos e exercícios com recursão no texto.

Vamos reconsiderar algumas observações que fazemos repetidamente em todo o livro. A boa engenharia de *software* é importante. O alto desempenho também é geralmente importante. Infelizmente, com freqüência esses objetivos são incompatíveis um com o outro. A boa engenharia de *software* é fundamental para tornar mais gerenciável a tarefa de desenvolver sistemas de *software* maiores e mais complexos. O alto desempenho nesses sistemas é fundamental para se imaginar os sistemas do futuro, que colocarão cargas de computação cada vez maiores sobre o *hardware*. Onde se enquadram os métodos aqui?



Observação de engenharia de software 6.11

Modularizar programas de maneira organizada e hierárquica promove a boa engenharia de software. Mas isso tem seu preço.

Dica de desempenho 6.5

Um programa pesadamente modularizado – em comparação com um programa monolítico (isto é, um bloco só) sem métodos – faz quantidades potencialmente grandes de chamadas de métodos, e estas consomem tempo de execução e espaço no(s) processador(es) de um computador. Mas programas monolíticos são difíceis de programar, testar, depurar, manter e desenvolver.

Portanto, modularize seus programas criteriosamente, sempre mantendo em mente o delicado equilíbrio entre desempenho e boa engenharia de software.

6.15 Sobrecarga de métodos

Java permite que vários métodos com o mesmo nome sejam definidos, contanto que esses métodos tenham conjuntos diferentes de parâmetros (com base na quantidade, nos tipos e na ordem dos parâmetros). Esse recurso é chamado de *sobrecarga de método*. Quando se chama um método sobrecarregado, o compilador Java seleciona o método adequado examinando a quantidade, os tipos e a ordem dos argumentos na chamada. A sobrecarga de métodos é comumente utilizada para criar vários métodos com o mesmo nome que realizam tarefas semelhantes, mas sobre tipos de dados diferentes.

**Boa prática de programação 6.9**

Sobreclarregar métodos que realizam tarefas intimamente relacionadas pode tornar os programas mais legíveis e comprehensíveis.

A Fig. 6.16 utiliza o método sobreclarregado **square** para calcular o quadrado de um **int** e o quadrado de um **double**.

```

1 // Fig. 6.16: MethodOverload.java
2 // Utilizando métodos sobreclarregados
3
4 // Pacotes do núcleo de Java
5 import java.awt.Container;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class MethodOverload extends JApplet {
11
12     // configura a GUI e chama as versões do método square
13     public void init()
14     {
15         JTextArea outputArea = new JTextArea();
16         Container container = getContentPane();
17         container.add( outputArea );
18
19         outputArea.setText(
20             "The square of integer 7 is " + square( 7 ) +
21             "\nThe square of double 7.5 is " + square( 7.5 ) );
22     }
23
24     // método square com argumento int
25     public int square( int intValue )
26     {
27         System.out.println(
28             "Called square with int argument: " + intValue );
29
30         return intValue * intValue;
31
32     } // fim do método square com argumento int

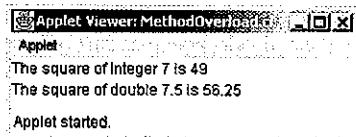
```

Fig. 6.16 Utilizando métodos sobreclarregados (parte 1 de 2).

```

33
34    // método square com argumento double
35    public double square( double doubleValue )
36    {
37        System.out.println(
38            "Called square with int argument: " + doubleValue );
39
40        return doubleValue * doubleValue;
41
42    } // fim do método square com argumento double
43
44 } // fim da classe MethodOverload

```



```

Called square with int argument: 7
Called square with double argument: 7.5

```

Fig. 6.16 Utilizando métodos sobrecarregados (parte 2 de 2).

Os métodos sobrecarregados são distinguidos por sua *assinatura* – uma combinação do nome do método e dos tipos de seus parâmetros. Se o compilador Java só olhasse os nomes dos métodos durante a compilação, o código na Fig. 6.16 seria ambíguo – o compilador não saberia como distinguir entre os dois métodos **square**. Logicamente, o compilador utiliza nomes “adulterados” ou “decorados”, mais longos, que incluem o nome original do método, o tipo de cada parâmetro e a ordem exata dos parâmetros, para determinar se os métodos em uma classe são únicos naquela classe.

Por exemplo, na Fig. 6.16, o compilador poderia utilizar o nome lógico “*square of int*” (quadrado de int) para o método **square** que especifica um parâmetro **int** e “*square of double*” (quadrado de double) para o método **square** que especifica um parâmetro **double**. Se a definição de um método **foo*** inicia como segue:

```
void foo( int a, float b )
```

o compilador talvez utilize o nome lógico “*foo of int and float*” (*foo de int e float*). Se os parâmetros são especificados como segue:

```
void foo( float a, int b )
```

o compilador talvez utilize o nome lógico “*foo of float and int*”. Observe que a ordem dos parâmetros é importante para o compilador. Os dois métodos **foo** precedentes são considerados distintos pelo compilador.

Os nomes lógicos de métodos utilizados pelo compilador não mencionaram os tipos dos valores devolvidos pelos métodos, porque os métodos não podem ser distinguidos por tipo do valor devolvido. O programa da Fig. 6.17 ilustra os erros de compilação gerados quando dois métodos têm a mesma assinatura e tipos dos valores devolvidos diferentes. Os métodos sobrecarregados podem ter tipos de valores devolvidos diferentes, mas devem ter listas de parâmetros diferentes. Além disso, os métodos sobrecarregados não precisam ter o mesmo número de parâmetros.



Erro comum de programação 6.17

Criar métodos sobrecarregados com listas de parâmetros idênticas e tipos de valores devolvidos diferentes é um erro de sintaxe.

* N. de T.: *foo* - Um *string* usado pelos programadores no lugar de informações mais específicas. O nome *foo* pode ser usado para apresentar variáveis ou funções contida em exemplos de código cujo objetivo é demonstrar a sintaxe, e em arquivos-rascunhos temporários. da mesma forma, um programador pode digitar *foo* para testar um tratador de entrada de *string*. Se um segundo *string* for necessário, normalmente ele será bar, sugerindo que a origem de ambos é FUBAR (um acrônimo que significa *Fooled Up Beyond All Recognition/Repair*), uma expressão muito utilizada no Exército dos Estados Unidos. (*Dicionário de Informática*. Microsoft Press. Rio de Janeiro, Editora Campus, 1998.)

```

1 // Fig. 6.17: MethodOverload.java
2 // Métodos sobrecarregados com assinaturas idênticas e
3 // tipos de valores devolvidos diferentes.
4
5 // Pacotes de extensão de Java
6 import javax.swing.JApplet;
7
8 public class MethodOverload extends JApplet {
9
10    // primeira definição do método square com argumento double
11    public int square( double x )
12    {
13        return x * x;
14    }
15
16    // segunda definição do método square com argumento double
17    // causa erro de sintaxe
18    public double square( double y )
19    {
20        return y * y;
21    }
22
23 } // fim da classe MethodOverload

```

```

MethodOverload.java:18: square(double) is already defined in
MethodOverload
    public double square( double y )
                           ^
MethodOverload.java:13: possible loss of precision
found   : double
required: int
    return x * x;
                           ^
2 errors

```

Fig. 6.17 Mensagens de erro do compilador geradas por métodos sobrecarregados com listas de parâmetros idênticas e tipos diferentes de valores devolvidos.

6.16 Métodos da classe JApplet

Escrevemos muitos *applets* até este ponto no texto, mas ainda não discutimos os métodos fundamentais da classe **JApplet** que o contêiner de *applets* chama durante a execução de um *applet*. A Fig. 6.18 lista os métodos fundamentais da classe **JApplet**, especifica quando são chamados e explica o propósito de cada método.

Esses métodos da classe **JApplet** são definidos na Java API para não fazer nada, a menos que você forneça uma definição para eles na definição de classe do seu *applet*. Se você quiser utilizar um desses métodos em um *applet* que você está definindo, você *deve* definir a primeira linha de cada método como mostrado na Fig. 6.18. Caso contrário, o contêiner de *applets* não chamará suas versões dos métodos durante a execução do *applet*. Definir os métodos como discutido aqui é conhecido como *sobrescrever* a definição original do método. O contêiner de *applets* vai chamar a versão sobreescrita de um método para seu *applet* antes de tentar chamar a versão *default* herdada de **JApplet**. A sobreescrita de métodos é discutida em detalhes no Capítulo 9.



Erro comum de programação 6.18

Fornecer uma definição para um dos métodos init, start, paint, stop ou destroy de JApplet que não corresponda aos cabeçalhos de método mostrados na Fig. 6.18 resulta em um método que não será chamado automaticamente durante a execução do applet.

Método Quando o método é chamado e seu propósito

public void init()

Esse método é chamado uma vez pelo `appletviewer` ou pelo navegador quando um *applet* é carregado para execução. Executa a inicialização de um *applet*. Ações típicas executadas aqui são a inicialização de variáveis de instância e dos componentes GUI do *applet* e a carga de sons para reproduzir ou imagens a exibir (ver Capítulo 18) e criação de *threads* (ver Capítulo 15).

public void start()

É chamado depois que o método `init` completa sua execução e toda a vez que o usuário do navegador retorna para a página HTML em que o *applet* reside (depois de navegar por outra página HTML). Esse método executa quaisquer tarefas que devam ser completadas quando o *applet* é carregado pela primeira vez no navegador e que deve ser executada cada vez que a página HTML em que o *applet* reside é revisitada. Entre as ações típicas realizadas aqui incluem-se iniciar uma animação (ver Capítulo 18) e iniciar outras *threads* de execução (ver Capítulo 15).

public void paint (Graphics g)

É chamado depois que o método `init` completou sua execução e o método `start` começou a ser executado para desenhar no *applet*. Também é chamado automaticamente cada vez que o *applet* precisa ser repintado*. Por exemplo, se o usuário do navegador cobre o *applet* com outra janela aberta na tela e depois descobre o *applet*, é chamado o método `paint`. Entre as ações típicas realizadas aqui está desenhar com o objeto `Graphics g` que é passado automaticamente para o método `paint` para você.

public void stop()

É chamado quando o *applet* deve parar de ser executado – normalmente quando o usuário do navegador deixa a página HTML em que o *applet* reside. Esse método executa quaisquer tarefas que são exigidas para suspender a execução do *applet*. Entre as ações típicas realizadas aqui estão parar a execução de animações e *threads*.

public void destroy()

É chamado quando o *applet* está sendo removido da memória – normalmente quando o usuário do navegador sai da sessão de navegação. Esse método executa quaisquer tarefas que são necessárias para destruir recursos alocados para o *applet*.

Fig. 6.18 Métodos de `JApplet` que o contêiner de *applets* chama durante a execução de um *applet*.

O método `repaint` também é de interesse para muitos programadores de *applets*. Em geral, o método `paint` do *applet* é chamado pelo contêiner de *applets*. E se você quisesse alterar a aparência do *applet* em resposta às interações do usuário com o *applet*? Em tais situações, você pode querer chamar `paint` diretamente. Mas para chamar `paint`, devemos passar para ele o parâmetro `Graphics` que ele espera. Essa exigência representa um problema para nós. Não temos um objeto `Graphics` à nossa disposição para passar para `paint` (discutiremos essa questão no Capítulo 18). Por essa razão, a classe `JApplet` fornece o método `repaint`. A instrução

```
repaint();
```

obtém o objeto `Graphics` e invoca um outro método, chamado `update`. O método `update` invoca o método `paint` e passa para ele o objeto `Graphics`. O método `repaint` é discutido em detalhes no Capítulo 18.

6.17 (Estudo de caso opcional) Pensando em objetos: identificando operações de classes

Nas seções “Pensando em objetos” nos finais dos Capítulos 3, 4 e 5, executamos as etapas iniciais de um projeto orientado a objetos para nosso simulador de elevador. No Capítulo 3, identificamos as classes que precisamos implementar. No Capítulo 4, criamos um diagrama de classes que modela a estrutura do nosso sistema. No Capítulo 5, examinamos os estados de objetos e modelamos as atividades e as transições de estados dos objetos.

* N. de R.T.: significa desenhar novamente na tela a janela o *applet* com todo os seus componentes com valores atualizados.

Nesta seção, concentramo-nos na determinação das *operações* (ou dos *comportamentos*) das classes necessárias para implementar o simulador de elevador. No Capítulo 7, concentrar-nos-emos nas colaborações (interações) entre os objetos de nossas classes.

Uma operação de uma classe é um serviço que a classe presta aos “clientes” (usuários) daquela classe. Consideremos as operações de algumas classes do mundo real. Entre as operações de um rádio incluem-se ajustar sua estação e volume (normalmente invocadas por um ouvinte que está ajustando os controles do rádio). As operações de um carro incluem acelerar (invocada apertando-se o pedal do acelerador), desacelerar (invocada pressionando-se o pedal do freio e/ou soltando-se o pedal do acelerador), girar o volante e trocar de marchas.

Podemos derivar muitas das operações de cada classe diretamente da definição do problema. Para fazer isso, examinamos os verbos e as frases com verbos da definição do problema. Relacionamos, então, cada uma destas a classes particulares em nosso sistema (Fig. 6.19). Muitas das frases com verbos na Fig. 6.19 nos ajudam a determinar as operações de nossas classes.

Classe	Frases com verbos
Elevator	move-se para outro andar, chega em um andar, desliga o botão do elevador, soa a campainha do elevador, sinaliza sua chegada, abre a porta, fecha a porta
ElevatorShaft	desliga a luz, liga a luz, desliga botão de andar
Person	caminha em um andar, aperta o botão do andar, aperta o botão do elevador, anda no elevador, entra no elevador, sai do elevador
Floor	[nenhuma na definição do problema]
FloorButton	chama o elevador
ElevatorButton	fecha porta do elevador, sinaliza elevador para se mover até o outro andar
FloorDoor	avisa pessoa para entrar no elevador (abindo)
ElevatorDoor	avisa pessoa para sair do elevador (abindo), abre a porta do andar, fecha a porta do andar
Bell	[nenhuma na definição do problema]
Light	[nenhuma na definição do problema]
ElevatorModel	cria pessoa

Fig. 6.19 Frases com verbos para cada classe no simulador.

Para criar operações, examinamos as frases com verbos listadas com cada classe. A frase “move-se para outro andar”, listada com a classe **Elevator**, refere-se à atividade na qual o elevador se move entre andares. “Move-se” deve ser uma operação da classe **Elevator**? O elevador decide se mover, em resposta ao pressionamento de um botão. O botão sinaliza para o elevador se mover, mas ele não move efetivamente o elevador – portanto, “move-se para o outro andar” não corresponde a uma operação (incluímos as operações para informar ao elevador que deve se mover para o outro andar mais tarde na discussão, quando discutimos as frases com verbos associadas com os botões). A frase “chega em um andar” também não é uma operação, porque o elevador decide sozinho quando chegou em um andar, depois de cinco segundos de viagem.

A frase “desliga o botão do elevador” associada com a classe **Elevator** implica que o elevador informe ao botão do elevador que deve se desligar. Portanto, a classe **ElevatorButton** precisa de uma operação para oferecer este serviço para o elevador. Colocamos esta operação (**resetButton**) no compartimento inferior da classe **ElevatorButton** em nosso diagrama de classes (Fig. 6.20).

Representamos os nomes das operações como nomes de métodos (colocando um par de parênteses após os nomes) e incluímos as informações sobre o tipo de valor devolvido depois do dois-pontos:

```
resetButton() : void
```

Os parênteses podem conter uma lista separada por vírgulas dos parâmetros que a operação recebe – neste caso, nenhum. Por enquanto, a maioria das nossas operações não tem parâmetros e tem um tipo devolvido **void**; isto pode mudar à medida que nossos processos de projeto e implementação continuarem.

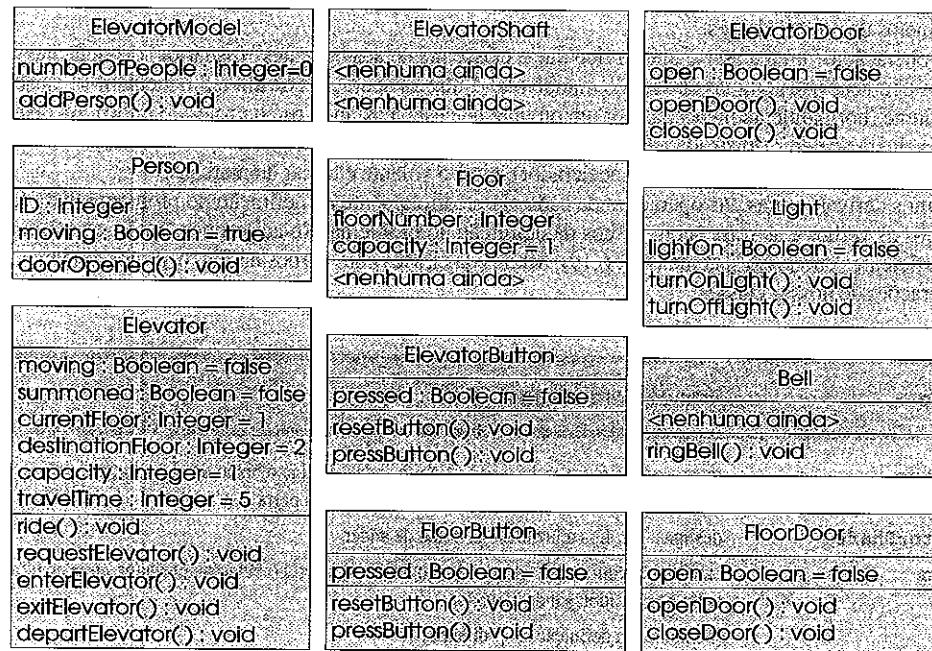


Fig. 6.20 Classe com atributos e operações.

Da frase “soa a campainha do elevador” listada com a classe **Elevator**, concluímos que a classe **Bell** deve ter uma operação que presta um serviço – a saber, soar. Listamos a operação **ringBell** sob a classe **Bell**.

Quando o elevador chega em um andar, ele “avisa sua chegada” para as portas. A porta do elevador responde se abrindo, como implícito na frase “abre a porta [do elevador]” associada com a classe **Elevator**. Portanto, a classe **ElevatorDoor** precisa de uma operação que abra a porta. Colocamos a operação **openDoor** no compartimento inferior desta classe. A frase “fecha a porta [do elevador]” indica que a classe **ElevatorDoor** precisa de uma operação que feche a porta, de modo que colocamos a operação **closeDoor** no mesmo compartimento.

A classe **ElevatorShaft** lista “desliga a luz” e “liga a luz” em sua coluna de frases com verbos, de modo que criamos as operações **turnOffLight** e **turnOnLight** e as listamos sob a classe **Light**. A frase “desliga botão do andar” implica que o elevador instrua um botão de andar a se desligar. Portanto, a classe **FloorButton** precisa de uma operação **resetButton**.

A frase “caminha em um andar”, listada pela classe **Person**, não é uma operação, porque uma pessoa decide caminhar através do andar em resposta à criação daquela pessoa. Entretanto, as frases “aperta botão do andar” e “aperta botão do elevador” são operações que pertencem às classes de botão. Portanto, colocamos a operação **pressButton** sob as classes FB e EB em nosso diagrama de classes (Fig. 6.20). A frase “anda no elevador” implica que o elevador precisa de um método que permita a uma pessoa andar no elevador, de modo que colocamos a operação **ride** no compartimento inferior de E. As frases “entra no elevador” e “sai do elevador”, listadas com a classe **Person**, sugerem que a classe **Elevator** precisa operações que correspondam a estas ações.¹ Colocamos as operações **enterElevator** e **exitElevator** no compartimento inferior da classe **Elevator**.

¹ Neste ponto, só podemos adivinhar o que estas operações fazem. Por exemplo, talvez estas operações modelem elevadores do mundo real, e alguns deles têm sensores que detectam quando os passageiros entram e saem. Por enquanto, simplesmente listamos estas operações. Descobriremos quais ações, se for o caso, esta classe executa, à medida que continuarmos com nosso processo de projeto.

A frase “chama o elevador”, listada sob a classe `FloorButton`, implica que a classe `Elevator` precisa de uma operação `requestElevator`. A frase “sinaliza elevador para se mover até o outro andar”, listada com a classe `ElevatorButton`, implica que `ElevatorButton` informe ao elevador de que deve partir. Portanto, o `Elevator` precisa oferecer um serviço de “partida”; colocamos uma operação `departElevator` no compartimento inferior da classe `Elevator`.

As frases listadas com a classe `FloorDoor` e `ElevatorDoor` mencionam que as portas – ao abrir – sinalizam a um objeto `Person` para entrar no elevador ou sair dele. Especificamente, a porta informa a uma pessoa que a porta está aberta (a pessoa então entra no elevador ou sai dele, conforme o caso). Colocamos a operação `doorOpened` no compartimento inferior para a classe `Person`. Além disso, a `ElevatorDoor` abre e fecha a `FloorDoor`, de modo que atribuímos `openDoor` e `closeDoor` ao compartimento inferior da classe `FloorDoor`.

Por último, a ação “cria pessoa” associada à classe `ElevatorModel` se refere a criar um objeto `Person` e adicioná-lo à simulação. Embora possamos exigir que `ElevatorModel` envie uma mensagem “criar pessoa” e uma “adicionar pessoa”, um objeto da classe `Person` não pode responder a estas mensagens, porque aquele objeto ainda não existe. Discutimos novos objetos quando considerarmos a implementação no Capítulo 8. Colocamos a operação `addPerson` no compartimento inferior de `ElevatorModel` no diagrama de classes da Fig. 6.20 e prevemos que o usuário do aplicativo vai invocar esta operação.

Por enquanto, não nos preocupamos com os parâmetros das operações ou dos tipos de valores devolvidos; só estamos tentando obter um entendimento básico das operações de cada classe. À medida que continuarmos em nosso processo de projeto, o número de operações pertencentes a cada classe pode variar – podemos descobrir que novas operações são necessárias ou que algumas das operações atuais são desnecessárias – e podemos determinar que algumas de nossas operações de classes precisam de tipos de valores devolvidos diferentes de `void`.

Resumo

- A melhor maneira de desenvolver e manter um programa grande é dividi-lo em vários módulos menores. Os módulos são escritos em Java como classes e métodos.
- O método é invocado por uma chamada de método. A chamada de método menciona o método pelo nome e fornece argumentos entre parênteses de que o método chamado precisa para realizar sua tarefa. Se o método estiver em uma outra classe, a chamada precisa ser precedida por um nome de referência e um operador ponto. Se o método for `static`, ele deve ser precedido por um nome de classe e um operador ponto.
- Cada argumento de um método pode ser uma constante, uma variável ou uma expressão.
- Uma variável local é conhecida somente em uma definição de método. Os métodos não têm permissão para conhecer os detalhes de implementação de qualquer outro método (incluindo suas variáveis locais).
- A área de exibição na tela para um `JApplet` tem um painel de conteúdo ao qual os componentes GUI devem ser anexados a fim de poderem ser exibidos durante a execução. O painel de conteúdo é um objeto de classe `Container` do pacote `java.awt`.
- O método `getContentPane` retorna uma referência ao painel de conteúdo do *applet*.
- O formato geral para uma definição de método é

```
tipo do valor de retorno nome do método (lista de parâmetros)
{
    declarações e instruções
}
```

O *tipo do valor de retorno* declara o tipo do valor de retorno para o método chamado. Se um método não devolve um valor, o *tipo do valor de retorno* é `void`. O *nome do método* é qualquer identificador válido. A *lista de parâmetros* é uma lista separada por vírgulas que contém as declarações das variáveis que serão passadas para o método. Se um método não recebe nenhum valor, a *lista de parâmetros* é vazia. O corpo do método é o conjunto das *declarações e instruções* que constituem o método.

- Os argumentos passados para um método devem corresponder em número, tipo e ordem aos parâmetros na definição do método.
- Quando um programa encontra um método, o controle é transferido do ponto de invocação para o método chamado, o método é executado e devolve o controle para o chamador.
- Um método chamado pode devolver o controle para o chamador de três maneiras. Se o método não retorna um valor, o controle é devolvido quando a chave direita de fechamento do método é alcançada ou executando-se a instrução

```
return;
```

Se o método retorna um valor, a instrução

```
return expressão;
```

retorna o valor de expressão.

- Há três maneiras de chamar um método – o nome do método sozinho, uma referência a um objeto seguido pelo operador ponto (.) e o nome do método e um nome de classe seguido pelo operador ponto (.) seguido por um nome de método. A última sintaxe é só para métodos **static** de uma classe.
- Um recurso importante das definições de método é a *coerção de argumentos*. Em muitos casos, valores de argumentos que não correspondem precisamente ao tipo de parâmetro na definição do método são convertidos no tipo adequado antes de o método ser chamado. Em alguns casos, essas conversões podem levar a erros de compilação se as regras de promoção de Java não forem seguidas.
- As regras de promoção especificam a maneira como os tipos podem ser convertidos em outros tipos sem perder dados. As regras de promoção se aplicam a expressões do tipo misto. O tipo de cada valor em uma expressão do tipo misto é promovido para o tipo “mais alto” na expressão.
- O método **Math.random** gera um valor **double** de 0.0 até (mas não incluindo) 1.0. Os valores produzidos por **Math.random** podem ser escalonados e deslocados para produzir valores em um intervalo.
- A equação geral para escalar e deslocar um número aleatório é

```
n = a + (int) ( Math.random() * b );
```

onde **a** é o valor de deslocamento (o primeiro número no intervalo desejado de inteiros consecutivos) e **b** é o fator de escala (a largura do intervalo desejado de inteiros consecutivos).

- A classe pode herdar atributos e comportamentos (dados e métodos) existentes de outra classe especificada à direita da palavra-chave **extends** na definição de classe. Além disso, uma classe pode implementar uma ou mais *interfaces*. A interface especifica um ou mais comportamentos (isto é, métodos) que você deve definir na sua definição de classe.
- A interface **ActionListener** especifica que essa classe deve definir um método com a primeira linha

```
public void actionPerformed( ActionEvent actionEvent )
```

- A tarefa do método **actionPerformed** é processar uma interação do usuário com um componente GUI que gera um evento de ação. Esse método é chamado em resposta à interação do usuário (o evento). Esse processo é chamado de *tratamento de evento*. O tratador de evento é o método **actionPerformed**, que é chamado em resposta ao evento. Esse estilo de programação é conhecido como *programação baseada em eventos*.
- A palavra-chave **final** é utilizada para declarar constantes. As constantes devem ser inicializadas antes de serem utilizadas em um programa. As constantes são chamadas de *constantes identificadas* ou de *variáveis apenas de leitura*.
- O **JLabel** contém um *string* de caracteres a ser exibido na tela. Normalmente, o **JLabel** indica o propósito de um outro elemento GUI na tela.
- **JTextFields** obtêm informações do usuário ou exibem informações na tela.
- Quando o usuário pressiona um **JButton**, normalmente o programa responde executando uma tarefa.
- O método **setLayout** da classe **Container** define o gerenciador de layout para a interface com o usuário do *applet*. Os gerenciadores de layout são fornecidos para organizar componentes GUI em um **Container** para fins de apresentação.
- **FlowLayout** é o gerenciador de layout mais simples. Os componentes GUI são colocados em um **Container** da esquerda para a direita, na ordem em que são anexados ao **Container** com o método **add**. Quando a borda do container é alcançada, os componentes continuam na próxima linha.
- Antes de que qualquer evento possa ser processado, cada componente GUI deve conhecer qual o objeto no programa que define o método de tratamento de evento que será chamado quando ocorrer um evento. Utiliza-se o método **addActionListener** para dizer a um **JButton** ou a um **JTextField** que outro objeto está ouvindo eventos de ação e define o método **actionPerformed**. Isso se chama *registrar o tratador de evento com o componente GUI*. Para responder a um evento de ação, devemos definir uma classe que implementa **ActionListener** e define o método **actionPerformed**. Além disso, devemos registrar o tratador de evento com o componente GUI.
- O método **showStatus** exibe um **String** na barra de estado do container de *applets*.
- Cada identificador de variável tem os atributos duração (tempo de vida) e escopo. A duração de um identificador determina quando esse identificador existe na memória. O escopo de um identificador é onde o identificador pode ser usado em um programa.

- Os identificadores que representam as variáveis locais em um método têm duração automática. As variáveis de duração automática são criadas quando o controle de um programa alcança sua declaração, elas existem enquanto o bloco em que são declaradas está ativo e são destruídas quando o bloco em que são declaradas termina.
- Java também tem identificadores de duração estática. As variáveis e referências de duração estática existem a partir do ponto em que a classe que as define é carregada na memória para execução até o término do programa.
- Os escopos para um identificador são escopo de classe e escopo de bloco. Uma variável de instância declarada fora de qualquer método tem escopo de classe. Tal identificador “é conhecido” em todos os métodos da classe. Os identificadores declarados dentro de um bloco têm escopo de bloco. Os escopos de bloco terminam na chave direita de fechamento (}) do bloco.
- As variáveis locais declaradas no início de um método têm escopo de bloco, da mesma forma que os parâmetros do método, que são considerados variáveis locais do método.
- Qualquer bloco pode conter declarações de variáveis.
- Um método recursivo é um método que chama a si próprio direta ou indiretamente.
- Se um método recursivo é chamado com um caso básico, o método devolve um resultado. Se o método é chamado com um problema mais complexo, o método divide o problema em dois ou mais pedaços conceituais: um pedaço que o método sabe como resolver e uma versão um pouco menor do problema original. Como esse novo problema se parece com o problema original, o método dispara uma chamada recursiva para trabalhar no problema menor.
- Para a recursão terminar, a sequência de problemas cada vez menores deve convergir para o caso básico. Quando o método reconhece o caso básico, o resultado é retornado para a chamada de método anterior e uma sequência de retornos segue para cima até que a chamada original do método retorna o resultado final.
- A iteração e a recursão baseiam-se em uma estrutura de controle: a iteração utiliza uma estrutura de repetição; a recursão utiliza uma estrutura de seleção.
- A iteração e a recursão envolvem repetição: a iteração utiliza explicitamente uma estrutura de repetição; a recursão alcança a repetição por chamadas repetidas do método.
- A iteração e a recursão envolvem um teste de terminação: a iteração termina quando a condição de continuação do laço falha; a recursão termina quando um caso básico é reconhecido.
- A iteração e a recursão podem ocorrer infinitamente: um laço infinito ocorre com iteração se o teste de continuação do laço nunca se tornar falso; a recursão infinita ocorre se o passo de recursão não reduzir o problema de maneira que converja para o caso básico.
- A recursão repetidamente invoca o mecanismo e, consequentemente, a sobrecarga das chamadas de método. Essa repetição pode ser cara, tanto em tempo de processador como em espaço de memória.
- O usuário pressiona a tecla *Enter* ao digitar em um `JTextField` para gerar o evento de ação. O tratamento de evento para esse componente GUI é configurado como para um `JButton`: Deve ser definida uma classe que implementa `ActionListener` e define o método `actionPerformed`. Além disso, o método `addActionListener` do `JTextField` deve ser chamado para registrar o evento.
- É possível definir métodos com o mesmo nome mas com listas diferentes de parâmetros. Esse recurso se chama sobre-carga de método. Quando se chama um método sobrecarregado, o compilador seleciona o método adequado examinando-se os argumentos na chamada.
- Os métodos sobrecarregados podem retornar valores diferentes e devem ter listas diferentes de parâmetros. Dois métodos que sejam diferentes somente pelo tipo do valor retornado resultarão em um erro de sintaxe.
- O método `init` do *applet* é chamado uma vez pelo contêiner de *applets* quando um *applet* é carregado para execução. Ele executa a inicialização de um *applet*. O método `start` do *applet* é chamado depois que o método `init` completa sua execução e toda vez que o usuário do navegador retorna para a página HTML em que o *applet* reside (depois de navegar por outra página HTML).
- O método `paint` do *applet* é chamado depois que o método `init` completa sua execução e o método `start` começou a ser executado para desenhar no *applet*. Ele também é chamado automaticamente toda vez que o *applet* precisa ser repintado.
- O método `stop` do *applet* é chamado quando o *applet* deve suspender a execução – normalmente quando o usuário do navegador deixa a página HTML em que o *applet* reside.
- O método `destroy` do *applet* é chamado quando o *applet* está sendo removido da memória – normalmente quando o usuário do navegador sai da sessão de navegação.
- O método `repaint` pode ser chamado em um *applet* para fazer uma nova chamada a `paint`. O método `repaint` invoca outro método denominado `update` e passa para ele o objeto `Graphics`. O método `update` invoca o método `paint` e passa para ele o objeto `Graphics`.

Terminologia

<i>argumento em uma chamada de método</i>	<i>Java API (biblioteca de classes Java)</i>
<i>assinatura</i>	Math.E
<i>bloco</i>	Math.PI
<i>capacidade de reutilização de software</i>	<i>método</i>
<i>caso básico em recursão</i>	<i>método actionPerformed</i>
<i>chamada de método</i>	<i>método chamado</i>
<i>chamada recursiva</i>	<i>método definido pelo programador</i>
<i>chamador</i>	<i>método destroy de JApplet</i>
<i>chamar um método</i>	<i>método fatorial</i>
<i>classe</i>	<i>método init de JApplet</i>
<i>classe ActionEvent</i>	<i>método Math.random</i>
<i>classe FlowLayout</i>	<i>método paint de JApplet</i>
<i>classe JButton do pacote javax.swing</i>	<i>método que chama</i>
<i>classe JLabel do pacote javax.swing</i>	<i>método recursivo</i>
<i>classe JTextField do pacote javax.swing</i>	<i>método repaint de JApplet</i>
<i>coerção de argumentos</i>	<i>método setLayout de JApplet</i>
<i>constante</i>	<i>método showStatus de JApplet</i>
<i>constante identificada</i>	<i>método start de JApplet</i>
<i>cópia de um valor</i>	<i>método stop de JApplet</i>
<i>declaração de método</i>	<i>método update de JApplet</i>
<i>definição de método</i>	<i>métodos da classe Math</i>
<i>deslocamento</i>	<i>operador de chamada de método, ()</i>
<i>dividir para conquistar</i>	<i>parâmetro de referência</i>
<i>duração</i>	<i>parâmetro em uma definição de método</i>
<i>duração automática</i>	<i>programa modular</i>
<i>duração de memória estática</i>	<i>recursão</i>
<i>elemento de sorte</i>	<i>regras de promoção</i>
<i>engenharia de software</i>	<i>retorno</i>
<i>escala</i>	<i>simulação</i>
<i>escopo</i>	<i>sobrecarregar</i>
<i>escopo de bloco</i>	<i>tipo de referência</i>
<i>escopo de classe</i>	<i>tipo de valor retornado</i>
<i>etapa de recursão</i>	<i>variável automática</i>
<i>expressão de tipo misto</i>	<i>variável de somente leitura</i>
<i>final</i>	<i>variável local</i>
<i>geração de números aleatórios</i>	void
<i>interface ActionListener</i>	
<i>invocar um método</i>	
<i>iteração</i>	

Exercícios de auto-revisão

- 6.1 Preencha as lacunas em cada uma das afirmações a seguir:
- Os módulos de programa em Java são chamados de _____ e _____.
 - Um método é invocado com uma _____.
 - Uma variável conhecida somente dentro do método em que ela é definida é chamada de _____.
 - A instrução em um método chamado _____ pode ser utilizada para passar o valor de uma expressão de volta para o método de chamada.
 - A palavra-chave _____ é utilizada no cabeçalho de um método para indicar que o método não retorna um valor.
 - O _____ de um identificador é a parte do programa em que o identificador pode ser utilizado.
 - As três maneiras de retornar controle a partir de um método chamado para um chamador são _____, _____ e _____.
 - O método _____ é invocado uma vez quando um *applet* inicia a execução.
 - O método _____ produz números aleatórios.
 - O método _____ é invocado toda vez que o usuário de um navegador revisita a página HTML em que um *applet* reside.

- k) O método _____ é invocado para desenhar em um *applet*.
- l) As variáveis declaradas em um bloco ou em uma lista de parâmetros do método são de duração _____.
- m) O método _____ invoca o método **update** do *applet*, que, por sua vez, invoca o método **paint** do *applet*.
- n) O método _____ é invocado para um *applet* toda vez que o usuário de um navegador deixa a página HTML em que o *applet* reside.
- o) O método que chama a si próprio, seja direta ou indiretamente, é um método _____.
- p) O método recursivo em geral tem dois componentes: um que fornece uma maneira de a recursão terminar testando quanto a um caso _____ e um que expressa o problema como uma chamada recursiva para um problema um pouco mais simples que a chamada original.
- q) Em Java, é possível ter vários métodos com o mesmo nome que operam, cada um deles, sobre diferentes tipos e/ou números de argumentos. Chama-se método de _____.
- r) O qualificador _____ é utilizado para declarar variáveis de somente leitura.

- 6.2** Para o seguinte programa, declare o escopo (de classe ou de bloco) de cada um dos seguintes elementos:
- a) A variável **x**.
 - b) A variável **y**.
 - c) O método **cube**.
 - d) O método **paint**.
 - e) A variável **yPos**.

```

1  public class CubeTest extends JApplet {
2      int x;
3
4      public void paint( Graphics g )
5      {
6          int yPos = 25;
7
8          for ( x = 1; x <= 10; x++ ) {
9              g.drawString( cube( x ), 25, yPos );
10             yPos += 15;
11         }
12     }
13
14     public int cube( int y )
15     {
16         return y * y * y;
17     }
18 }
```

- 6.3** Escreva um aplicativo que testa se os exemplos de chamadas de métodos da biblioteca de matemática mostrado na Fig. 6.2 realmente produzem os resultados indicados.

- 6.4** Escreva o cabeçalho do método para cada um dos seguintes métodos:
- a) O método **hypotenuse**, que aceita dois argumentos em ponto flutuante de precisão dupla **side1** e **side2** e retorna um resultado em ponto flutuante com precisão dupla.
 - b) O método **smallest**, que recebe três inteiros **x**, **y**, **z** e retorna um inteiro.
 - c) O método **instructions**, que não recebe nenhum argumento e não retorna um valor. [Nota: esses métodos são comumente utilizados para exibição de instruções para o usuário.]
 - d) O método **intToFloat**, que aceita um argumento inteiro, **number**, e retorna um resultado de ponto flutuante.

- 6.5** Localize o erro em cada um dos seguintes segmentos de programa e explique como o erro pode ser corrigido:

```

a) int g() {
    System.out.println( "Inside method g" );
    int h() {
        System.out.println( "Inside method h" );
    }
}
```

```

b) int sum( int x, int y ) {
    int result;
    result = x + y;
}

c) int sum( int n ) {
    if ( n == 0 )
        return 0;
    else
        n + sum( n - 1 );
}

d) void f( float a ); {
    float a;
    System.out.println( a );
}

e) void product() {
    int a = 6, b = 5, c = 4, result;
    result = a * b * c;
    System.out.println( "Result is " + result );
    return result;
}

```

6.6 Escreva um *applet* Java completo para solicitar ao usuário o raio de uma esfera (do tipo **double**) e chamar o método **sphereVolume** para que calcule e exiba o volume dessa esfera com a atribuição

```
volume = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radius, 3 )
```

O usuário deve digitar o raio com um **JTextField**.

Respostas aos exercícios de auto-revisão

6.1 a) métodos e classes. b) chamada de método. c) variável local. d) **return**. e) **void**. f) escopo. g) **return**; ou **return expressão**; ou encontrar a chave direita de fechamento de um método. h) **init**. i) **Math.random**. j) **start**. k) **paint**. l) automática. m) **repaint**. n) **stop**. o) recursivo. p) básico. q) sobrecarga. r) **final**.

6.2 a) Escopo de classe. b) Escopo de bloco. c) Escopo de classe. d) Escopo de classe. e) Escopo de bloco.

6.3 A seguinte solução demonstra os métodos da classe **Math** na Fig. 6.2.

```

1 // Exercício 6.3: Mathtest.java
2 // Testando os métodos da classe Math
3
4 public class MathTest {
5     public static void main( String args[] )
6     {
7         System.out.println( "Math.abs( 23.7 ) = " +
8                         Math.abs( 23.7 ) );
9         System.out.println( "Math.abs( 0.0 ) = " +
10                        Math.abs( 0.0 ) );
11         System.out.println( "Math.abs( -23.7 ) = " +
12                        Math.abs( -23.7 ) );
13         System.out.println( "Math.ceil( 9.2 ) = " +
14                        Math.ceil( 9.2 ) );
15         System.out.println( "Math.ceil( -9.8 ) = " +
16                        Math.ceil( -9.8 ) );
17         System.out.println( "Math.cos( 0.0 ) = " +
18                        Math.cos( 0.0 ) );
19         System.out.println( "Math.exp( 1.0 ) = " +
20                        Math.exp( 1.0 ) );

```

```

21     System.out.println( "Math.exp( 2.0 ) = " +
22                     Math.exp( 2.0 ) );
23     System.out.println( "Math.floor( 9.2 ) = " +
24                     Math.floor( 9.2 ) );
25     System.out.println( "Math.floor( -9.8 ) = " +
26                     Math.floor( -9.8 ) );
27     System.out.println( "Math.log( 2.718282 ) = " +
28                     Math.log( 2.718282 ) );
29     System.out.println( "Math.log( 7.389056 ) = " +
30                     Math.log( 7.389056 ) );
31     System.out.println( "Math.max( 2.3, 12.7 ) = " +
32                     Math.max( 2.3, 12.7 ) );
33     System.out.println( "Math.max( -2.3, -12.7 ) = " +
34                     Math.max( -2.3, -12.7 ) );
35     System.out.println( "Math.min( 2.3, 12.7 ) = " +
36                     Math.min( 2.3, 12.7 ) );
37     System.out.println( "Math.min( -2.3, -12.7 ) = " +
38                     Math.min( -2.3, -12.7 ) );
39     System.out.println( "Math.pow( 2, 7 ) = " +
40                     Math.pow( 2, 7 ) );
41     System.out.println( "Math.pow( 9, .5 ) = " +
42                     Math.pow( 9, .5 ) );
43     System.out.println( "Math.sin( 0.0 ) = " +
44                     Math.sin( 0.0 ) );
45     System.out.println( "Math.sqrt( 25.0 ) = " +
46                     Math.sqrt( 25.0 ) );
47     System.out.println( "Math.tan( 0.0 ) = " +
48                     Math.tan( 0.0 ) );
49   }
50 }
```

```

Math.abs( 23.7 ) = 23.7
Math.abs( 0.0 ) = 0
Math.abs( -23.7 ) = 23.7
Math.ceil( 9.2 ) = 10
Math.ceil( -9.8 ) = -9
Math.cos( 0.0 ) = 1
Math.exp( 1.0 ) = 2.71828
Math.exp( 2.0 ) = 7.38906
Math.floor( 9.2 ) = 9
Math.floor( -9.8 ) = -10
Math.log( 2.718282 ) = 1
Math.log( 7.389056 ) = 2
Math.max( 2.3, 12.7 ) = 12.7
Math.max( -2.3, -12.7 ) = -2.3
Math.min( 2.3, 12.7 ) = 2.3
Math.min( -2.3, -12.7 ) = -12.7
Math.pow( 2, 7 ) = 128
Math.pow( 9, .5 ) = 3
Math.sin( 0.0 ) = 0
Math.sqrt( 25.0 ) = 5
Math.tan( 0.0 ) = 0
```

- 6.4 a) double hypotenuse(double side1, double side2)
 b) int smallest(int x, int y, int z)
 c) void instructions()
 d) float intToFloat(int number)
- 6.5 a) Erro: o método h está definido no método g.
 Correção: mover a definição de h para fora da definição de g.

- b) Erro: o método supostamente deve retornar um inteiro, mas não o faz.

Correção: excluir a variável `result` e colocar a instrução:

```
return x + y;
```

no método, ou adicionar a seguinte instrução ao fim do corpo do método:

```
return result;
```

- c) Erro: o resultado de `n + sum(n - 1)` não é retornado por esse método recursivo, o que resulta em um erro de sintaxe.

Correção: reescrever a instrução na cláusula `else` como

```
return n + sum(n - 1);
```

- d) Erro: o ponto-e-vírgula após o parêntese direito que inclui a lista de parâmetros e a redefinição do parâmetro `a` na definição de método estão incorretos.

Correção: excluir o ponto-e-vírgula após o parêntese direito da lista de parâmetros e excluir a declaração `float a;`

- e) Erro: o método retorna um valor quando supostamente não deveria.

Correção: alterar o tipo de retorno para `int`.

6.6 A seguinte solução calcula o volume de uma esfera com o raio inserido pelo usuário.

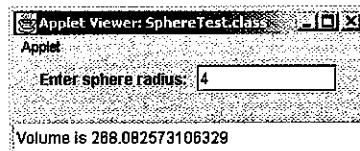
```

1 // Exercício 6.6: SphereTest.java
2
3 // Pacotes do núcleo de Java
4 import java.awt.*;
5 import java.awt.event.*;
6
7 // Pacotes do núcleo de Java
8 import javax.swing.*;
9
10 public class SphereTest extends JApplet
11     implements ActionListener {
12
13     JLabel promptLabel;
14     JTextField inputField;
15
16     public void init()
17     {
18         Container container = getContentPane();
19         container.setLayout( new FlowLayout() );
20
21         promptLabel = new JLabel( "Enter sphere radius: " );
22         inputField = new JTextField( 10 );
23         inputField.addActionListener( this );
24         container.add( promptLabel );
25         container.add( inputField );
26     }
27
28     public void actionPerformed( ActionEvent actionEvent )
29     {
30         double radius =
31             Double.parseDouble( actionEvent.getActionCommand() );
32
33         showStatus( "Volume is " + sphereVolume( radius ) );
34     }
35
36     public double sphereVolume( double radius )
37     {
38         double volume =
39             ( 4.0 / 3.0 ) * Math.PI * Math.pow( radius, 3 );

```

```

40
41     return volume;
42 }
43 }
```



Exercícios

6.7 Qual é o valor de **x** depois que cada uma das seguintes instruções é executada?

- a) **x** = **Math.abs(7.5);**
- b) **x** = **Math.floor(7.5);**
- c) **x** = **Math.abs(0.0);**
- d) **x** = **Math.ceil(0.0);**
- e) **x** = **Math.abs(-6.4);**
- f) **x** = **Math.ceil(-6.4);**
- g) **x** = **Math.ceil(-Math.abs(-8 + Math.floor(-5.5)));**

6.8 Um estacionamento cobra uma taxa mínima de R\$ 2,00 para estacionar por até três horas. O estacionamento cobra um adicional de R\$ 0,50 por hora para cada hora ou *fração de hora* após as três primeiras horas. A taxa máxima para qualquer período de 24 horas é R\$ 10,00. Suponha que nenhum carro fica estacionado por mais de 24 horas. Escreva um *applet* que calcula e exibe as taxas de estacionamento para cada cliente que estacionou o carro nessa garagem ontem. Você deve digitar em um **JTextField** as horas de estacionamento para cada cliente. O programa deve exibir o valor a cobrar para o cliente atual e deve calcular e exibir o total acumulado dos recibos de ontem. O programa deve utilizar o método **calculateCharges** para determinar o valor a cobrar de cada cliente.

6.9 Uma aplicação do método **Math.floor** é arredondar um valor para o inteiro mais próximo. A instrução

```
y = Math.floor( x + .5 );
```

arredondará o número **x** para o inteiro mais próximo e atribuirá o resultado a **y**. Escreva um *applet* que lê valores **double** e utiliza a instrução precedente para arredondar cada um desses números para o inteiro mais próximo. Para cada número processado, exiba ambos os números, o original e o arredondado

6.10 Pode-se usar **Math.floor** para arredondar um número para um número específico de casas decimais. A instrução

```
y = Math.floor( x * 10 + .5 ) / 10;
```

arredonda **x** para a casa decimal de décimos (a primeira posição à direita do ponto de fração decimal). A instrução

```
y = Math.floor( x * 100 + .5 ) / 100;
```

arredonda **x** para a casa decimal de centésimos (isto é, a segunda posição à direita da casa decimal). Escreva um *applet* que define quatro métodos para arredondar um número **x** de várias maneiras:

- a) **roundToInteger(number)**
- b) **roundToTenths(number)**
- c) **roundToHundredths(number)**
- d) **roundToThousandths(number)**

Para cada valor lido, seu programa deve exibir o valor original, o número arredondado para o inteiro mais próximo, o número arredondado para o décimo mais próximo, o número arredondado para o centésimo mais próximo e o número arredondado para o milésimo mais próximo.

6.11 Responda cada uma das seguintes perguntas.

- a) O que significa escolher números “aleatoriamente”?
- b) Por que o método **Math.random** é útil para simular jogos de azar?
- c) Por que é freqüentemente necessário escalar e/ou deslocar os valores produzidos por **Math.random**?

d) Por que a simulação computadorizada de situações do mundo real é uma técnica útil?

6.12 Escreva instruções que atribuem inteiros aleatórios à variável *n* nos seguintes intervalos:

- a) $1 \leq n \leq 2$
- b) $1 \leq n \leq 100$
- c) $0 \leq n \leq 9$
- d) $1000 \leq n \leq 1112$
- e) $-1 \leq n \leq 1$
- f) $-3 \leq n \leq 11$

6.13 Para cada um dos seguintes conjuntos de inteiros, escreva uma única instrução que imprimirá aleatoriamente um número do conjunto.

- a) 2, 4, 6, 8, 10.
- b) 3, 5, 7, 9, 11.
- c) 6, 10, 14, 18, 22.

6.14 Escreva um método `integerPower(base, exponent)` que retorna o valor de

$$\text{base}^{\text{exponente}}$$

Por exemplo, `integerPower(3, 4)` calcula 3^4 (ou $3 * 3 * 3 * 3$). Suponha que `exponent` é um inteiro diferente de zero positivo e `base` é um inteiro. O método `integerPower` deve utilizar `for` ou `while` para controlar o cálculo. Não utilize nenhum método da biblioteca de matemática. Incorpore esse método em um *applet* que lê valores inteiros digitados pelo usuário em `JtextFields` para a `base` e o `exponent` e realiza o cálculo com o método `integerPower`. [Nota: registre para tratamento de eventos somente no segundo `JtextField`. O usuário deve interagir com o programa digitando números em ambos os `JtextFields` e pressionando *Enter* somente no segundo `JtextField`.]

6.15 Defina um método `hypotenuse` que calcula o comprimento da hipotenusa de um triângulo retângulo quando os outros dois lados são fornecidos (os exemplos aparecem na Fig. 6.21). O método deve receber dois argumentos do tipo `double` e retornar a hipotenusa como um `double`. Incorpore esse método em um *applet* que lê valores inteiros para `side1` e `side2` a partir de `JtextFields` e realiza o cálculo com o método `hypotenuse`. Determine o comprimento da hipotenusa para cada um dos seguintes triângulos. [Nota: registre para o tratamento de eventos somente no segundo `JtextField`. O usuário deve interagir com o programa digitando números em ambos os `JtextFields` e pressionando *Enter* somente no segundo `JtextField`.]

Triângulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

Fig. 6.21 Valores para os lados dos triângulos no Exercício 6.15.

6.16 Escreva um método `multiple` que determina para um par de inteiros se o segundo inteiro é um múltiplo do primeiro. O método deve receber dois argumentos inteiros e devolver `true` se o segundo for um múltiplo do primeiro e `false` caso contrário. Incorpore esse método a um *applet* que lê uma série de pares de inteiros (um par por vez utilizando `JTextFields`). [Nota: registre para o tratamento de eventos só o segundo `JTextField`. O usuário deve interagir com o programa digitando números em ambos os `JTextFields` e pressionando *Enter* só no segundo `JTextField`.]

6.17 Escreva um método `isEven` que utiliza o operador de módulo para determinar se um inteiro é par. O método deve receber um argumento inteiro e retornar `true` se o inteiro for par e `false` caso contrário. Incorpore esse método a um *applet* que lê uma série de inteiros (um por vez, utilizando um `JTextField`).

6.18 Escreva um método `squareOfAsterisks` que exibe um quadrado sólido de asteriscos cujo lado é especificado no parâmetro inteiro `side`. Por exemplo, se `side` for 4, o método exibe

```
*****
*****
*****
*****
```

Incorpore esse método em um *applet* que lê um valor de inteiro para **side** digitado pelo usuário no teclado e faz o desenho com o método **squareOfAsterisks**. Observe que esse método deve ser chamado a partir do método **paint** do *applet* e deve receber o objeto **Graphics** de **paint**.

6.19 Modifique o método criado no Exercício 6.18 para formar o quadrado com qualquer caractere que esteja contido no parâmetro de caractere **fillCharacter**. Assim, se **side** for 5 e **fillCharacter** for "#", o método deve imprimir

```
#####
#####
#####
#####
#####
#####
```

6.20 Utilize técnicas semelhantes àquelas desenvolvidas nos Exercícios 6.18 e 6.19 para produzir um programa que representa graficamente uma ampla variedade de formas.

6.21 Modifique o programa do Exercício 6.18 para desenhar um quadrado sólido com o método **fillRect** da classe **Graphics**. O método **fillRect** recebe quatro argumentos: coordenada *x*, coordenada *y*, largura e altura. Permita que o usuário digite as coordenadas em que o quadrado deve aparecer.

6.22 Escreva segmentos de programa que realizem cada uma das tarefas seguintes:

- Calcular a parte inteira do quociente quando o inteiro *a* é dividido pelo inteiro *b*.
- Calcular o resto inteiro quando o inteiro *a* é dividido pelo inteiro *b*.
- Utilizar os pedaços de programa desenvolvidos em a) e b) para escrever um método **displayDigits** que recebe um inteiro entre 1 e 99999 e o imprime como uma série de dígitos, cada um deles separado do outro por dois espaços. Por exemplo, o inteiro 4562 deve ser impresso como

4 5 6 2

- Incorporar o método desenvolvido na parte c) em um *applet* que lê um inteiro a partir de um diálogo de entrada e invoca **displayDigits** passando para o método o inteiro lido. Exiba o resultado em um diálogo de mensagem.

6.23 Implemente os seguintes métodos inteiros:

- O método **celsius** retorna o equivalente em Celsius de uma temperatura em Fahrenheit com o cálculo

$$C = 5.0 / 9.0 * (F - 32);$$

- O método **fahrenheit** retorna o equivalente em Fahrenheit de uma temperatura em Celsius.

$$F = 9.0 / 5.0 * C + 32;$$

- Utilize esses métodos para escrever um *applet* que permita que o usuário digite uma temperatura em Fahrenheit e exiba o equivalente em Celsius ou digite uma temperatura em Celsius e exiba o equivalente em Fahrenheit.

[Nota: esse *applet* exigirá que dois objetos **JTextField** tenham eventos de ação registrados. Quando **actionPerformed** é invocado, o parâmetro **ActionEvent** tem o método **getSource()** para determinar o componente GUI com que o usuário interagiu. Seu método **actionPerformed** deve conter uma estrutura **if/else** na seguinte forma:

```
if (e.getSource() == input1) {
    // interação do processo input1 aqui
}
else { // e.getSource () == input2
    // interação do processo input2 aqui
}
```

onde **input1** e **input2** são referências para os **JTextField**.]

6.24 Escreva um método **minimum3** que retorna o menor de três números em ponto flutuante. Utilize o método **Math.min** para implementar **minimum3**. Incorpore o método em um *applet* que lê três valores do usuário e determina o menor valor. Exiba o resultado na barra de status.

6.25 Dizemos que um número inteiro é um *número perfeito* se a soma de seus fatores, incluindo 1 (mas não o próprio número), é igual ao número. Por exemplo, 6 é um número perfeito porque $6 = 1 + 2 + 3$. Escreva um método **perfect** que determina se seu parâmetro **number** é um número perfeito. Utilize esse método em um *applet* que determina e exibe todos os números perfeitos entre 1 e 1000. Imprima os fatores de cada número perfeito para confirmar que o número é de

fato perfeito. Teste o poder de cálculo de seu computador testando números bem maiores que 1000. Exiba os resultados em uma `JTextArea` que tenha a funcionalidade de rolagem.

6.26 Dizemos que um inteiro é *primo* se ele for divisível somente por 1 e por ele próprio. Por exemplo, 2, 3, 5 e 7 são primos, mas 4, 6, 8 e 9 não o são.

- Escreva um método que determina se um número é primo.
- Utilize esse método em um *applet* que determina e imprime todos os números primos entre 1 e 10.000. Quantos desses 10.000 números você realmente tem de testar antes de se certificar de que encontrou todos os primos? Exiba os resultados em uma `JTextArea` que tenha a funcionalidade de rolagem.
- Inicialmente você poderia pensar que $n/2$ é o limite superior que você deve testar para ver se um número é primo, mas você precisa ir apenas até a raiz quadrada de n . Por quê? Rescreva o programa e execute-o de ambas as maneiras. Avalie o aumento de desempenho.

6.27 Escreva um método que recebe um valor inteiro e retorna o número com seus dígitos invertidos. Por exemplo, dado o número 7631, o método deve retornar 1367. Incorpore o método a um *applet* que lê o valor do usuário. Exiba o resultado do método na barra de estado.

6.28 O *máximo divisor comum (MDC)* de dois inteiros é o maior inteiro que divide cada um dos dois números deixando resto zero. Escreva um método `gcd` que retorna o máximo divisor comum de dois inteiros. Incorpore o método a um *applet* que lê dois valores do usuário. Exiba o resultado do método na barra de estado.

6.29 Escreva um método `qualityPoints` que lê a média de um aluno e retorna 4 se a média do aluno estiver entre 90–100, 3 se a média estiver entre 80–89, 2 se a média estiver entre 70–79, 1 se a média estiver entre 60–69 e 0 se a média estiver abaixo de 60. Incorpore o método a um *applet* que lê o valor de um usuário. Exiba o resultado do método na barra de estado.

6.30 Escreva um *applet* que simula o lançamento de uma moeda. Faça o programa arremessar a moeda toda vez que o usuário pressionar o botão “**Toss**”. Conte o número de vezes que cada lado da moeda aparece. Exiba os resultados. O programa deve chamar um método separado `flip` que não recebe nenhum argumento e retorna `false` para coroa e `true` para cara. [Nota: se o programa simular o arremesso da moeda realisticamente, cada lado da moeda deve aparecer aproximadamente metade das vezes.]

6.31 Os computadores estão desempenhando um papel crescente na educação. Escreva um programa que ajudará um aluno da escola elementar a aprender multiplicação. Utilize `Math.random` para gerar dois inteiros positivos de um algarismo. Então você deve exibir uma pergunta na barra de estado, tal como

How much is 6 times 7? [Quanto é 6 vezes 7?]

O aluno então digita a resposta em um `JTextField`. A seguir, o programa verifica a resposta do aluno. Se ela estiver correta, ele desenha o string “**Very good!**” [“**Muito bem!**”] no *applet*, e faz outra pergunta de multiplicação. Se a resposta estiver errada, ele desenha o string “**No. Please try again.**” [“**Não, tente novamente.**”] no *applet*, e deixa o aluno responder à mesma pergunta novamente repetidas vezes até que o aluno por fim responda corretamente. Deve-se utilizar um método separado para gerar cada nova pergunta. Esse método deve ser chamado uma vez quando o *applet* inicia a execução e toda vez que o usuário responde a pergunta corretamente. Todo desenho no *applet* deve ser realizado pelo método `paint`.

6.32 O uso de computadores na educação é conhecido como *instrução auxiliada por computador (computer-assisted instruction – CAI)*. Um problema que ocorre em ambientes CAI é a fadiga do aluno. Isso pode ser eliminado variando-se o diálogo do computador para prender a atenção do aluno. Modifique o programa do Exercício 6.31 de modo que vários comentários sejam impressos para cada resposta correta ou resposta incorreta, como segue:

Frases para uma resposta correta:

Very good!	[Muito bom!]
Excellent!	[Excelente!]
Nice work!	[Bom trabalho!]
Keep up the good work!	[Continue o bom trabalho!]

Frases para uma resposta incorreta:

No. Please try again.	[Não, tente novamente.]
Wrong. Try once more.	[Errado. Tente mais uma vez.]
Don't give up!	[Não desista!]
No. Keep trying.	[Não. Continue tentando.]

Utilize geração de números aleatórios para escolher um número entre 1 a 4 que será utilizado para selecionar uma mensagem apropriada para cada resposta. Utilize uma estrutura `switch` no método `paint` para exibir as respostas.

6.33 Os sistemas mais sofisticados de instrução auxiliada por computador monitoram o desempenho do aluno durante um período de tempo. A decisão sobre começar um novo tópico freqüentemente se baseia no sucesso do aluno com tópicos anteriores. Modifique o programa do Exercício 6.32 para contar o número de respostas corretas e incorretas digitadas pelo aluno. Depois que o aluno digitar 10 respostas, o programa deve calcular a porcentagem de respostas corretas. Se a porcentagem for inferior a 75%, o programa deve imprimir **Please ask your instructor for extra help** [Peça ao seu professor ajuda extra] e reiniciar o programa para outro aluno poder experimentá-lo.

6.34 Escreva um *applet* que reproduz o jogo “adivinhe o número” como segue: o programa escolhe o número para ser adivinhado selecionando um inteiro aleatório no intervalo 1 a 1000. O *applet* exibe o *prompt* **Guess a number between 1 and 1000** [Advinhe um número entre 1 e 1000] ao lado de um *JTextField*. O jogador digita um primeiro palpite no *JTextField* e pressiona a tecla *Enter*. Se o palpite do jogador estiver incorreto, o programa deve exibir **Too high. Try again.** [Muito alto. Tente novamente.] ou **Too low. Try again.** [Muito baixo. Tente novamente.] na barra de estado para ajudar o jogador a se aproximar da resposta correta e deve limpar o *JTextField* de modo que o usuário possa inserir o próximo palpite. Quando o usuário inserir a resposta correta, exiba **Congratulations. You guessed the number!** [Parabéns. Você adivinhou o número.] na barra de estado e limpe o *JTextField* para que o usuário possa jogar novamente. [Nota: a técnica de adivinhação empregada nesse problema é semelhante a uma *pesquisa binária*.]

6.35 Modifique o programa do Exercício 6.34 para contar o número de suposições que o jogador faz. Se o número for 10 ou menos, imprima **Either you know the secret or you got lucky!** [Ou você sabe o segredo ou tem muita sorte!] se o jogador adivinhar o número em 10 tentativas, imprima **Ahah! You know the secret!** [Ah! Você sabe o segredo!] se o jogador fizer mais que 10 tentativas, imprima **You should be able to do better!** [Você deveria ser capaz de fazer melhor.] Por que esse jogo não deve precisar de mais que 10 tentativas? Bem, a cada “bom palpite” o jogador deve ser capaz de eliminar metade dos números. Agora mostre por que qualquer número de 1 a 1000 pode ser adivinhado em 10 ou menos tentativas.

6.36 Escreva um método recursivo **power (base, exponent)** que, quando invocado, retorna

$$\text{base}^{\text{exponente}}$$

por exemplo, **power (3, 4) = 3 * 3 * 3 * 3**. Suponha que **exponente** é um inteiro maior que ou igual a 1. (Dica: a etapa de recursão deve usar a relação

$$\text{base}^{\text{exponente}} = \text{base} \cdot \text{base}^{\text{exponente}-1}$$

e a condição de terminação ocorre quando **exponent** é igual a 1 porque

$$\text{base}^1 = \text{base}$$

Incorpore esse método em um *applet* que permite ao usuário digitar a **base** e o **exponente**.)

6.37 (*Torres de Hanói*) Todo novo cientista de computação deve lidar com certos problemas clássicos, e as Torres de Hanói (ver Fig. 6.22) é um dos mais famosos desses problemas. Diz a lenda que, em um templo no Extremo Oriente, os sacerdotes estavam tentando mover uma pilha de discos de um pino para outro. A pilha inicial tinha 64 discos sobrepostos em um pino e ordenados de baixo para cima por tamanho decrescente. Os sacerdotes tentavam mover a pilha desse pino para um segundo pino com as restrições de que apenas um disco podia ser movido por vez e em nenhum momento um disco maior podia ser colocado sobre um disco menor. Havia um terceiro pino disponível para conter os discos temporariamente. Supostamente, o mundo acabaria quando os sacerdotes completassem a tarefa, portanto havia pouco incentivo para nós facilitarmos seus esforços.

Vamos supor que os sacerdotes estivessem tentando mover os discos do pino 1 para o pino 3. Desejamos desenvolver um algoritmo que imprimirá a seqüência precisa de transferências de discos de um pino para outro.

Se abordássemos esse problema com métodos convencionais, rapidamente ficaríamos desesperados gerenciando os discos. Em vez disso, se abordássemos o problema com a recursão em mente, ele imediatamente se tornaria tratável. Mover n discos pode ser visualizado em termos de mover somente $n - 1$ discos (e daí a recursão) como segue:

- Mover $n - 1$ discos do pino 1 para o pino 2, utilizando o pino 3 como área de armazenamento temporário.
- Mover o último disco (o maior) do pino 1 para o pino 3.
- Mover os $n - 1$ discos do pino 2 para o pino 3, utilizando o pino 1 como área de armazenamento temporário.

O processo termina quando a última tarefa envolver mover $n = 1$ disco (isto é, o caso básico). Essa tarefa é realizada movendo-se o disco diretamente sem a necessidade de uma área de armazenamento temporário.

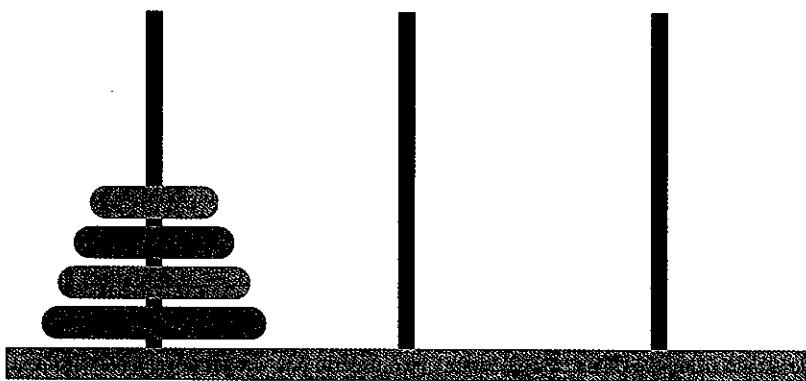


Fig. 6.22 Torres de Hanói para o caso com quatro discos.

Escreva um *applet* para resolver o problema das Torres de Hanói. Permita que o usuário digite o número de discos em um **JTextField**. Utilize um método recursivo **tower** com quatro parâmetros:

- O número de discos a serem movidos;
- O pino em que esses discos inicialmente estão empilhados;
- O pino para o qual essa pilha de discos deve ser movida;
- O pino a ser utilizado como área de armazenamento temporário.

O programa deve exibir em uma **JTextArea** com funcionalidade de rolagem as instruções exatas necessárias para mover os discos do pino inicial para o pino de destino. Por exemplo, para mover uma pilha de três discos do pino 1 para o pino 3, o programa deve imprimir a seguinte série de movimentos:

```
1 → 3 (significa mover um disco do pino 1 para o pino 3.)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3
```

6.38 Qualquer programa que possa ser implementado recursivamente pode ser implementado iterativamente, embora às vezes com mais dificuldade e menos clareza. Tente escrever uma versão iterativa das Torres de Hanói. Se você for bem-sucedido, compare a versão iterativa com a versão recursiva que você desenvolveu no Exercício 6.37. Investigue questões de desempenho, clareza e sua capacidade de demonstrar a correção dos programas.

6.39 (*Visualizando a recursão*) É interessante observar a recursão “em ação”. Modifique o método fatorial da Fig. 6.12 para imprimir a variável local e o parâmetro de chamada recursiva dele. Para cada chamada recursiva, exiba as saídas em uma linha separada e adicione um nível de recuo. Faça o melhor que você puder para tornar a saída limpa, interessante e significativa. O objetivo aqui é projetar e implementar um formato de saída que ajude uma pessoa a entender melhor a recursão. Você pode querer adicionar essas capacidades de exibição aos muitos outros exemplos e exercícios de recursão ao longo de todo este texto.

6.40 O máximo divisor comum dos inteiros **x** e **y** é o maior inteiro que divide **x** e **y**. Escreva um método recursivo **gcd** que retorna o máximo divisor comum de **x** e **y**. O **gcd** de **x** e **y** é definido recursivamente como segue: se **y** for igual a 0, então **gcd(x, y)** é **x**; caso contrário, **gcd(x, y)** é **gcd(y, x % y)**, onde **%** é o operador de módulo. Utilize esse método para substituir o que você escreveu no *applet* do Exercício 6.28.

6.41 Os Exercícios 6.31 a 6.33 desenvolveram um programa de instrução auxiliada por computador (CAI) para ensinar multiplicação a um aluno da escola elementar. Esse exercício sugere aprimoramentos nesse programa.

- Modifique o programa para permitir que o usuário especifique um nível de dificuldade. O nível 1 significa utilizar somente números de um único dígito nos problemas, o nível 2 significa utilizar números com dois dígitos, etc.

- b) Modifique o programa para permitir que o usuário selecione os tipos de problemas aritméticos que ele(a) deseja estudar. A opção 1 significa somente problemas de adição, 2 significa somente problemas de subtração, 3 significa somente problemas de multiplicação, 4 significa somente problemas de divisão e 5 significa problemas de todos esses tipos misturados aleatoriamente.

6.42 Escreva um método **distance** que calcula a distância entre dois pontos ($x1, y1$) e ($x2, y2$). Todos os números e valores retornados devem ser do tipo **double**. Incorpore esse método a um *applet* que permite que o usuário digite as coordenadas dos pontos.

6.43 O que faz o seguinte método?

```
// O parâmetro b deve ser um inteiro
// positivo para evitar recursão infinita
public int mystery( int a, int b )
{
    if ( b == 1 )
        return a;
    else
        return a + mystery( a, b - 1 );
}
```

6.44 Depois de determinar o que faz o programa do Exercício 6.43, modifique o método para operar adequadamente depois de eliminar a restrição de o segundo argumento ser não-negativo. Além disso, incorpore o método a um *applet* que permita digitar dois inteiros e testar o método.

6.45 Escreva um aplicativo que testa o maior número possível de métodos da biblioteca matemática da Fig. 6.2 que você puder. Exercite cada um desses métodos fazendo o programa imprimir tabelas de valores retornados para diversos valores de argumentos.

6.46 Localize o erro no seguinte método recursivo e explique como corrigi-lo:

```
public int sum( int n )
{
    if ( n == 0 )
        return 0;
    else
        return n + sum( n );
}
```

6.47 Modifique o programa de jogo de dados *craps* da Fig. 6.9 para permitir apostas. Inicialize a variável **bankBalance** com 1000 reais. Peça para o jogador digitar um **wager** (valor a apostar). Verifique se **wager** é menor que ou igual a **bankBalance** e, se não, faça o usuário digitar um novo **wager** até um **wager** válido ser digitado. Depois que um **wager** correto tiver sido digitado, execute um jogo de *craps*. Se o jogador ganhar, some **wager** a **bankBalance** e imprima o novo **bankBalance**. Se o jogador perder, subtraia **wager** de **bankBalance**, imprima o novo **bankBalance**, verifique se **bankBalance** tornou-se zero e, se isso ocorreu, imprima a mensagem "Desculpe, mas você faliu!". Enquanto o jogo se desenvolve, imprima várias mensagens para criar uma "conversa", como "Oh, parece que você vai quebrar, hein?" ou "Ah, vamos lá, arrisque sua sorte!" ou "Você está montado na grana. Agora é hora de trocar essas fichas e embolsar o dinheiro!". Implemente a "conversa" como um método separado que escolhe aleatoriamente o **string** a exibir.

6.48 Escreva um *applet* que utiliza um método **circleArea** para solicitar ao usuário o raio de um círculo e calcular e imprimir a área desse círculo.

7

Arrays

Objetivos

- Apresentar a estrutura de dados de *array*.
- Entender o uso de *arrays* para armazenar, classificar e pesquisar listas e tabelas de valores.
- Entender como declarar um *array*, inicializar um *array* e referir-se a elementos individuais de um *array*.
- Ser capaz de passar *arrays* para métodos.
- Entender técnicas básicas de classificação.
- Ser capaz de declarar e manipular *arrays* de múltiplos subscritos.

*Com soluções e lágrimas ele selecionou
aqueles de maior tamanho...*

Lewis Carroll

*Mire o final e nunca pare para duvidar;
nada é tão difícil, mas a busca irá descobri-lo.*

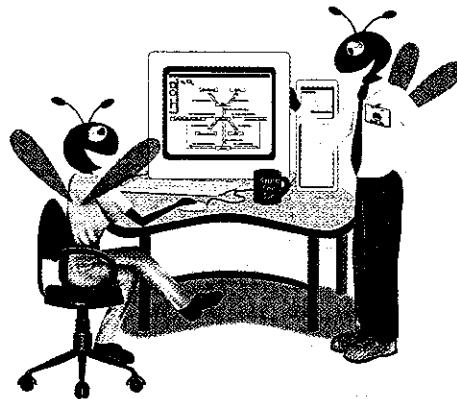
Robert Herrick

*Vai, pois, agora, escreve isso numa
tábua perante eles, regista-o num livro.*

Isaías 30:8

*Fechei vossas palavras na memória a chave:
vós mesmo a guardareis.*

William Shakespeare



Sumário do capítulo

- 7.1 Introdução**
- 7.2 Arrays**
- 7.3 Declarando e alocando arrays**
- 7.4 Exemplos com arrays**
 - 7.4.1 Alocando um array e inicializando seus elementos**
 - 7.4.2 Utilizando uma lista de inicializadores para inicializar os elementos de um array**
 - 7.4.3 Calculando o valor a armazenar em cada elemento de um array**
 - 7.4.4 Somando os elementos de um array**
 - 7.4.5 Utilizando histogramas para exibir dados de arrays graficamente**
 - 7.4.6 Utilizando os elementos de um array como contadores**
 - 7.4.7 Utilizando arrays para analisar resultados de pesquisas**
- 7.5 Referências e parâmetros por referência**
- 7.6 Passando arrays para métodos**
- 7.7 Classificando arrays**
- 7.8 Pesquisando arrays: pesquisa linear e pesquisa binária**
 - 7.8.1 Pesquisando um array com pesquisa linear**
 - 7.8.2 Pesquisando um array com pesquisa binária**
- 7.9 Arrays multidimensionais**
- 7.10 (Estudo de caso opcional) Pensando em objetos: colaboração entre objetos**

*Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão
 • Exercícios • Exercícios de recrusão • Seção especial: construindo seu próprio computador*

7.1 Introdução

Este capítulo serve como introdução para o importante tópico das estruturas de dados. Os *arrays* são estruturas de dados que consistem em itens de dados relacionados do mesmo tipo. São entidades “estáticas” no sentido de que, uma vez criadas, mantêm o mesmo tamanho, embora uma referência de *array* possa ser atribuída a um novo *array* de tamanho diferente. O Capítulo 19 apresenta estruturas de dados dinâmicas como listas, filas, pilhas e árvores que podem crescer e encolher enquanto os programas são executados. O Capítulo 20 discute a classe **Vector**, que é uma classe parecida com *array* cujos objetos podem crescer e encolher em resposta à alteração de requisitos de armazenamento de um programa Java. O Capítulo 21 apresenta as estruturas de dados predefinidas de Java que permitem ao programador utilizar estruturas de dados existentes para listas, filas, pilhas e árvores, em vez de “reinventar a roda”. A Collections API também oferece a classe **Arrays**, que define um conjunto de métodos utilitários para manipulação de *arrays*.

7.2 Arrays

O *array* é um grupo de posições contíguas na memória que possuem o mesmo nome e o mesmo tipo. Para referir-se a uma localização ou elemento particular no *array*, especificamos o nome do *array* e o *número da posição* (ou *índice* ou *subscrito*) do elemento particular no *array*.

A Fig. 7.1 mostra um *array* de inteiros chamado *c*. Esse *array* contém 12 *elementos*. O programa faz referência a qualquer um desses elementos fornecendo o nome do *array* seguido pelo número da posição do elemento particular entre colchetes (`[]`). O primeiro elemento em todos os *arrays* tem o *número de posição zero* (às vezes chamado de *zero-ésimo elemento*). Portanto, o primeiro elemento do *array* *c* é *c[0]*, o segundo elemento de *array* *c* é *c[1]*, o sétimo elemento de *array* *c* é *c[6]* e, em geral, o *i*-ésimo elemento do *array* *c* é *c[i - 1]*. Os nomes de *array* seguem as mesmas convenções que outros nomes de variável.

O número de posição entre colchetes é chamado mais formalmente de *subscrito* (ou *índice*). O subscrito deve ser um inteiro ou uma expressão de tipo inteiro. Se o programa utilizar uma expressão como um subscrito, o programa avalia a expressão para determinar o subscrito. Por exemplo, se assumirmos que a variável *a* é 5 e que a variável *b* é 6, então a instrução

```
c[ a + b ] += 2;
```

adiciona 2 ao elemento do *array* *c[11]*. Observe que um nome de *array* subscrito é um *lvalue* – ele pode ser utilizado no lado esquerdo de uma atribuição para colocar um novo valor em um elemento do *array*.

Nome do <i>array</i> (note que todos os elementos desse <i>array</i> têm o mesmo nome, <i>c</i>)	<i>c[0]</i>	-45
	<i>c[1]</i>	6
	<i>c[2]</i>	0
	<i>c[3]</i>	72
	<i>c[4]</i>	1543
	<i>c[5]</i>	-89
	<i>c[6]</i>	0
	<i>c[7]</i>	62
	<i>c[8]</i>	-3
	<i>c[9]</i>	1
	<i>c[10]</i>	6453
	<i>c[11]</i>	78

Fig. 7.1 Um *array* de 12 elementos.

Vamos examinar o *array* *c* na Fig. 7.1 mais atentamente. O *nome do array* é *c*. Todos os *arrays* em Java *conhecem* seu próprio comprimento e mantêm esta informação em uma variável chamada *length*. A expressão *c.length* acessa a variável *length* do *array* para determinar o comprimento do *array*. Os 12 elementos do *array* são *c[0]*, *c[1]*, *c[2]*, ..., *c[11]*. O valor de *c[0]* é -45, o valor de *c[1]* é 6, o valor de *c[2]* é 0, o valor de *c[7]* é 62 e o valor de *c[11]* é 78. Para calcular a soma dos valores contidos nos primeiros três elementos do *array* *c* e armazenar o resultado na variável *sum*, escreveríamos

```
sum = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

Para dividir o valor do sétimo elemento do *array* *c* por 2 e atribuir o resultado à variável *x*, escreveríamos

```
x = c[ 6 ] / 2;
```



Erro comum de programação 7.1

É importante notar a diferença entre “o sétimo elemento do array” e “elemento sete do array”. Uma vez que os subscritos de array iniciam em 0, “o sétimo elemento do array” tem o subscrito 6, enquanto o “elemento sete do array” tem um subscrito de 7 e é na realidade o oitavo elemento do array. Essa confusão é uma fonte comum de erros do tipo “erro por um”.

Os colchetes utilizados para incluir o subscrito de um *array* são um operador em Java. Os colchetes têm o mais alto nível de precedência em Java. A tabela na Fig. 7.2 mostra a precedência e a associatividade dos operadores apresentados até aqui. Eles são mostrados de cima para baixo na ordem crescente de precedência com sua associatividade e tipo. Consulte o Apêndice C para ver a tabela completa de precedência de operadores.

Operadores	Associatividade	Tipo
() [] .	da esquerda para a direita	mais alto
++ --	da direita para a esquerda	únário pós-fixo
++ - + - ! (tipo)	da direita para a esquerda	únário
* / %	da esquerda para a direita	multiplicativo
+ -	da esquerda para a direita	aditivo
< <= > >=	da esquerda para a direita	relacional
== !=	da esquerda para a direita	igualdade
&	da esquerda para a direita	E lógico booleano
^	da esquerda para a direita	OU lógico booleano exclusivo
	da esquerda para a direita	OU lógico booleano inclusivo
&&	da esquerda para a direita	E lógico
	da esquerda para a direita	OU lógico
? :	da direita para a esquerda	condicional
= += -= *= /= %=	da direita para a esquerda	atribuição

Fig. 7.2 Precedência e associatividade dos operadores discutidos até agora.

7.3 Declarando e alocando arrays

Os *arrays* são objetos que ocupam espaço na memória. Todos os objetos em Java (inclusive *arrays*) devem ser alocados dinamicamente com o operador `new`. Para um *array*, o programador especifica o tipo dos elementos do *array* e o número de elementos como parte da operação `new`. Para alocar 12 elementos para o *array* de inteiros `c` da Fig. 7.1, use a declaração

```
int c[] = new int[ 12 ];
```

A declaração precedente também pode ser executada em duas etapas, como segue:

```
int c[];           // declara o array
c = new int[ 12 ]; // aloca o array
```

Quando se aloca um *array*, cada elemento do *array* recebe por *default* um valor zero para as variáveis numéricas do tipo de dados primitivo, `false` para variáveis `boolean` ou `null` para referências (qualquer tipo não-primitivo).



Erro comum de programação 7.2

Diferentemente das declarações de arrays em várias outras linguagens de programação (como C ou C++), as declarações de arrays em Java não devem especificar o número de elementos no array nos colchetes depois do nome

de array; caso contrário, ocorre um erro de sintaxe. Por exemplo, a declaração `int c[12];` causa um erro de sintaxe.

O programa pode alocar memória para vários *arrays* com uma única declaração. A declaração seguinte reserva 100 elementos para o *array String b* e 27 elementos para o *array String x*:

```
String b[] = new String[ 100 ], x[] = new String[ 27 ];
```

Ao declarar um *array*, o tipo do *array* e os colchetes podem ser combinados no início da declaração para indicar que todos os identificadores na declaração representam *arrays*, como em

```
double[] array1, array2;
```

que declara *array1* e *array2* como *arrays* de valores `double`. Como mostrado anteriormente, a declaração e a inicialização do *array* podem ser combinadas na declaração. A declaração seguinte reserva 10 elementos para *array1* e 20 elementos para *array2*:

```
double[] array1 = new double[ 10 ],
        array2 = new double[ 20 ];
```

O programa pode declarar *arrays* de qualquer tipo de dados. É importante lembrar que cada elemento de um *array* de um tipo de dados primitivos, contém um valor do tipo de dados declarado. Por exemplo, cada elemento de um *array int* contém um valor `int`. Entretanto, em um *array* de um tipo não-primitivo, cada elemento do *array* é uma referência a um objeto do tipo de dados declarado para o *array*. Por exemplo, cada elemento de um *array String* é uma referência para um `String`. Em *arrays* que armazenam referências, as referências têm o valor `null` por *default*.

7.4 Exemplos com arrays

Esta seção apresenta diversos exemplos que usam *arrays* que demonstram a declaração de *arrays*, alocação de *arrays* e a manipulação de elementos de *arrays* de várias maneiras. Para simplificar, os exemplos nesta seção usam *arrays* que contêm elementos do tipo `int`. Lembre-se de que um programa pode declarar um *array* de qualquer tipo de dado.

7.4.1 Alocando um *array* e inicializando seus elementos

O aplicativo da Fig. 7.3 utiliza o operador `new` para alocar dinamicamente um *array* de 10 elementos `int` que são inicialmente zero (o valor *default* em um *array* do tipo `int`). O programa exibe os elementos do *array* em formato tabular em uma `JTextArea`.

A linha 12 declara *array* – uma referência capaz de referir-se a um *array* de elementos `int`. A linha 14 aloca os 10 elementos do *array* com `new` e inicializa a referência. O programa constrói sua saída no `String` chamado *output* que será exibido em uma `JTextArea` dentro de um diálogo de mensagem. A linha 16 acrescenta a *output* os títulos para as colunas exibidas pelo programa. As colunas representam o subscrito para cada elemento do *array* e o valor de cada elemento do *array*, respectivamente.

As linhas 19 e 20 utilizam uma estrutura `for` para acrescentar o número do subscrito (representado por `counter`) e o valor de cada elemento do *array* a *output*. Observe o uso de contagem baseada em zero (lembre-se, subscritos iniciam em 0) de modo que o laço possa acessar cada elemento do *array*. Além disso, observe a expressão `array.length` na condição de estrutura `for` para determinar o comprimento do *array*. Nesse exemplo, o comprimento do *array* é 10, de modo que o laço continua a ser executado enquanto o valor de variável de controle `counter` é menor que 10. Para um *array* de 10 elementos, os valores dos subscritos são 0 a 9, de modo que utilizar o operador menor que, <, garante que o laço não tente acessar um elemento além do fim do *array*.

```
1 // Fig. 7.3: InitArray.java
2 // Criando um array
3
```

Fig. 7.3 Inicializando os elementos de um *array* com zero (parte 1 de 2).

```

4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class InitArray {
8
9     // o método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        int array[];           // declara referência para um array
13
14        array = new int[ 10 ]; // aloca dinamicamente o array
15
16        String output = "Subscript\tValue\n";
17
18        // acrescenta o valor de cada elemento do array ao String output
19        for ( int counter = 0; counter < array.length; counter++ )
20            output += counter + "\t" + array[ counter ] + "\n";
21
22        JTextArea outputArea = new JTextArea();
23        outputArea.setText( output );
24
25        JOptionPane.showMessageDialog( null, outputArea,
26                                     "Initializing an Array of int Values",
27                                     JOptionPane.INFORMATION_MESSAGE );
28
29        System.exit( 0 );
30    }
31 }
```

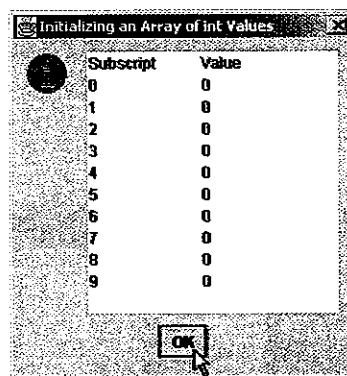


Fig. 7.3 Inicializando os elementos de um array com zero (parte 2 de 2).

7.4.2 Utilizando uma lista de inicializadores para inicializar os elementos de um array

O programa pode alocar e inicializar os elementos de um *array* na declaração seguindo a declaração com um sinal de igual e uma *lista de inicializadores* separados por vírgulas colocados entre chaves ({ e }). Nesse caso, o tamanho do *array* é determinado pelo número de elementos na lista de inicializadores. Por exemplo, a declaração

```
int n[] = { 10, 20, 30, 40, 50 };
```

cria um *array* de cinco elementos com os subscritos 0, 1, 2, 3 e 4. Observe que a declaração precedente não precisa do operador *new* para criar o objeto *array*. Quando o compilador encontra uma declaração de *array* que inclui uma lista de inicializadores, o compilador conta a quantidade de inicializadores na lista e configura uma operação (*new*) para alocar o número apropriado de elementos de *array*.

O aplicativo da Fig. 7.4 inicializa um *array* de inteiros com 10 valores (linha 14) e exibe o *array* em formato de tabela em uma **JTextArea** dentro de um diálogo de mensagem. O código para exibir os elementos do *array* é idêntico àquele da Fig. 7.3.

```

1 // Fig. 7.4: InitArray.java
2 // Inicializando um array com uma declaração
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class InitArray {
8
9     // o método main inicia a execução do aplicativo
10    public static void main( String args[] )
11    {
12        // Lista de inicializadores especifica a quantidade
13        // de elementos e o valor para cada elemento.
14        int array[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
15
16        String output = "Subscript\tValue\n";
17
18        // acrescenta o valor de cada elemento do array ao String output
19        for ( int counter = 0; counter < array.length; counter++ )
20            output += counter + "\t" + array[ counter ] + "\n";
21
22        JTextArea outputArea = new JTextArea();
23        outputArea.setText( output );
24
25        JOptionPane.showMessageDialog( null, outputArea,
26            "Initializing an Array with a Declaration",
27            JOptionPane.INFORMATION_MESSAGE );
28
29        System.exit( 0 );
30    }
31 }
```

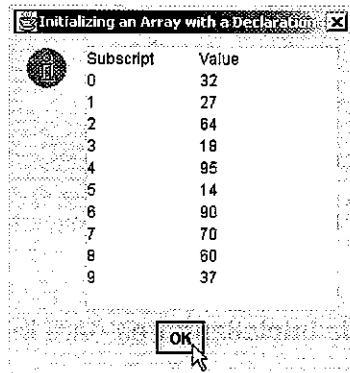


Fig. 7.4 Inicializando os elementos de um *array* com uma declaração.

7.4.3 Calculando o valor a armazenar em cada elemento de um *array*

Alguns programas calculam o valor armazenado em cada elemento do *array*. O aplicativo da Fig. 7.5 inicializa os elementos de um *array* de 10 elementos com os inteiros pares de 2 a 20 (2, 4, 6, ..., 20) e exibe o *array* em forma-

to de tabela. A estrutura **for** nas linhas 18 e 19 gera o valor de um elemento do *array* multiplicando o valor atual de **counter** (a variável de controle do laço) por 2 e adicionando 2.

```

1 // Fig. 7.5: InitArray.java
2 // Inicializa array com os inteiros pares de 2 a 20
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class InitArray {
8
9     // o método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        final int ARRAY_SIZE = 10;
13        int array[];                      // referência para array de int
14
15        array = new int[ ARRAY_SIZE ]; // aloca array
16
17        // calcula o valor para cada elemento do array
18        for ( int counter = 0; counter < array.length; counter++ )
19            array[ counter ] = 2 + 2 * counter;
20
21        String output = "Subscript\tValue\n";
22
23        for ( int counter = 0; counter < array.length; counter++ )
24            output += counter + "\t" + array[ counter ] + "\n";
25
26        JTextArea outputArea = new JTextArea();
27        outputArea.setText( output );
28
29        JOptionPane.showMessageDialog( null, outputArea,
30             "Initializing to Even Numbers from 2 to 20",
31             JOptionPane.INFORMATION_MESSAGE );
32
33        System.exit( 0 );
34    }
35 }
```

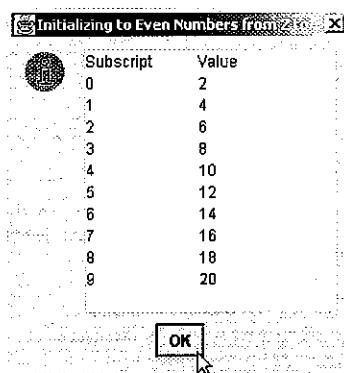


Fig. 7.5 Gerando valores para serem colocados em elementos de um *array*.

A linha 12 utiliza o qualificador **final** para declarar a variável constante **ARRAY_SIZE** cujo valor é 10. Lembre-se de que se devem inicializar as variáveis constantes antes de serem utilizadas, e elas não podem ser mo-

dificadas depois. Se se fizer uma tentativa para modificar uma variável **final** depois de ser declarada como mostrado na linha 12, o compilador emite uma mensagem parecida com

```
cannot assign a value to final variable nomeDaVariável
[Não é possível atribuir um valor à variável final nomeDaVariável]
```

Se se fizer uma tentativa de utilizar uma variável **final** antes de ela ser inicializada, o compilador emite a mensagem de erro

```
Variable nomeDaVariável may not have been initialized
[A variável nomeDaVariável pode não ter sido inicializada]
```

Variáveis constantes também são chamadas de *constants identificadas* ou *variáveis apenas de leitura (read-only)*. Tais variáveis freqüentemente podem tornar os programas mais legíveis. Observe que o termo “variável constante” é um oxímoro – uma contradição de termos – como “silêncio barulhento” ou “inocente culpa”.



Erro comum de programação 7.3

Atribuir um valor a uma variável constante depois que a variável foi inicializada é um erro de sintaxe.

7.4.4 Somando os elementos de um array

Freqüentemente, os elementos de um *array* representam uma série de valores que devem ser usados em um cálculo. Por exemplo, se os elementos de um *array* representam as notas de uma prova para uma turma, o professor pode querer totalizar os elementos do *array* e depois calcular a média da turma para aquela prova.

O aplicativo da Fig. 7.6 soma os valores contidos no *array* de inteiros de 10 elementos. O programa declara, aloca e inicializa o *array* na linha 12. A estrutura **for** nas linhas 16 e 17 executa os cálculos. [Nota: é importante lembrar que os valores sendo fornecidos como inicializadores para *array* normalmente seriam lidos para o programa. Por exemplo, em um *applet*, o usuário poderia digitar os valores por meio de um **JTextField** ou, em um aplicativo, os valores poderiam ser lidos de um arquivo em disco (discutido no Capítulo 16). Ler os dados para um programa torna o programa mais flexível, porque ele pode ser usado com conjuntos diferentes de dados.]

```

1 // Fig. 7.6: SumArray.java
2 // Totaliza os valores dos elementos de um array
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class SumArray {
8
9     // o método main inicia a execução do aplicativo
10    public static void main( String args[] )
11    {
12        int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13        int total = 0;
14
15        // adiciona o valor de cada elemento a total
16        for ( int counter = 0; counter < array.length; counter++ )
17            total += array[ counter ];
18
19        JOptionPane.showMessageDialog( null,
20             "Total of array elements: " + total,
21             "Sum the Elements of an Array",
22             JOptionPane.INFORMATION_MESSAGE );
23
24        System.exit( 0 );
25    }
26 }
```

Fig. 7.6 Calculando a soma dos elementos de um *array* (parte 1 de 2).

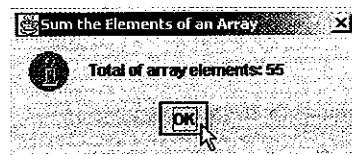


Fig. 7.6 Calculando a soma dos elementos de um array (parte 2 de 2).

7.4.5 Utilizando histogramas para exibir dados de arrays graficamente

Muitos programas apresentam dados aos usuários graficamente. Por exemplo, os valores numéricos freqüentemente são exibidos como barras em um gráfico de barras. Num gráfico com esse, barras mais longas representam valores numéricos maiores. Uma maneira simples de exibir dados numéricos graficamente é com um *histograma* que mostre cada valor numérico como uma barra de asteriscos (*).

Nosso próximo aplicativo (Fig. 7.7) lê números de um *array* e exibe as informações sob a forma de um gráfico de barras (histograma). O programa exibe cada número seguido por uma barra que consiste naquela quantidade de asteriscos. O laço aninhado **for** (linhas 17 a 24) acrescenta as barras ao **String** que será exibido na **JTextArea outputArea** dentro de um diálogo de mensagem. Observe a condição de continuação do laço da estrutura interna **for** na linha 22 (**stars <= array[counter]**). Cada vez que o programa atinge a estrutura interna **for**, o laço conta de 1 até **array[counter]**, usando, assim, um valor que está em **array** para determinar o valor final da variável de controle **stars** e a quantidade de asteriscos a exibir.

```

1 // Fig. 7.8: Histograma.java
2 // Programa de impressão de histograma
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class Histogram {
8
9     // o método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        int array[] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
13
14        String output = "Element\tValue\tHistogram";
15
16        // para cada elemento do array, desenha uma barra no histograma
17        for ( int counter = 0; counter < array.length; counter++ ) {
18            output +=
19                "\n" + counter + "\t" + array[ counter ] + "\t";
20
21            // imprime uma barra de asteriscos
22            for ( int stars = 0; stars < array[ counter ]; stars++ )
23                output += "*";
24        }
25
26        JTextArea outputArea = new JTextArea();
27        outputArea.setText( output );
28
29        JOptionPane.showMessageDialog( null, outputArea,
30                                     "Histogram Printing Program",
31                                     JOptionPane.INFORMATION_MESSAGE );

```

Fig. 7.7 Programa que imprime histogramas (parte 1 de 2).

```

32
33     System.exit( 0 );
34 }
35 }
```

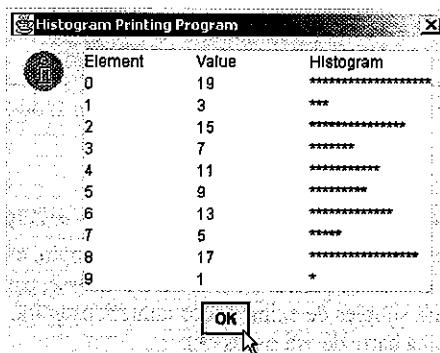


Fig. 7.7 Programa que imprime histogramas (parte 2 de 2).

7.4.6 Utilizando os elementos de um array como contadores

Algumas vezes os programas usam uma série de variáveis contadoras para resumir dados, como os resultados de uma pesquisa. No Capítulo 6, usamos uma série de contadores em nosso programa de lançamento de dados para contar o número de ocorrências de cada face de um dado de seis faces quando o programa lança os dados 6000 vezes. Também indicamos que existe um método mais elegante de escrever o programa da Fig. 6.8. Uma versão com *array* desse aplicativo é mostrada na Fig. 7.8.

```

1 // Fig. 7.8: RollDie.java
2 // Lança um dado de seis faces 6000 vezes
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class RollDie {
8
9     // o método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        int face, frequency[] = new int[ 7 ];
13
14        // lança o dado 6000 vezes
15        for ( int roll = 1; roll <= 6000; roll++ ) {
16            face = 1 + ( int ) ( Math.random() * 6 );
17
18            // usa o valor da face como subscrito para o array de freqüências
19            ++frequency[ face ];
20        }
21
22        String output = "Face\tFrequency";
23
24        // acrescenta freqüências ao String output
25        for ( face = 1; face < frequency.length; face++ )
26            output += "\n" + face + "\t" + frequency[ face ];
```

Fig. 7.8 Programa de jogo de dados que utiliza arrays em vez de switch (parte 1 de 2).

```

27
28     JTextArea outputArea = new JTextArea();
29     outputArea.setText( output );
30
31     JOptionPane.showMessageDialog( null, outputArea,
32         "Rolling a Die 6000 Times",
33         JOptionPane.INFORMATION_MESSAGE );
34
35     System.exit( 0 );
36 }
37 }
```

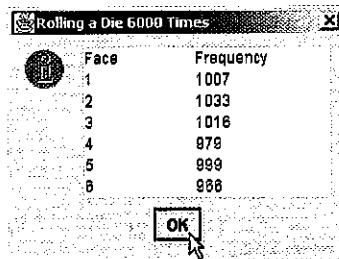


Fig. 7.8 Programa de jogo de dados que utiliza arrays em vez de `switch` (parte 2 de 2).

O programa usa o array de sete elementos `frequency` para contar as ocorrências de cada face do dado. Nesse programa, as linhas 20 a 46 da Fig. 6.8 são substituídas pela linha 19. A linha 19 utiliza o valor aleatório `face` como o subscrito para array `frequency`, para determinar qual é o elemento que o programa deve incrementar durante cada iteração do laço. Uma vez que o cálculo de número aleatório na linha 19 produz números de 1 a 6 (os valores para um dado de seis faces), o array `frequency` deve ser suficientemente grande para armazenar seis contadores. Entretanto, neste programa, optamos por usar um array de sete elementos. Ignoramos o primeiro elemento do array, `frequency[0]`, porque é mais lógico fazer o valor da face 1 incrementar `frequency[1]` do que `frequency[0]`. Isso nos permite usar cada valor de face diretamente como um subscrito para o array `frequency`.



Boa prática de programação 7.1

Empenhe-se na clareza do programa. Às vezes vale a pena trocar a utilização mais eficiente da memória ou do tempo de processador pela escrita de programas mais claros.



Dica de desempenho 7.1

Às vezes as considerações de desempenho superam em muito as considerações de clareza.

Além disso, as linhas 25 e 26 desse programa substituem as linhas 52 a 55 da Fig. 6.8. Uma vez que podemos fazer um laço para percorrer o array `frequency`, não precisamos enumerar cada linha de texto para exibir na `JTextArea` como fizemos na Fig. 6.8.

7.4.7 Utilizando arrays para analisar resultados de pesquisas

Nosso próximo exemplo usa arrays para resumir os resultados de dados coletados em uma pesquisa. Considere a definição do problema:

Pedi-se a 40 alunos para avaliar a qualidade da comida do restaurante universitário em uma escala de 1 a 10 (1 significa péssimo e 10 significa excelente). Coloque as 40 respostas em um array de inteiros e resuma os resultados da pesquisa.

Este é um aplicativo típico de processamento de array (veja a Fig. 7.9). Queremos resumir o número de respostas de cada tipo (isto é, 1 a 10). O array `responses` é um array de inteiros de 40 elementos com as respostas dos alunos à pesquisa. Utilizamos um array de 11 elementos `frequency` para contar o número de ocorrências de ca-

da resposta. Como na Fig. 7.8, ignoramos o primeiro elemento (`frequency[0]`) porque é mais lógico fazer a resposta 1 incrementar `frequency[1]` do que `frequency[0]`. Isso permite utilizar cada resposta diretamente como um subscrito para o array `frequency`. Usa-se cada elemento do array como contador para uma das respostas da pesquisa.

```

1 // Fig. 7.9: StudentPoll.java
2 // Programa de pesquisa de aluno
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class StudentPoll {
8
9     // o método main inicia a execução do aplicativo Java
10    public static void main( String args[] )
11    {
12        int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
13                            1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
14                            6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
15                            5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
16        int frequency[] = new int[ 11 ];
17
18        // para cada resposta, seleciona o valor de um elemento
19        // do array responses e usa aquele valor como subscrito do
20        // array frequency para determinar o elemento a incrementar
21        for ( int answer = 0; answer < responses.length; answer++ )
22            ++frequency[ responses[ answer ] ];
23
24        String output = "Rating\tFrequency\n";
25
26        // acrescenta freqüências ao string output
27        for ( int rating = 1; rating < frequency.length; rating++ )
28            output += rating + "\t" + frequency[ rating ] + "\n";
29
30        JTextArea outputArea = new JTextArea();
31        outputArea.setText( output );
32
33        JOptionPane.showMessageDialog( null, outputArea,
34                                     "Student Poll Program",
35                                     JOptionPane.INFORMATION_MESSAGE );
36
37        System.exit( 0 );
38    }
39 }
```

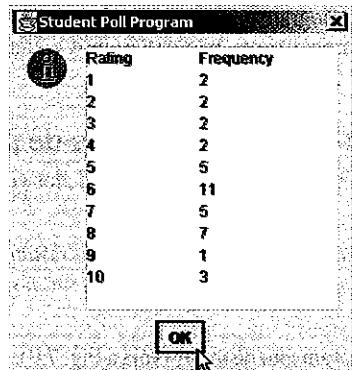


Fig. 7.9 Um programa simples de análise de pesquisa de alunos.

O laço `for` (linhas 21 a 22) pega uma resposta por vez do array `responses` e incrementa um dos 10 contadores no array `frequency` (`frequency[1]` a `frequency[10]`). A instrução fundamental no laço é a linha 22, que incrementa o contador apropriado em `frequency`, dependendo do valor de `responses[answer]`.

Consideremos várias iterações do laço `for`. Quando o contador `answer` for 0, `responses[answer]` é o valor do primeiro elemento do array `responses` (isto é, 1), de modo que o programa interpreta `++frequency[responses[answer]]`; como

```
++frequency[ 1 ];
```

que incrementa o elemento um do array. Para avaliar a expressão, inicie com o valor no conjunto mais interno de colchetes (`answer`). Uma vez que você conheça o valor de `answer`, insira esse valor na expressão e avalie o próximo conjunto externo de colchetes (`responses[answer]`). Depois, utilize esse valor como o subscrito para o array `frequency` para determinar qual contador incrementar.

Quando `answer` for 1, `responses[answer]` será o valor do segundo elemento de `responses` (2), de modo que o programa interpreta `++frequency[responses[answer]]`; como

```
++frequency[ 2 ];
```

que incrementa o elemento dois do array (o terceiro elemento do array).

Quando `answer` for 2, `responses[answer]` será o valor do terceiro elemento de array `responses` (6), de modo que o programa interpreta `++frequency[responses[answer]]`; como

```
++frequency[ 6 ];
```

que incrementa o elemento seis do array (o sétimo elemento do array) e assim por diante. Independentemente do número de respostas processadas na pesquisa, o programa precisa de só um array de 11 elementos (ignorando o elemento zero) para resumir os resultados, porque todos os valores de resposta estão entre 1 e 10 e os valores de subscrito para um array de 11 elementos são 0 a 10. Além disso, observe que os resultados resumidos estão corretos, porque os elementos do array `frequency` foram inicializados com zero quando o array foi alocado com `new`.

Se os dados contivessem valores inválidos, como 13, o programa tentaria adicionar 1 a `frequency[13]`. Isso está além dos limites do array. Nas linguagens de programação C e C++, tal referência seria permitida pelo compilador e durante a execução. O programa “andaria” além do fim do array até onde ele pensasse que o elemento de número 13 estivesse localizado e adicionaria 1 a qualquer coisa que eventualmente estivesse nessa posição da memória. Isso poderia potencialmente modificar outra variável no programa ou mesmo resultar no término prematuro do programa. Java fornece mecanismos para evitar o acesso a elementos além dos limites do array.

Erro comum de programação 7.4



Fazer referência a um elemento além dos limites de um array é um erro de lógica.

Dica de teste e depuração 7.1



Quando se executa um programa Java, o interpretador Java verifica os subscritos dos elementos do array para certificar-se de que eles são válidos (isto é, todos os subscritos devem ser maiores que ou iguais a 0 e menores que o comprimento do array). Se houver um subscrito inválido, Java gera uma exceção.

Dica de teste e depuração 7.2



As exceções indicam que ocorreu um erro em um programa. O programador pode escrever código para se recuperar de uma exceção e continuar a execução do programa, em vez de terminá-lo anormalmente. Quando ocorre uma referência para array inválida, Java gera uma exceção do tipo `ArrayIndexOutOfBoundsException`. O Capítulo 14 aborda o tratamento de exceção em detalhes.

Dica de teste e depuração 7.3



Ao percorrer um array com um laço, o subscrito de array nunca deve ficar abaixo de 0 e sempre deve ser menor que o número total de elementos no array (um a menos que o tamanho do array). A condição de terminação do laço deve evitar o acesso a elementos fora desse intervalo.

Dica de teste e depuração 7.4



Os programas devem validar todos os valores de entrada para impedir que informações errôneas afetem os cálculos de um programa.

7.5 Referências e parâmetros por referência

Duas maneiras de passar argumentos para métodos (ou funções) em muitas linguagens de programação (como C e C++) são a *passagem por valor* e a *passagem por referência* (também chamadas de *chamada por valor* e *chamada por referência*). Quando um argumento é passado por valor, faz-se uma *cópia* do valor do argumento e passa-se para o método chamador.



Dica de teste e depuração 7.5

Com a chamada por valor, alterações na cópia feitas pelo método chamado não afetam o valor da variável original no método que chamou. Isso impede os efeitos colaterais acidentais que tanto entravam o desenvolvimento de sistemas de software confiáveis e corretos.

Quando um argumento é passado por referência, o chamador dá ao método chamado a capacidade de acessar diretamente os dados do chamador e modificar esses dados se o método chamado assim o escolher. A passagem por referência melhora o desempenho, porque ela elimina o *overhead* de copiar quantidades grandes de dados.



Observação de engenharia de software 7.1

Diferentemente de outras linguagens, Java não permite que o programador escolha entre passar cada argumento por valor ou por referência. As variáveis dos tipos de dados primitivos sempre são passadas por valor. Os objetos não são passados para métodos; em vez disso, são passadas aos métodos referências para objetos. As próprias referências são passadas por valor – uma cópia da referência é passada para o método chamado. Quando um método recebe uma referência para um objeto, o método pode manipular o objeto diretamente.



Observação de engenharia de software 7.2

Ao retornar as informações de um método através de uma instrução `return`, variáveis dos tipos de dados primitivos sempre são retornados por valor (isto é, é devolvida uma cópia) e os objetos sempre são retornados por referência (isto é, é devolvida uma referência para o objeto).

Para passar para um método uma referência para um objeto, simplesmente especifique o nome da referência na chamada do método. Mencionar a referência pelo seu nome de parâmetro no corpo do método chamado, na realidade, faz referência ao objeto original na memória e o objeto original pode ser acessado diretamente pelo método chamado.

Os *arrays* são tratados como objetos em Java; portanto, os *arrays* são passados aos métodos por referência – um método chamado pode acessar os elementos dos *arrays* originais do chamador. O nome de um *array* é, na verdade, uma referência para um objeto que contém os elementos do *array* e a variável de instância `length`, que indica o número de elementos no *array*. Na próxima seção, demonstramos a passagem por valor e a passagem por referência utilizando *arrays*.



Dica de desempenho 7.2

Passar arrays por referência faz sentido por razões de desempenho. Se os arrays fossem passados por valor, uma cópia de cada elemento seria passada. Para os arrays grandes passados com frequência, isso desperdiçaria tempo e consumiria considerável capacidade de armazenamento para as cópias dos arrays.

7.6 Passando arrays para métodos

Para passar um *array* como argumento para um método, especifique o nome do *array* sem nenhum colchete. Por exemplo, se o *array* `hourlyTemperatures` é declarado como

```
int hourlyTemperatures[] = new int[ 24 ];
```

a chamada de método

```
modifyArray( hourlyTemperatures );
```

passa o *array* `hourlyTemperatures` para o método `modifyArray`. Em Java, cada objeto de um *array* “conhece” seu próprio tamanho (através da variável de instância `length`). Portanto, quando passamos um objeto de *array* para um método, não precisamos passar separadamente o tamanho do *array* como um argumento adicional.

Embora *arrays* inteiros e objetos mencionados por elementos individuais de um *array* de tipo não-primitivo sejam passados por referência, os elementos individuais de arrays dos tipos de dados primitivos são passados por valor exatamente como são passadas as variáveis simples. Esses pedaços de dados simples e isolados são chamados de

escalares ou quantidades escalares. Para passar um elemento de um *array* para um método, utilize o nome do *array* seguido pelo subscrito como um argumento na chamada de método.

Para um método receber um *array* através de uma chamada de método, a lista de parâmetros do método deve especificar um *array* como parâmetro (ou vários, se mais de um *array* deve ser recebido). Por exemplo, o cabeçalho de método para o método **modifyArray** poderia ser escrito assim

```
void modifyArray( int b[] )
```

indicando que **modifyArray** espera receber um *array* de inteiros no parâmetro **b**. Como os *arrays* são passados por referência, quando o método chamado utiliza o nome de *array* **b**, ele se refere ao *array* real no chamador (*o array hourlyTemperatures* na chamada precedente).

O *applet* da Fig. 7.10 demonstra a diferença entre passar um *array* inteiro e passar um elemento do *array*. Mais uma vez, estamos usando um *applet* aqui porque ainda não definimos um aplicativo que contenha outros métodos além de **main**. Ainda tiramos proveito de alguns recursos oferecidos gratuitamente em um *applet* (como a criação automática de um objeto de *applet* e as chamadas automáticas a **init**, **start** e **paint** pelo contêiner de *applets*). No Capítulo 9, apresentamos aplicativos que são executados em suas próprias janelas. Quando chegarmos lá, começaremos a ver aplicativos que contêm vários métodos.

As linhas 15 a 17 no método **init** definem a **JTextArea** denominada **outputArea** e a anexa ao painel de conteúdo do *applet*. A estrutura **for** nas linhas 26 e 27 acrescenta os cinco elementos de **array** (um *array* de valores **int**) ao **String** denominado **output**. A linha 29 invoca o método **modifyArray** e passa **array** como argumento para ele. O método **modifyArray** (linhas 50 a 54) multiplica cada elemento por 2. Para ilustrar que os elementos de **array** foram modificados, a estrutura **for** nas linhas 34 e 35 acrescenta os cinco elementos de **array** a **output** novamente. Como mostra a captura de tela, o método **modifyArray** modificou o valor de cada elemento.

```

1 // Fig. 7.10: PassArray.java
2 // Passando arrays e elementos individuais de arrays para métodos
3
4 // Pacotes do núcleo de Java
5 import java.awt.Container;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class PassArray extends JApplet {
11
12     // inicializa applet
13     public void init()
14     {
15         JTextArea outputArea = new JTextArea();
16         Container container = getContentPane();
17         container.add( outputArea );
18
19         int array[] = { 1, 2, 3, 4, 5 };
20
21         String output =
22             "Effects of passing entire array by reference:\n" +
23             "The values of the original array are:\n";
24
25         // acrescenta os elementos do array original ao String output
26         for ( int counter = 0; counter < array.length; counter++ )
27             output += "    " + array[ counter ];
28
29         modifyArray( array );    // array passado por referência
30
31         output += "\n\nThe values of the modified array are:\n";

```

Fig. 7.10 Passando arrays e elementos individuais de arrays para métodos (parte 1 de 2).

```

32
33     // acrescenta os elementos do array modificado ao String output
34     for ( int counter = 0; counter < array.length; counter++ )
35         output += "    " + array[ counter ];
36
37     output += "\n\nEffects of passing array " +
38             "element by value:\n" +
39             "a[3] before modifyElement: " + array[ 3 ];
40
41     // tentativa de modificar array[ 3 ]
42     modifyElement( array[ 3 ] );
43
44     output += "\na[3] after modifyElement: " + array[ 3 ];
45     outputArea.setText( output );
46
47 } // fim do método init
48
49 // multiplica cada elemento de um array por 2
50 public void modifyArray( int array2[] )
51 {
52     for ( int counter = 0; counter < array2.length; counter++ )
53         array2[ counter ] *= 2;
54 }
55
56 // multiplica o argumento por 2
57 public void modifyElement( int element )
58 {
59     element *= 2;
60 }
61
62 } // fim da classe PassArray

```

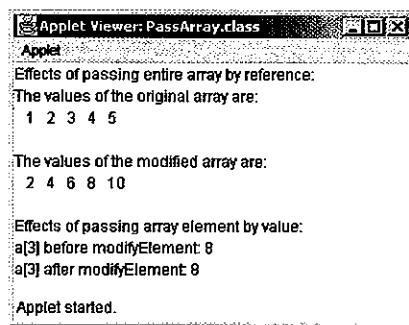


Fig. 7.10 Passando arrays e elementos individuais de arrays para métodos (parte 2 de 2).

A seguir, o programa demonstra que os elementos individuais de *arrays* de tipos primitivos são passados para os métodos por valor. Para mostrar o valor de `array[3]` antes de chamar o método `modifyElement`, as linhas 37 a 39 acrescentam o valor de `array[3]` (e outras informações) a `output`. A linha 42 invoca o método `modifyElement` e passa `array[3]` como argumento. Lembre-se de que `array[3]` é, na realidade, um valor `int` em `array`. Além disso, lembre-se de que valores de tipos primitivos são sempre passados aos métodos por valor. Portanto, o programa passa uma cópia de `array[3]`. O método `modifyElement` multiplica seu argumento por 2 e armazena o resultado em seu parâmetro `element`. Os parâmetros de métodos são variáveis locais, de modo que, quando termina o método `modifyElement`, a variável local `element` é destruída. Portanto, quando o programa retorna o controle para `init`, a linha 44 acrescenta o valor inalterado de `a[3]` a `output`. A linha 45 exibe os resultados na `JTextArea`.

7.7 Classificando arrays

Classificar dados (isto é, colocar os dados em alguma ordem particular como crescente ou decrescente) é uma das aplicações mais importantes da computação. O banco classifica todos os cheques pelo número da conta, de modo que possa preparar extratos bancários individuais no final de cada mês. As empresas de telefonia classificam suas listas de assinantes por sobrenome e, dentro desta classificação, pelo primeiro nome, para facilitar a localização de números de telefone.* Praticamente todas as organizações precisam classificar seus dados de alguma maneira e, em muitos casos, quantidades enormes de dados. Classificar dados é um problema intrigante que tem atraído alguns dos esforços mais intensos de pesquisa no campo da ciência da computação. Neste capítulo, discutimos um dos esquemas mais simples de classificação. Nos exercícios deste capítulo e dos Capítulos 19 e 21, investigamos esquemas mais complexos que oferecem desempenho superior.

Dica de desempenho 7.3



Às vezes, os algoritmos mais simples demonstram um desempenho deficiente. Sua virtude é que são fáceis de programar, testar e depurar. Às vezes são necessários algoritmos mais complexos para alcançar um desempenho máximo.

A Fig. 7.11 classifica os valores de **array** (um array de valores **int** com 10 elementos) em ordem crescente. A técnica que utilizamos se chama *classificação de bolhas* (*bubble sort*) ou *classificação por afundamento*, porque os valores menores “borbulham” gradualmente para o topo do *array* (isto é, em direção ao primeiro elemento) como bolhas de ar subindo na água, enquanto os valores maiores afundam para o fundo (fim) do *array*. A técnica usa laços aninhados para fazer várias passagens pelo *array*. Em cada passagem, pares de elementos sucessivos são comparados. Se um par estiver na ordem crescente (ou os valores forem iguais), a *bubble sort* deixa os valores como estão. Se um par estiver na ordem decrescente, a *bubble sort* troca seus valores no *array*. O *applet* contém os métodos **init**, **bubbleSort** e **swap**. O método **init** (linhas 13 a 16) inicializa o *applet*. O método **bubbleSort** (linhas 39 a 58) é chamado de **init** para classificar os elementos de **array**. O método **bubbleSort** chama o método **swap** (linhas 61 a 68) quando necessário, para trocar dois elementos do *array*.

As linhas 24 e 25 acrescentam os valores originais de **array** ao **String** chamado de **output**. A linha 27 invoca o método **bubbleSort** e passa **array** como o *array* a ser classificado.

O método **bubbleSort** recebe o *array* como parâmetro **array2**. A estrutura aninhada **for** nas linhas 42 a 56 realiza a classificação. O laço externo controla o número de passagens do *array*. O laço interno controla as comparações e trocas (se necessárias) dos elementos durante cada passagem.

```

1 // Fig. 7.11: BubbleSort.java
2 // Classifica os valores de um array em ordem crescente
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class BubbleSort extends JApplet {
11
12     // inicializa o applet
13     public void init()
14     {
15         JTextArea outputArea = new JTextArea();
16         Container container = getContentPane();
17         container.add( outputArea );
18
19         int array[] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
20
21         String output = "Data items in original order\n";

```

Fig. 7.11 Classificando um *array* com o algoritmo *bubble sort* (parte 1 de 2).

* N. de R. No Brasil, a classificação é feita pelo primeiro nome.

```

22
23     // acrescenta valores originais do array ao String output
24     for ( int counter = 0; counter < array.length; counter++ )
25         output += "    " + array[ counter ];
26
27     bubbleSort( array );    // ordena o array
28
29     output += "\n\nData items in ascending order\n";
30
31     // acrescenta valores ordenados do array ao String output
32     for ( int counter = 0; counter < array.length; counter++ )
33         output += "    " + array[ counter ];
34
35     outputArea.setText( output );
36 }
37
38 // ordena os elementos do array com bubble sort
39 public void bubbleSort( int array2[] )
40 {
41     // laço para controlar o número de passagens
42     for ( int pass = 1; pass < array2.length; pass++ ) {
43
44         // laço para controlar o número de comparações
45         for ( int element = 0;
46             element < array2.length - 1;
47             element++ ) {
48
49             // compara elementos adjacentes e os troca de lugar se
50             // o primeiro elemento for maior que o segundo elemento
51             if ( array2[ element ] > array2[ element + 1 ] )
52                 swap( array2, element, element + 1 );
53
54         } // fim do laço para controlar comparações
55
56     } // fim do laço para controlar passagens
57
58 } // fim do método bubbleSort
59
60 // troca dois elementos de um array
61 public void swap( int array3[], int first, int second )
62 {
63     int hold;    // área de armazenamento temporário para troca
64
65     hold = array3[ first ];
66     array3[ first ] = array3[ second ];
67     array3[ second ] = hold;
68 }
69
70 } // fim da classe BubbleSort

```

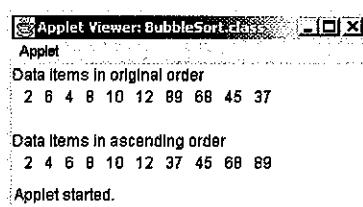


Fig. 7.11 Classificando um array com o algoritmo *bubble sort* (parte 2 de 2).

O método `bubbleSort` primeiro compara `array2[0]` com `array2[1]`, depois `array2[1]` com `array2[2]`, depois `array2[2]` com `array2[3]` e assim por diante, até completar a passagem comparando `array2[8]` com `array2[9]`. Embora haja 10 elementos, o laço de comparação faz apenas nove comparações. As comparações executadas em um *bubble sort* podem fazer um valor grande se mover para baixo no *array* (afundar) muitas posições em uma única passagem, mas um valor pequeno pode mover-se para cima (borbulhar) apenas uma posição. Na primeira passagem, garante-se que o elemento de maior valor afundará para a parte inferior do *array*, `array2[9]`. Na segunda passagem, garante-se que o segundo maior valor afundará para `array2[8]`. Na nona passagem, o nono maior valor afunda para `array2[1]`. Isso deixa o menor valor em `array2[0]`, portanto são necessárias nove passagens para classificar um *array* de 10 elementos.

Se uma comparação revela que os dois elementos estão na ordem decrescente, `bubbleSort` chama o método `swap` para trocar de posição os dois elementos, de modo que fiquem em ordem crescente no *array*. O método `swap` recebe uma referência para o *array* (que ele chama de `array3`) e dois inteiros que representam os subscritos dos dois elementos do *array* a trocar. A troca é realizada pelas três atribuições

```
hold = array3[ first ];
array3[ first ] = array3[ second ];
array3[ second ] = hold;
```

onde a variável extra `hold` armazena temporariamente um dos dois valores que estão sendo trocados de lugar. A troca não pode ser realizada com apenas as duas atribuições

```
array3[ first ] = array3[ second ];
array3[ second ] = array3[ first ];
```

se `array3[first]` for 7 e `array3[second]` for 5, após a primeira atribuição os dois elementos do *array* conterão 5 e o valor 7 é perdido. Daí a necessidade da variável extra `hold`.

A principal virtude do algoritmo *bubble sort* é que ele é fácil de programar. Entretanto, a execução do algoritmo *bubble sort* é muito lenta. Isso fica evidente quando classificamos *arrays* grandes. No Exercício 7.11, pedimos que você desenvolva versões mais eficientes do *bubble sort*. Outros exercícios investigam alguns algoritmos de classificação que são muito mais eficientes do que o *bubble sort*. Alguns cursos mais avançados (frequentemente intitulados “Estruturas de dados”, “Algoritmos” ou “Complexidade computacional”) investigam classificação e pesquisa em maior profundidade.

7.8 Pesquisando arrays: pesquisa linear e pesquisa binária

Frequentemente, o programador trabalhará com grandes quantidades de dados armazenados em *arrays*. Pode ser necessário determinar se o *array* contém um valor que corresponde a um certo *valor-chave*. O processo de localizar o valor de um elemento particular em um *array* chama-se *pesquisa*. Nesta seção, discutimos duas técnicas de pesquisa – a técnica simples da *pesquisa linear* e a técnica mais eficiente da *pesquisa binária*. Os Exercícios 7.31 e 7.32, no final deste capítulo, pedem para você implementar versões recursivas da pesquisa linear e da pesquisa binária.

7.8.1 Pesquisando um array com pesquisa linear

No *applet* da Fig. 7.12, o método `linearSearch` (definido nas linhas 52 a 62) utiliza uma estrutura `for` (linhas 55 a 59) que contém uma estrutura `if` para comparar cada elemento de um *array* com uma *chave de pesquisa*. Se a chave da pesquisa for encontrada, o método retorna o valor de subscrito para o elemento indicando a posição exata da chave de pesquisa no *array*. Se a chave de pesquisa não for encontrada, o método retorna `-1` para indicar que a chave de pesquisa não foi localizada. Retornamos `-1` porque ele não é um número de subscrito válido. Se o *array* que está sendo pesquisado não estiver em alguma ordem particular, é igualmente provável que o valor seja encontrado no primeiro elemento ou no último. Em média, portanto, o programa terá de comparar a chave de pesquisa com metade dos elementos do *array*.

A Fig. 7.12 contém um *array* de 100 elementos preenchido com os inteiros pares de 0 a 198. O usuário digita a chave de pesquisa em um `JTextField` e pressiona *Enter* para iniciar a pesquisa. [Nota: passamos o *array* para `linearSearch` mesmo que ele seja uma variável de instância da classe. Isso ocorre porque um *array* normalmente é passado para um método de outra classe para classificação. Por exemplo, a classe `Arrays` (veja o Capítulo 21)

contém uma variedade de métodos `static` para classificar *arrays*, pesquisá-los, comparar-lhes o conteúdo e preencher os *arrays* de todos os tipos primitivos, *Objects* e *Strings*.]

```

1 // Fig. 7.12: LinearSearch.java
2 // Pesquisa linear de um array
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class LinearSearch extends JApplet
12     implements ActionListener {
13
14     JLabel enterLabel, resultLabel;
15     JTextField enterField, resultField;
16     int array[];
17
18     // configura a GUI do applet
19     public void init()
20     {
21         // obtém painel de conteúdo e configura seu leiaute para FlowLayout
22         Container container = getContentPane();
23         container.setLayout( new FlowLayout() );
24
25         // configura Jlabel e JTextField para ler dados digitados pelo usuário
26         enterLabel = new JLabel( "Enter integer search key" );
27         container.add( enterLabel );
28
29         enterField = new JTextField( 10 );
30         container.add( enterField );
31
32         // registra este applet como o tratador de eventos de enterField
33         enterField.addActionListener( this );
34
35         // configura Jlabel e JTextField para exibir resultados
36         resultLabel = new JLabel( "Result" );
37         container.add( resultLabel );
38
39         resultField = new JTextField( 20 );
40         resultField.setEditable( false );
41         container.add( resultField );
42
43         // cria array e o preenche com inteiros pares de 0 a 198
44         array = new int[ 100 ];
45
46         for ( int counter = 0; counter < array.length; counter++ )
47             array[ counter ] = 2 * counter;
48
49     } // fim do método init
50
51     // Pesquisa no array o valor de chave especificado
52     public int linearSearch( int array[], int key )
53     {
54         // laço para percorrer elementos do array
55         for ( int counter = 0; counter < array.length; counter++ )

```

Fig. 7.12 Pesquisa linear de um *array* (parte 1 de 2).

```

56          // se elemento do array for igual ao valor da chave, devolve posição
57          if ( array[ counter ] == key )
58              return counter;
59
60          return -1;    // chave não-encontrada
61      }
62
63      // obtém dados digitados pelo usuário e chama o método linearSearch
64      public void actionPerformed( ActionEvent actionEvent )
65      {
66          // dados de entrada também podem ser obtidos com enterFiled.getText()
67          String searchKey = actionEvent.getActionCommand();
68
69          // Array é passado para linearSearch mesmo que ele
70          // seja uma variável de instância. Normalmente um array
71          // é passado a um método para fins de pesquisa.
72          int element =
73              linearSearch( array, Integer.parseInt( searchKey ) );
74
75          // exibe resultado da pesquisa
76          if ( element != -1 )
77              resultField.setText( "Found value in element " +
78                  element );
79          else
80              resultField.setText( "Value not found" );
81      }
82
83  } // fim da classe linearSearch

```

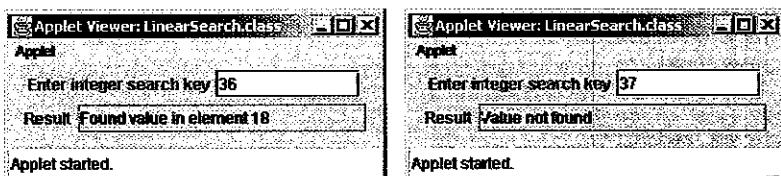


Fig. 7.12 Pesquisa linear de um array (parte 2 de 2).

O método de pesquisa linear funciona bem para arrays pequenos ou não-classificados. Entretanto, para arrays grandes a pesquisa linear é ineficiente. Se o array estiver ordenado, pode-se usar a técnica de alta velocidade denominada *pesquisa binária*, apresentada na próxima seção.

7.8.2 Pesquisando um array com pesquisa binária

O algoritmo de pesquisa binária elimina metade dos elementos no *array* que está sendo pesquisado a cada comparação. O algoritmo localiza o elemento do meio do *array* e o compara com a chave de pesquisa. Se forem iguais, a chave de pesquisa foi localizada e a pesquisa binária retorna o subscrito desse elemento. Caso contrário, a pesquisa binária reduz o problema ao pesquisar metade do *array*. Se a chave de pesquisa for menor que o elemento do meio do *array*, a primeira metade do *array* será pesquisada; caso contrário, a segunda metade do *array* seria pesquisada. Se a chave de pesquisa não for o elemento do meio no *subarray* (um pedaço do *array* original) especificado, o algoritmo será repetido para um quarto do *array* original. A pesquisa continua até que a chave de pesquisa seja igual ao elemento do meio de um *subarray* ou até que o *subarray* consista em apenas um elemento que não é igual à chave de pesquisa (isto é, a chave de pesquisa não foi localizada).

No pior caso, pesquisar um *array* de 1024 elementos exigirá apenas 10 comparações com uma pesquisa binária. Dividindo 1024 repetidamente por 2 (porque após cada comparação seremos capazes de eliminar metade do *ar-*

ray), obtemos os valores 512, 256, 128, 64, 32, 16, 8, 4, 2 e 1. O número 1024 (2^{10}) é dividido por 2 somente 10 vezes para obter o valor 1. Dividir por 2 é equivalente a uma comparação no algoritmo de pesquisa binária. Um *array* de 1.048.576 (2^{20}) elementos exige um máximo de 20 comparações para localizar a chave. O *array* de um bilhão de elementos exige um máximo de 30 comparações para localizar a chave. Essa é uma enorme melhora no desempenho em relação à pesquisa linear que exigia comparação da chave de pesquisa com uma média da metade dos elementos no *array*. Para um *array* de um bilhão de elementos, esta é uma diferença entre uma média de 500 milhões de comparações e um máximo de 30 comparações! O número máximo de comparações necessárias para a pesquisa binária de qualquer *array* ordenado é o expoente da primeira potência de 2 maior que o número de elementos no *array*.

A Fig. 7.13 apresenta a versão iterativa do método **binarySearch** (linhas 85 a 116). O método recebe dois argumentos – um *array* de inteiros denominado **array2** (o *array* a pesquisar) e um inteiro **key** (a chave de pesquisa). O programa passa o *array* para **binarySearch** mesmo que seja uma variável de instância da classe. Mais uma vez, isso ocorre porque um *array* é normalmente passado para um método de outra classe para classificar. Se **key** for igual ao elemento **middle** de um *subarray*, **binarySearch** devolve **middle** (o subscrito do elemento atual) para indicar que o valor foi encontrado e a pesquisa está completa. Se **key** não for igual ao elemento **middle** de um *subarray*, **binarySearch** ajusta o subscrito **low** ou o subscrito **high** (ambos declarados no método), para continuar a pesquisa com um *subarray* menor. Se **key** for menor que o elemento intermediário, o subscrito **high** é configurado como **middle - 1** e a pesquisa prossegue nos elementos de **low** a **middle - 1**. Se **key** for maior que o elemento intermediário, o subscrito **low** é configurado como **middle + 1** e a pesquisa prossegue nos elementos de **middle + 1** a **high**. O método **binarySearch** faz essas comparações na estrutura aninhada **if/else** nas linhas 102 a 111.

```

1 // Fig. 7.13: BinarySearch.java
2 // Pesquisa binária de um array
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.text.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class BinarySearch extends JApplet
13     implements ActionListener {
14
15     JLabel enterLabel, resultLabel;
16     JTextField enterField, resultField;
17     JTextArea output;
18
19     int array[];
20     String display = "";
21
22     // configura a GUI do applet
23     public void init()
24     {
25         // obtém o painel de conteúdo e configura seu layout para FlowLayout
26         Container container = getContentPane();
27         container.setLayout( new FlowLayout() );
28
29         // configura JLabel e JTextField para entrada de dados do usuário
30         enterLabel = new JLabel( "Enter integer search key" );
31         container.add( enterLabel );
32
33         enterField = new JTextField( 10 );
34         container.add( enterField );

```

Fig. 7.13 Pesquisa binária de um *array* ordenado (parte 1 de 4).

```

35      // registra este applet como tratador de eventos de enterField
36      enterField.addActionListener( this );
37
38
39      // configura JLabel e JTextField para exibir resultados
40      resultLabel = new JLabel( "Result" );
41      container.add( resultLabel );
42
43      resultField = new JTextField( 20 );
44      resultField.setEditable( false );
45      container.add( resultField );
46
47      // configura JTextArea para exibir dados da comparação
48      output = new JTextArea( 6, 60 );
49      output.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
50      container.add( output );
51
52      // cria array e preenche com inteiros pares de 0 a 28
53      array = new int[ 15 ];
54
55      for ( int counter = 0; counter < array.length; counter++ )
56          array[ counter ] = 2 * counter;
57
58 } // fim do método init
59
60 // obtém dados digitados pelo usuário e chama o método binarySearch
61 public void actionPerformed( ActionEvent actionEvent )
62 {
63     // dados de entrada também podem ser obtidos com enterField.getText()
64     String searchKey = actionEvent.getActionCommand();
65
66     // inicializa o string display para a nova pesquisa
67     display = "Portions of array searched\n";
68
69     // realiza a pesquisa binária
70     int element =
71         binarySearch( array, Integer.parseInt( searchKey ) );
72
73     output.setText( display );
74
75     // exibe resultado da pesquisa
76     if ( element != -1 )
77         resultField.setText(
78             "Found value in element " + element );
79     else
80         resultField.setText( "Value not found" );
81
82 } // fim do método actionPerformed
83
84 // método para fazer pesquisa binária em um array
85 public int binarySearch( int array2[], int key )
86 {
87     int low = 0;                      // subscrito baixo
88     int high = array2.length - 1;      // subscrito alto
89     int middle;                     // subscrito intermediário
90
91     // repete até que subscrito low seja maior do que subscrito high
92     while ( low <= high ) {
93

```

Fig. 7.13 Pesquisa binária de um array ordenado (parte 2 de 4).

```

94     // determina o subscrito do elemento do meio
95     middle = ( low + high ) / 2;
96
97     // exibe o subconjunto do array que está sendo usado
98     // durante esta iteração do laço de pesquisa binária
99     buildOutput( array2, low, middle, high );
100
101    // se a chave for igual ao elemento do meio, devolve posição do meio
102    if ( key == array2[ middle ] )
103        return middle;
104
105    // se a chave é menor do que o elemento do meio, ajusta high
106    else if ( key < array2[ middle ] )
107        high = middle - 1;
108
109    // chave maior do que o elemento do meio, ajusta low
110    else
111        low = middle + 1;
112    }
113
114    return -1; // chave não-encontrada
115
116 } // fim do método binarySearch
117
118 // Constrói uma linha de saída mostrando o subconjunto
119 // do array que está sendo processado atualmente
120 void buildOutput( int array3[],
121     int low, int mid, int high )
122 {
123     // cria um formato para números inteiros de 2 dígitos
124     DecimalFormat twoDigits = new DecimalFormat( "00" );
125
126     // percorre os elementos do array
127     for ( int counter = 0; counter < array3.length;
128         counter++ ) {
129
130         // se counter está fora do subconjunto atual do array,
131         // acrescenta espaços de enchimento ao String display
132         if ( counter < low || counter > high )
133             display += "   ";
134
135         // se elemento do meio, acrescenta o elemento ao String display
136         // seguido por um asterisco (*) para indicar o elemento do meio
137         else if ( counter == mid )
138             display +=
139                 twoDigits.format( array3[ counter ] ) + "* ";
140
141         // acrescenta elemento ao String display
142         else
143             display +=
144                 twoDigits.format( array3[ counter ] ) + "   ";
145
146     } // fim da estrutura for
147
148     display += "\n";
149
150 } // fim do método buildOutput
151
152 } // fim da classe binarySearch

```

Fig. 7.13 Pesquisa binária de um array ordenado (parte 3 de 4).

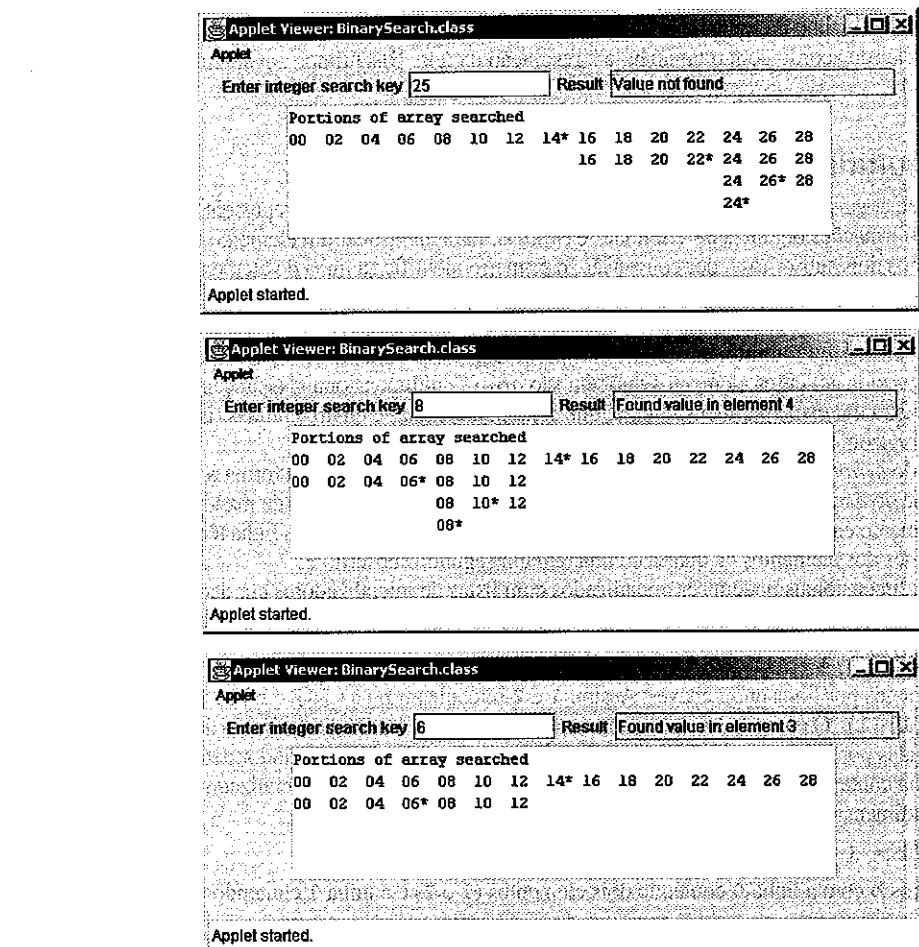


Fig. 7.13 Pesquisa binária de um array ordenado (parte 4 de 4).

O programa utiliza um *array* de 15 elementos. A primeira potência de 2 maior que o número de elementos do *array* é 16 (2^4) – portanto, `binarySearch` exige no máximo quatro comparações para localizar `key`. Para ilustrar isso, a linha 99 do método `binarySearch` chama o método `buildOutput` (definido nas linhas 120 a 150) para dar saída a cada *subarray* durante o processo de pesquisa binária. O método `buildOutput` marca elemento do meio em cada *subarray* com um asterisco (*) para indicar o elemento com o qual se compara `key`. Cada pesquisa nesse exemplo resulta em um máximo de quatro linhas de saída – uma por comparação.

`JTextArea output` utiliza *Monospaced* (uma *fonte de largura fixa* – isso é, todos os caracteres são da mesma largura) para ajudar a alinhar o texto exibido em cada linha de saída. A linha 49 usa o método `setFont` para alterar a fonte dos caracteres exibidos em `output`. O método `setFont` pode alterar a fonte de texto exibido na maioria dos componentes GUI. O método exige um objeto `Font` (pacote `java.awt`) como seu argumento. O objeto `Font` é inicializado com três argumentos – um `String` que especifica o nome da fonte ("Monospaced"), um `int` que representa o estilo da fonte (`Font.PLAIN` é uma constante inteira definida na classe `Font` que indica fonte normal) e um `int` que representa o tamanho da fonte em pontos (12). Java fornece nomes genéricos para várias fontes disponíveis em todas as plataformas Java. A fonte *Monospaced* também é chamada de *Courier*. Outras fontes comuns incluem *Serif* (também chamada de *TimesRoman*) e *SansSerif* (também chamada de *Helvetica*). Java 2, na verdade, fornece acesso a todas as fontes em seu sistema através de métodos da classe `GraphicsEnvironment`. O estilo também pode ser `Font.BOLD`, `Font.ITALIC` ou `Font.BOLD + Font.ITALIC`. O tamanho

em pontos representa o tamanho da fonte – há 72 pontos em uma polegada. O tamanho real do texto como ele aparece na tela pode variar de acordo com o tamanho e a resolução da tela. Discutimos novamente a manipulação de fontes no Capítulo 11.

7.9 Arrays multidimensionais

Os *arrays multidimensionais* com dois subscritos são utilizados freqüentemente para representar *tabelas* de valores que consistem em informações organizadas em linhas e *colunas*. Para identificar um elemento específico da tabela, devemos especificar os dois subscritos – por convenção, o primeiro identifica a linha do elemento e o segundo identifica a coluna do elemento. Os *arrays* que exigem dois subscritos para identificar um elemento específico são chamados de *arrays bidimensionais*. Observe que os *arrays multidimensionais* podem ter mais de dois subscritos. Java não suporta diretamente *arrays multidimensionais*, mas permite que o programador especifique *arrays* de um único subscrito (unidimensionais) cujos elementos também são *arrays unidimensionais*, alcançando, assim, o mesmo efeito. A Fig. 7.14 ilustra um *array bidimensional*, **a**, que contém três linhas e quatro colunas (isto é, um *array 3 por 4*). Em geral, um *array* com *m* linhas e *n* colunas chama-se *array m por n*.

Cada elemento do *array a* é identificado na Fig. 7.14 por um nome de elemento na forma **a [linha] [coluna]**; **a** é o nome do *array* e **linha** e **coluna** são os subscritos que identificam de maneira unívoca a linha e a coluna de cada elemento em **a**. Repare que todos os nomes dos elementos na primeira linha têm um primeiro subscrito 0; todos os nomes dos elementos na quarta coluna têm um segundo subscrito 3.

Os *arrays multidimensionais* podem ser inicializados com listas de inicializadores em declarações, da mesma forma que um *array unidimensional*. O *array bidimensional b [2] [2]* poderia ser declarado e inicializado com

```
int b[][] = { { 1, 2 }, { 3, 4 } };
```

Os valores são agrupados por linha entre chaves. Assim, 1 e 2 inicializam **b [0] [0]** e **b [0] [1]** e 3 e 4 inicializam **b [1] [0]** e **b [1] [1]**. O compilador determina o número de linhas contando o número de sublistas de inicializadores (representadas por conjuntos de chaves) na lista de inicializadores. O compilador determina o número de colunas em cada linha contando o número de valores inicializadores na sublista de inicializadores dessa linha.

Os *arrays multidimensionais* são mantidos como *arrays de arrays*. A declaração

```
int b[][] = { { 1, 2 }, { 3, 4, 5 } };
```

cria o *array* de inteiros **b** com a linha 0 contendo dois elementos (1 e 2) e a linha 1 contendo três elementos (3, 4 e 5).

Um *array multidimensional* com o mesmo número de colunas em cada linha pode ser alocado dinamicamente. Por exemplo, um *array 3 por 4* é alocado como segue:

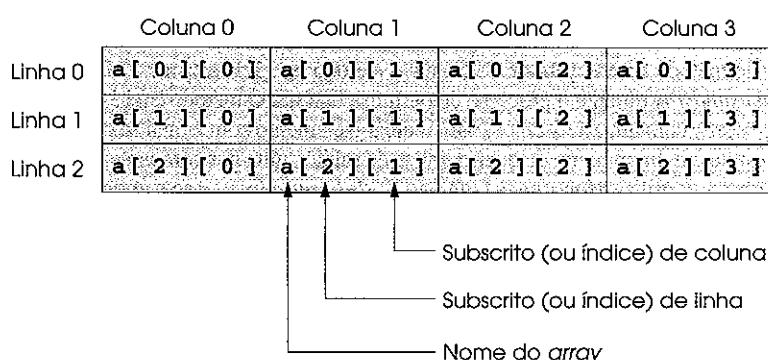


Fig. 7.14 Um *array bidimensional* com três linhas e quatro colunas.

```
int b[][];
b = new int[ 3 ][ 4 ];
```

Nesse caso, usamos os valores literais 3 e 4 para especificar o número de linhas e o número de colunas, respectivamente. Observe que os programas também podem usar variáveis para especificar as dimensões dos *arrays*. Assim como ocorre com *arrays* unidimensionais, os elementos de um *array* bidimensional são inicializados quando **new** cria o objeto *array*.

O *array* multidimensional em que cada linha tem um número diferente de colunas pode ser alocado dinamicamente como segue:

```
int b[][];
b = new int[ 2 ][ 1 ]; // aloca linhas
b[ 0 ] = new int[ 5 ]; // aloca colunas para a linha 0
b[ 1 ] = new int[ 3 ]; // aloca colunas para a linha 1
```

O código precedente cria um *array* bidimensional com duas linhas. A linha 0 tem cinco colunas e a linha 1 tem três colunas.

O *applet* da Fig. 7.15 demonstra a inicialização de *arrays* bidimensionais em declarações e o uso de laços aninhados **for** para percorrer os *arrays* (isto é, manipular todos os elementos do *array*).

```

1 // Fig. 7.15: InitArray.java
2 // Inicialização de arrays multidimensionais
3
4 // Pacotes do núcleo de Java
5 import java.awt.Container;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class InitArray extends JApplet {
11     JTextArea outputArea;
12
13     // configura a GUI e inicializa o applet
14     public void init()
15     {
16         outputArea = new JTextArea();
17         Container container = getContentPane();
18         container.add( outputArea );
19
20         int array1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
21         int array2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
22
23         outputArea.setText( "Values in array1 by row are\n" );
24         buildOutput( array1 );
25
26         outputArea.append( "\nValues in array2 by row are\n" );
27         buildOutput( array2 );
28     }
29
30     // acrescenta linhas e colunas de um array a outputArea
31     public void buildOutput( int array[][] )
32     {
33         // percorre as linhas do array com um laço
34         for ( int row = 0; row < array.length; row++ ) {
35
36             // percorre as colunas da linha atual com um laço
37             for ( int column = 0;
38                 column < array[ row ].length;
39                 column++ )
40                 outputArea.append( array[ row ][ column ] + " " );
```

Fig. 7.15 Inicializando *arrays* multidimensionais (parte 1 de 2).

```

41
42     outputArea.append( "\n" );
43 }
44 }
45 }
```

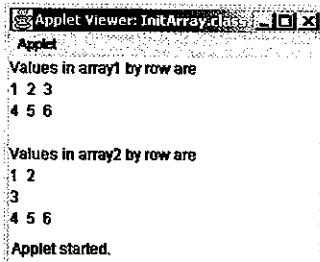


Fig. 7.15 Inicializando arrays multidimensionais (parte 2 de 2).

O programa declara dois *arrays* no método `init`. A declaração `array1` (linha 20) fornece seis inicializadores em duas sublistas. A primeira sublistas inicializa a primeira linha do *array* com os valores 1, 2 e 3; e a segunda sublistas inicializa a segunda linha do *array* com os valores 4, 5 e 6. A declaração de `array2` (linha 21) fornece seis inicializadores em três sublistas. A sublistas para a primeira linha inicializa explicitamente a primeira linha para ter dois elementos com os valores 1 e 2, respectivamente. A sublistas para a segunda linha inicializa a segunda linha para ter um elemento com o valor 3. A sublistas para a terceira linha inicializa a terceira linha com os valores 4, 5 e 6.

A linha 24 do método `init` chama o método `buildOutput` (definido nas linhas 31 a 44) para acrescentar cada elemento do *array* a `outputArea` (uma `JTextArea`). O método `buildOutput` especifica o parâmetro de *array* como `int array[] []` para indicar que o método recebe como um argumento um *array* bidimensional. Repare o uso de uma estrutura aninhada `for` (linhas 34 a 43) para dar saída às linhas de um *array* bidimensional. Na estrutura externa `for`, a expressão `array.length` determina o número de linhas no *array*. Na estrutura interna `for`, a expressão `array[row].length` determina o número de colunas na linha corrente do *array*. Essa condição permite que o laço determine o número exato de colunas em cada linha.

Muitas operações comuns sobre *arrays* utilizam estruturas de repetição `for`. Por exemplo, a seguinte estrutura `for` atribui a todos os elementos da terceira linha do *array* `a` da Fig. 7.14 o valor zero:

```

for ( int column = 0; column < a[ 2 ].length; column++ )
    a[ 2 ][ column ] = 0;
```

Especificamos a *terceira* linha; portanto, sabemos que o primeiro subscrito é sempre 2 (0 é a primeira linha e 1 é a segunda linha). O laço `for` varia apenas o segundo subscrito (isto é, o subscrito de coluna). A estrutura `for` precedente é equivalente às instruções de atribuição

```

a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

A estrutura aninhada `for` a seguir determina o total de todos os elementos no *array* `a`.

```

int total = 0;

for ( int row = 0; row < a.length; row++ )

    for ( int column = 0; column < a[ row ].length; column++ )

        total += a[ row ][ column ];
```

A estrutura `for` soma os elementos do *array* uma linha por vez. A estrutura externa `for` inicia configurando o subscrito `row` como 0 para que os elementos da primeira linha possam ser somados pela estrutura interna `for`. A estrutura externa `for` então incrementa `row` para 1, assim a segunda linha pode ser somada. Depois, a estrutura externa

for incrementa **row** para 2, assim a terceira linha pode ser somada. O resultado pode ser exibido quando a estrutura aninhada **for** termina.

O *applet* da Fig. 7.16 realiza várias outras operações comuns sobre o *array* 3 por 4 **grades**. Cada linha do *array* representa um aluno e cada coluna representa uma nota obtida em um dos quatro exames que os alunos fizeram durante o semestre. As operações sobre o *array* são realizadas por quatro métodos. O método **minimum** (linhas 52 a 69) determina a nota mais baixa de qualquer aluno no semestre. O método **maximum** (linhas 72 a 89) determina a nota mais alta de qualquer aluno no semestre. O método **average** (linhas 93 a 103) determina a média do semestre de um aluno em particular. O método **buildString** (linhas 106 a 121) acrescenta o *array* bidimensional ao **String output** em formato de tabela.

Os métodos **minimum**, **maximum** e **buildString** utilizam o *array* **grades** e as variáveis **students** (número de linhas no *array*) e **exams** (número de colunas no *array*). Cada método percorre o *array* **grades** utilizando estruturas aninhadas **for** – por exemplo, a estrutura aninhada **for** na definição do método **minimum** (linhas 58 a 66). A estrutura externa **for** inicializa **row** (o subscrito de linha) como 0, de modo que os elementos da primeira linha possam ser comparados com a variável **lowGrade** no corpo da estrutura interna **for**. A estrutura interna **for** percorre com um laço as quatro notas de uma linha particular e compara cada nota com **lowGrade**. Se uma nota for menor que **lowGrade**, essa nota é atribuída a **lowGrade**. A estrutura externa **for** então incrementa o subscrito de linha por 1. Os elementos da segunda linha são comparados com a variável **lowGrade**. A estrutura externa **for** então incrementa o subscrito da linha para 2. Os elementos da terceira linha são comparados com a variável **lowGrade**. Quando a execução da estrutura aninhada estiver completa, **lowGrade** conterá a menor nota no *array* bidimensional. O método **maximum** funciona de maneira semelhante ao método **minimum**.

```

1 // Fig. 7.16: DoubleArray.java
2 // Exemplo de array de subscrito duplo
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class DoubleArray extends JApplet {
11     int grades[][] = { { 77, 68, 86, 73 },
12                         { 96, 87, 89, 81 },
13                         { 70, 90, 86, 81 } };
14
15     int students, exams;
16     String output;
17     JTextArea outputArea;
18
19     // inicializa as variáveis de instância
20     public void init()
21     {
22         students = grades.length;      // número de alunos
23         exams = grades[ 0 ].length;   // número de exames
24
25         // cria JTextArea e anexa ao applet
26         outputArea = new JTextArea();
27         Container container = getContentPane();
28         container.add( outputArea );
29
30         // constrói o string output
31         output = "The array is:\n";
32         buildString();
33
34         // chama os métodos minimum e maximum
35         output += "\n\nLowest grade: " + minimum() +
36                   "\nHighest grade: " + maximum() + "\n";

```

Fig. 7.16 Exemplo de utilização de arrays bidimensionais (parte 1 de 3).

```

37
38     // chama o método average para calcular a média de cada aluno
39     for ( int counter = 0; counter < students; counter++ )
40         output += "\nAverage for student " + counter + " is " +
41             average( grades[ counter ] );
42
43     // muda a fonte de exibição na outputArea
44     outputArea.setFont(
45         new Font( "Courier", Font.PLAIN, 12 ) );
46
47     // coloca o string output em outputArea
48     outputArea.setText( output );
49 }
50
51 // localiza a nota mínima
52 public int minimum()
53 {
54     // supõe que o primeiro elemento de grades é o menor
55     int lowGrade = grades[ 0 ] [ 0 ];
56
57     // percorre com um laço as linhas do array grades
58     for ( int row = 0; row < students; row++ )
59
60         // percorre com um laço as colunas da linha atual
61         for ( int column = 0; column < exams; column++ )
62
63             // testa se a nota atual é menor que lowGrade
64             // se é, atribui a nota atual a lowGrade
65             if ( grades[ row ][ column ] < lowGrade )
66                 lowGrade = grades[ row ][ column ];
67
68     return lowGrade;    // retorna nota mais baixa
69 }
70
71 // localiza a nota máxima
72 public int maximum()
73 {
74     // supõe que o primeiro elemento do array grades é o maior
75     int highGrade = grades[ 0 ] [ 0 ];
76
77     // percorre com um laço as linhas do array grades
78     for ( int row = 0; row < students; row++ )
79
80         // percorre com um laço as colunas da linha atual
81         for ( int column = 0; column < exams; column++ )
82
83             // testa se a nota atual é maior que highGrade
84             // se for, atribui a nota atual a highGrade
85             if ( grades[ row ][ column ] > highGrade )
86                 highGrade = grades[ row ][ column ];
87
88     return highGrade;    // devolve a nota mais alta
89 }
90
91 // determina a nota média para um aluno
92 // (ou conjunto de notas) específico
93 public double average( int setOfGrades[] )
94 {
95     int total = 0;    // inicializa total
96

```

Fig. 7.16 Exemplo de utilização de arrays bidimensionais (parte 2 de 3).

```

97      // soma as notas para um aluno
98      for ( int count = 0; count < setOfGrades.length; count++ )
99          total += setOfGrades[ count ];
100
101     // retorna a média das notas
102     return ( double ) total / setOfGrades.length;
103 }
104
105    // constrói o string output
106    public void buildString()
107    {
108        output += "           "; // usado para alinhar os títulos de coluna
109
110        // cria cabeçalhos de colunas
111        for ( int counter = 0; counter < exams; counter++ )
112            output += "[" + counter + "]   ";
113
114        // cria linhas/colunas de texto que representam as notas do array
115        for ( int row = 0; row < students; row++ ) {
116            output += "\ngrades[" + row + "]   ";
117
118            for ( int column = 0; column < exams; column++ )
119                output += grades[ row ][ column ] + "   ";
120        }
121    }
122 }

```

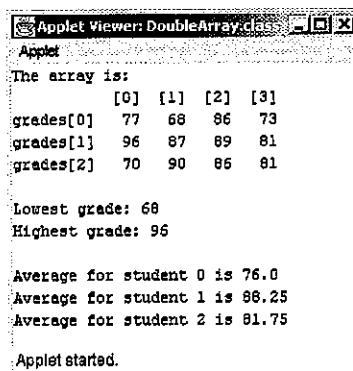


Fig. 7.16 Exemplo de utilização de arrays bidimensionais (parte 3 de 3).

O método `average` recebe um argumento – um *array unidimensional* de resultados de exames para um aluno em particular. Quando a linha 41 chama `average`, o argumento é `grades[counter]`, que especifica que uma linha particular do *array bidimensional* `grades` deve ser passada para `average`. Por exemplo, o argumento `grades[1]` representa os quatro valores (um *array unidimensional* de notas) armazenados na segunda linha do *array bidimensional* `grades`. Lembre-se de que, em Java, um *array bidimensional* é um *array* com elementos que são *arrays unidimensionais*. O método `average` calcula a soma dos elementos do *array*, divide o total pelo número de resultados de exames e retorna o resultado em ponto flutuante como um valor `double`.

7.10 (Estudo de caso opcional) Pensando em objetos: colaboração entre objetos

Nesta seção, concentrar-nos nas colaborações (interações) entre objetos. Quando dois objetos se comunicam um com o outro, para executar determinada tarefa, diz-se que eles *colaboram* – os objetos fazem isto invocando uns as operações dos outros. Dizemos que os objetos enviam mensagens a outros objetos. Explicamos como as mensagens

são enviadas e recebidas em Java na Seção 10.22. A *colaboração* consiste em um objeto de uma classe enviando uma mensagem particular para um objeto de uma outra classe.

A mensagem enviada pelo primeiro objeto invoca uma operação do segundo objeto. Na Seção 6.17 de “Pensando em objetos”, determinamos muitas das operações das classes em nosso sistema. Nesta seção, concentrarmo-nos nas mensagens que invocam estas operações. A Fig. 7.17 é a tabela de classes e frases com verbos da Seção 6.17. Removemos todas as frases com verbos das classes **Elevator** e **Person** que não correspondem a operações. As frases restantes são nossa primeira estimativa das colaborações em nosso sistema. À medida que prosseguimos nessa seção e nas seções “Pensando em objetos” restantes, descobriremos colaborações adicionais.

Examinamos a lista de frases com verbos para determinar as colaborações em nosso sistema. Por exemplo, a classe **Elevator** lista a frase “desliga o botão do elevador”. Para executar esta tarefa, um objeto da classe **Elevator** envia a mensagem **resetButton** para um objeto da classe **ElevatorButton** (invocando a operação **resetButton** de **ElevatorButton**). A Fig. 7.18 lista todas as colaborações que podem ser vislumbradas em nossa tabela de frases com verbos. De acordo com a Fig. 7.17, o **Elevator**¹ toca a **Bell** e abre (e fecha) a **ElevatorDoor**, de modo que incluímos as mensagens **ringBell**, **openDoor** e **closeDoor** na Fig. 7.18. Entretanto, precisamos pensar em como a **FloorDoor** abre e fecha. De acordo com o diagrama de classes da Fig. 3.23, **ElevatorShaft** é a única classe que se associa com **FloorDoor**. O **Elevator** sinaliza a chegada (supomos que para o **ElevatorShaft**) enviando uma mensagem **elevatorArrives**. O **ElevatorShaft** responde a esta mensagem desligando o **FloorButton** apropriado e ligando a **Light** apropriada. O **ElevatorShaft** envia as mensagens **resetButton** e **turnOnLight**. Neste ponto do projeto, podemos supor que o **Elevator** também sinaliza sua partida – i. e., o **ElevatorShaft** envia uma mensagem **elevatorDeparted** para o **ElevatorShaft**, que então desliga a **Light** apropriada enviando-lhe uma mensagem **turnOffLight**.

Classe	Frases com verbos
Elevator	desliga o botão do elevador, soa a campainha do elevador,
	sinaliza sua chegada, sinaliza sua partida, abre a porta, fecha a porta
ElevatorShaft	desliga a luz, liga a luz, desliga o botão do andar
Person	aperta o botão do andar, aperta o botão do elevador, anda no elevador,
	entra no elevador, sai do elevador
FloorButton	chama o elevador
ElevatorButton	avisa o elevador para se mover para o outro andar
FloorDoor	avisa a pessoa para entrar no elevador (se abrindo)
ElevatorDoor	avisa a pessoa para sair do elevador (se abrindo),
	abre a porta do andar, fecha a porta do andar
ElevatorModel	cria pessoa

Fig. 7.17 Frases com verbos para cada classe, mostrando comportamentos na simulação.

A **Person** pode pressionar **FloorButton** e **ElevatorButton**. O objeto **Person** pode entrar no **Elevator** e sair dele. Portanto, a **Person** pode enviar mensagens **pressButton**, **enterElevator** e **exitElevator**. O **FloorButton** chama o **Elevator**, de modo que o **FloorButton** pode enviar uma mensagem **requestElevator**. O **ElevatorButton** sinaliza para o **Elevator** que comece a se mover para o outro andar, de modo que o **ElevatorButton** pode enviar uma mensagem **moveElevator**.

¹ Referimo-nos a um objeto usando o nome da classe daquele objeto precedido por um artigo (“um”, “uma”, “o” ou “a”) – por exemplo, o **Elevator** se refere ao objeto da classe **Elevator**. Nossa sintaxe evita a redundância – i.e., evitamos repetir a frase “um objeto da classe ...”.

Um objeto da classe...	Envia a mensagem...	Para um objeto da classe...
Elevator	resetButton ringBell elevatorArrived elevatorDeparted openDoor closeDoor	ElevatorButton Bell ElevatorShaft ElevatorShaft ElevatorDoor ElevatorDoor
ElevatorShaft	resetButton turnOnLight turnOffLight	FloorButton Light Light
Person	pressButton enterElevator exitElevator	FloorButton, ElevatorButton Elevator Elevator
FloorButton	requestElevator	Elevator
ElevatorButton	moveElevator	Elevator
FloorDoor	doorOpened doorClosed	Person Person
ElevatorDoor	doorOpened doorClosed openDoor closeDoor	Person Person FloorDoor FloorDoor

Fig. 7.18 Colaborações no sistema do elevador.

A **FloorDoor** e a **ElevatorDoor** informam a uma **Person** que elas abriram ou fecharam, de modo que os dois objetos enviam as mensagens **doorOpen** e **doorClose**². Finalmente, o **ElevatorModel** cria uma **Person**, mas o **ElevatorModel** não pode enviar uma mensagem **personCreated**, porque aquela pessoa ainda não existe. Descobrimos, na Seção 8.6, como usar um método especial que cria, ou instancia, novos objetos. Este método chama-se *construtor* – o **ElevatorModel** vai criar uma **Person** chamando o construtor daquela **Person** (o que é semelhante a enviar a mensagem **personCreated**).

Por último, a **ElevatorDoor** precisa enviar as mensagens **openDoor** e **closeDoor** para uma **FloorDoor**, para garantir que estas portas abram e fechem juntas.

Diagramas de colaborações

Consideremos, agora, os objetos que precisam interagir para que as pessoas em nossa simulação possam entrar e sair do elevador quando ele chega em um andar. A UML oferece o *diagrama de colaborações* para modelar tais interações. Os diagramas de colaborações são um tipo de *diagrama de interações*. Eles modelam aspectos comportamentais do sistema, fornecendo informações sobre como os objetos interagem. Os diagramas de colaborações destacam quais objetos participam das interações. O outro tipo de diagrama de interação é o *diagrama de seqüência*, que apresentamos no Capítulo 15. A Fig. 7.19 mostra um diagrama de colaborações que modela uma pessoa que está pressionando um botão de andar. Um objeto em um diagrama de colaboração é representado como um retângulo que

² Note que a maioria das mensagens executa alguma ação específica sobre o objeto que a recebe; por exemplo, o **Elevator** desliga o **ElevatorButton**. Entretanto, outras mensagens informam aos objetos que as recebem sobre *eventos* que já aconteceram; por exemplo, a **FloorDoor** informa à **Person** que a **FloorDoor** abriu. Na Seção 10.22 aprofundamos este tópico sobre eventos – por enquanto, no entanto, continuamos como se os dois tipos de mensagens não sejam distintos.

contém o nome do objeto. Escrevemos os nomes de objetos no diagrama de colaborações usando a convenção que apresentamos no diagrama de objetos da Fig. 3.24 – os objetos são escritos na forma **nomeDoObjeto:NomeDaClasse**. Neste exemplo, omitimos o nome do objeto porque estamos interessados somente no tipo do objeto. Os objetos que colaboram entre si são conectados por linhas cheias e as mensagens são passadas entre os objetos, ao longo destas linhas, na direção mostrada pelas setas. O nome da mensagem que aparece próximo à seta é o nome de um método que pertence ao objeto receptor – pense no nome como um “serviço” que o objeto receptor (um “servidor”) fornece para seus objetos solicitantes (seus “clientes”).



Fig. 7.19 Diagrama de colaborações para uma pessoa pressionando o botão de andar.

A seta na Fig. 7.19 representa uma mensagem em UML e um método – ou chamada síncrona – em Java. Esta seta indica que o fluxo de controle parte do objeto emissor em direção ao objeto receptor – e o objeto emissor não pode enviar outra mensagem até que o objeto receptor processe a mensagem e devolva o controle ao objeto emissor. Por exemplo, na Fig. 7.19, a pessoa chama o método **pressButton** de um **FloorButton** e não pode enviar outra mensagem para um objeto até que **pressButton** tenha terminado e devolvido o controle para aquela pessoa. Se o nosso programa contiver um objeto **FloorButton** denominado **firstFloorButton** e supormos que a pessoa gerencie o fluxo de controle, o código Java implementado na classe **Person** que representa este diagrama de colaboração é

```
firstFloorButton.pressButton();
```

A Fig. 7.20 mostra um diagrama de colaboração que modela a interação entre objetos do sistema enquanto os objetos da classe **Person** entram no elevador e saem dele. A colaboração inicia quando o **Elevator** chega em um **Floor**. O número à esquerda do nome da mensagem indica a ordem na qual a mensagem é passada. A *seqüência de mensagens* em um diagrama de colaborações progride, em ordem numérica, da menor para a maior. Neste diagrama, a numeração começa com a mensagem 1 e termina com a mensagem 4 . 2. A seqüência de passagem de mensagens segue uma estrutura aninhada – por exemplo, a mensagem 1 . 1 é a *primeira mensagem* aninhada na mensagem 1, e a mensagem 3 . 2 é a *segunda mensagem* aninhada na mensagem 3. A mensagem 3 . 2 . 1 poderia ser a *primeira mensagem* aninhada na mensagem 3 . 2. A mensagem pode ser passada somente quando todas as mensagens aninhadas da mensagem anterior tenham sido passadas. Por exemplo, na Fig. 7.20, o **Elevator** passa a mensagem 4 após as mensagens 3, 3 . 1, 3 . 1 . 1, 3 . 1 . 1 . 1, 3 . 2 e 3 . 2 . 1 terem sido passadas.

O **Elevator** envia a mensagem **resetButton** (mensagem 1) para o **ElevatorButton**, para desligar o **ElevatorButton**. O **Elevator** envia a mensagem **ringBell** (mensagem 2) para a **Bell**. Depois, abre a **ElevatorDoor**, passando a mensagem **openDoor** (mensagem 3). A **ElevatorDoor** abre a **FloorDoor** enviando uma mensagem **openDoor** (mensagem 3 . 1 no topo do diagrama) para aquela **FloorDoor**. A **FloorDoor** informa ao **waitingPassenger** que a **FloorDoor** abriu (mensagem 3 . 1 . 1) e o **waitingPassenger** entra no **Elevator** (mensagem 3 . 1 . 1 . 1). A **ElevatorDoor** informa ao **ridingPassenger** que a **ElevatorDoor** abriu (mensagem 3 . 2), de modo que o **ridingPassenger** pode sair do **Elevator** (mensagem 3 . 2 . 1). Finalmente, o **Elevator** informa ao **ElevatorShaft** da chegada (mensagem 4), de modo que o **ElevatorShaft** pode desligar o **FloorButton** (mensagem 4 . 1) e ligar a luz (mensagem 4 . 2).

Infelizmente, este projeto cria um problema. De acordo com o diagrama, o **waitingPassenger** entra no elevador (mensagem 3 . 1 . 1 . 1) antes que o **ridingPassenger** (mensagem 3 . 2 . 1) saia. Na Seção 15.12 de “Pensando em objetos” usamos *multithreading*, sincronização e classes ativas no nosso diagrama de colaborações para forçar o **waitingPassenger** a aguardar a saída do **ridingPassenger** do **Elevator**. Antes de corrigir este problema, modificamos este diagrama para indicar mais precisamente a passagem de mensagens na seção 10.22, quando discutimos tratamento de eventos.

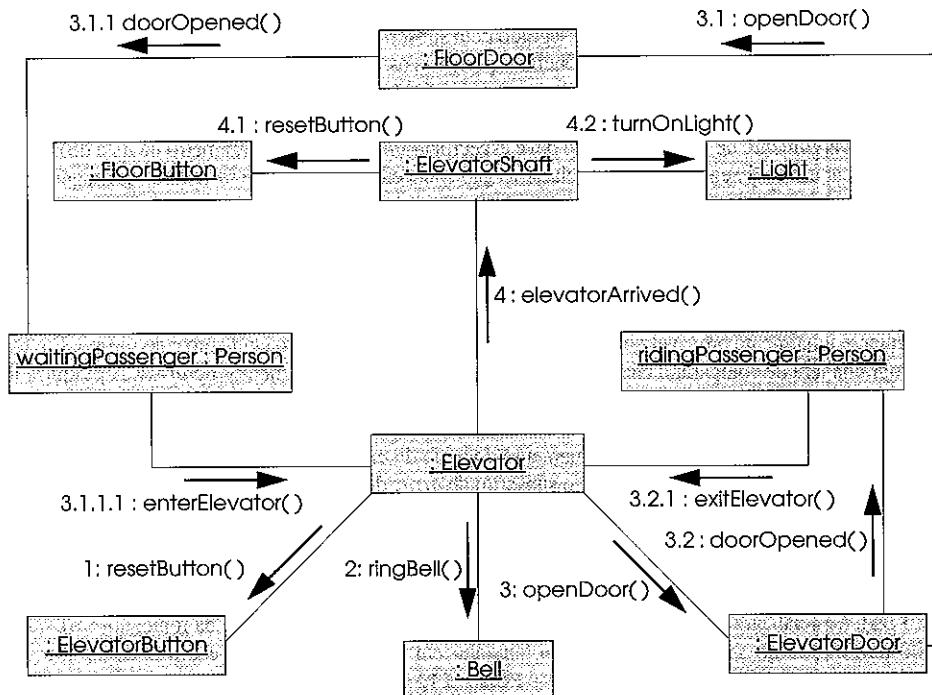


Fig. 7.20 Diagrama de colaborações para passageiros entrando no elevador e saindo dele.

Resumo

- Java armazena listas de valores em *arrays*. O *array* é um grupo contíguo de posições de memória relacionadas. Essas posições são relacionadas pelo fato de todas terem o mesmo nome e o mesmo tipo. Para se referir a uma posição ou a um elemento em particular dentro do *array*, especificamos o nome do *array* e o subscrito (ou índice ou número da posição) do elemento.
- Cada *array* tem um membro `length` que é ajustado com o número de elementos no *array* no momento em que o programa cria o objeto *array*.
- O subscrito pode ser um inteiro ou uma expressão do tipo inteiro. Se o programa utilizar uma expressão como um subscrito, a expressão será avaliada para determinar o elemento particular do *array*.
- Os *arrays* Java sempre iniciam com o elemento 0; é importante notar a diferença ao se referir ao “sétimo elemento do *array*” em oposição ao “elemento sete do *array*”. O sétimo elemento tem um subscrito de 6, enquanto o elemento sete tem um subscrito de 7 (na verdade, o oitavo elemento do *array*).
- Os *arrays* ocupam espaço na memória e são considerados objetos. Deve-se utilizar operador `new` para reservar espaço para um *array*. Por exemplo, o seguinte código cria um *array* de 100 valores `int`:

```
int b[] = new int[ 100 ];
```

- Ao declarar um *array*, o tipo do *array* e os colchetes podem ser combinados no início da declaração para indicar que todos os identificadores na declaração representam *arrays*, como em

```
double[] array1, array2;
```

- Os elementos de um *array* podem ser inicializados utilizando-se listas de inicializadores na declaração e por entrada.
- Java impede a referência de elementos além dos limites de um *array*. Se isso acontecer durante a execução do programa, ocorre uma exceção do tipo `ArrayIndexOutOfBoundsException`.

- As variáveis constantes devem ser inicializadas com uma expressão constante antes de elas serem utilizadas, e não podem ser modificadas depois.
- Para passar um *array* para um método, passe o nome do *array*. Para passar um único elemento de um *array* para um método, simplesmente passe o nome do *array* seguido pelo subscrito do elemento em particular.
- Os *arrays* são passados a métodos como referências – portanto, os métodos chamados podem modificar os valores de elementos nos *arrays* originais do chamador. Os elementos individuais de *arrays* dos tipos de dados primitivos são passados para métodos por valor.
- Para receber um argumento de *array*, o método deve especificar um *array* como parâmetro na lista de parâmetros.
- Pode-se classificar um *array* com a técnica de classificação conhecida como *bubble sort*. Várias passagens do *array* são feitas. A cada passagem, pares de elementos adjacentes são comparados. Se um par estiver na ordem (ou os valores forem idênticos), ele é deixado como está. Se um par estiver fora da ordem, os valores são trocados. Para *arrays* pequenos, a classificação pelo *bubble sort* é aceitável, mas para *arrays* maiores é ineficiente comparada com algoritmos de classificação mais sofisticados.
- A pesquisa linear compara cada elemento do *array* com a chave de pesquisa. Se o *array* não estiver em qualquer ordem particular, é igualmente provável que o valor esteja localizado no primeiro elemento e no último. Em média, portanto, o programa terá de comparar a chave de pesquisa com a metade dos elementos do *array*. A pesquisa linear funciona bem para *arrays* pequenos e é aceitável mesmo para *arrays* grandes não-ordenados.
- Para *arrays* ordenados, a pesquisa binária deixa de levar em consideração uma metade dos elementos no *array* depois de cada comparação. O algoritmo localiza o elemento do meio do *array* e o compara à chave de pesquisa. Se eles forem iguais, a chave de pesquisa é localizada e o subscrito de *array* desse elemento é devolvido. Caso contrário, o problema reduz-se a pesquisar uma metade do *array* que está ainda sob consideração. No pior caso, pesquisar um *array* ordenado de 1024 elementos vai exigir somente 10 comparações quando usada a pesquisa binária.
- A maioria dos componentes GUI tem um método `setFont` para alterar a fonte do texto no componente GUI. O método exige um objeto `Font` (pacote `java.awt`) como seu argumento.
- O objeto `Font` é inicializado com três argumentos – um `String` que indica o nome da fonte, um `int` que representa o estilo da fonte e um `int` que representa o tamanho em pontos da fonte. O estilo pode ser `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` ou `Font.BOLD + Font.ITALIC`. O tamanho em pontos representa o tamanho da fonte – há 72 pontos em uma polegada. O tamanho real na tela pode variar de acordo com o tamanho da tela e a resolução da tela.
- Podem-se utilizar os *arrays* para representar tabelas de valores que consistem em informações organizadas em linhas e colunas. Para identificar um elemento particular de uma tabela, são especificados dois subscritos: o primeiro identifica a linha em que o elemento está contido e o segundo identifica a coluna em que o elemento está contido. As tabelas ou os *arrays* que exigem dois subscritos para identificar um elemento particular são chamados de *arrays* bidimensionais.
- O *array* bidimensional pode ser inicializado com uma lista de inicializadores na forma

```
tipoDoArray nomeDoArray [] [] = { { sublista linha1 }, { sublista linha2 }, ... };
```

- Para criar dinamicamente um *array* com um número fixo de linhas e colunas, utilize

```
tipoDoArray nomeDoArray [] [] = new tipoDoArray [ numLinhas ] [ numColunas ];
```

- Para passar uma linha de um *array* bidimensional para um método que recebe um *array* unidimensional, simplesmente passe o nome do *array* seguido pelo subscrito da linha.

Terminologia

<code>a[i]</code>	<i>classificação por afundamento</i>
<code>a[i][j]</code>	<i>classificando um array</i>
<i>área temporária para troca de valores</i>	<i>colchetes []</i>
<i>array</i>	<i>constante identificada</i>
<i>array bidimensional</i>	<i>declarar um array</i>
<i>array m por n</i>	<i>elemento de um array</i>
<i>array multidimensional</i>	<i>erro por um</i>
<i>array unidimensional</i>	<i>final</i>
<i>chave de pesquisa</i>	<code>Font.BOLD</code>
<i>classe Font de java.awt</i>	<code>Font.ITALIC</code>
<i>classificação</i>	<code>Font.PLAIN</code>
<i>classificação com o bubble sort</i>	<i>formato de tabela</i>

<i>índice</i>	<i>pesquisa linear em um array</i>
<i>inicializador</i>	<i>pesquisando um array</i>
<i>inicializar um array</i>	<i>subscrito</i>
<i>lista de inicializadores de array</i>	<i>subscrito de coluna</i>
<i>lvalue</i>	<i>subscrito de linha</i>
<i>método setFont</i>	<i>tabela de valores</i>
<i>nome de um array</i>	<i>valor de um elemento</i>
<i>número de posição</i>	<i>variável constante</i>
<i>passagem de uma classificação com bubble sort</i>	<i>verificação de limites</i>
<i>passando arrays para métodos</i>	<i>zero-ésimo elemento</i>
<i>passar por referência</i>	
<i>passar por valor</i>	
<i>pesquisa binária em um array</i>	

Exercícios de auto-revisão

- 7.1** Preencha as lacunas em cada uma das frases seguintes:
- Listas e tabelas de valores podem ser armazenadas em _____.
 - Os elementos de um *array* são relacionados pelo fato de eles terem o mesmo _____ e o mesmo _____.
 - O número utilizado para se referir a um elemento particular de um *array* é chamado de seu _____.
 - O processo de colocar os elementos de um *array* em ordem é chamado de _____ do *array*.
 - Determinar se um *array* contém um certo valor-chave é um processo que se chama _____ do *array*.
 - O *array* que utiliza dois subscritos é um *array* _____.
- 7.2** Determine se cada uma das seguintes afirmações é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- O *array* pode armazenar muitos tipos de valores diferentes.
 - O subscrito de *array* normalmente deve ser do tipo de dados **float**.
 - O elemento individual de um *array* que é passado para um método e modificado nesse método conterá o valor modificado quando o método chamado completar sua execução.
- 7.3** Responda às questões seguintes relacionadas a um *array* chamado **fractions**.
- Definir uma variável constante **ARRAY_SIZE** inicializada com 10.
 - Declarar um *array* com elementos **ARRAY_SIZE** do tipo **float** e inicializá-lo com 0.
 - Identificar o quarto elemento do *array*.
 - Fazer referência ao elemento 4 do *array*.
 - Atribuir o valor 1.667 ao elemento 9 do *array*.
 - Atribuir o valor 3.333 ao sétimo elemento do *array*.
 - Somar todos os elementos do *array* utilizando uma estrutura de repetição **for**. Defina a variável inteira **x** como uma variável de controle para o laço.
- 7.4** Responda às questões seguintes relacionadas a um *array* chamado **table**.
- Declarar e criar o *array* como um *array* de inteiros com 3 linhas e 3 colunas. Assuma que a variável constante **ARRAY_SIZE** foi definida como 3.
 - Quantos elementos o *array* contém?
 - Utilizar uma estrutura de repetição **for** para inicializar cada elemento do *array* com a soma de seus subscritos. Suponha que as variáveis inteiros **x** e **y** são declaradas como variáveis de controle.
- 7.5** Localize o erro em cada um dos seguintes segmentos de programa e corrija o erro.
- final int ARRAY_SIZE = 5;**
ARRAY_SIZE = 10;
 - Suponha **int b[] = new int[10];**
for (int i = 0; i <= b.length; i++)
b[i] = 1;
 - Suponha **int a[][] = { { 1, 2 }, { 3, 4 } };**
a[1, 1] = 5;

Respostas aos exercícios de auto-revisão

- 7.1** a) arrays. b) nome, tipo. c) subscrito (ou índice ou número de posição). d) classificação. e) pesquisa. f) bidimensional (ou com dois subscritos).
- 7.2** a) Falsa. O *array* pode armazenar apenas valores do mesmo tipo.
 b) Falsa. O subscrito de *array* deve ser um inteiro ou uma expressão de tipo inteiro.
 c) Falsa para elementos individuais de um *array* dos tipos de dados primitivos porque eles são passados por valor. Se for passada uma referência para um *array*, as modificações nos elementos do *array* são refletidas no original. Além disso, um elemento individual de um tipo não-primitivo é passado para um método por referência e as alterações no objeto serão refletidas no elemento do *array* original.
- 7.3** a) `final int ARRAY_SIZE = 10;`
 b) `float fractions[] = new float[ARRAY_SIZE];`
 c) `fractions[3]`
 d) `fractions[4]`
 e) `fractions[9] = 1.667;`
 f) `fractions[6] = 3.333;`
 g) `float total = 0;`
`for (int x = 0; x < fractions.length; x++)`
 `total += fractions[x];`
- 7.4** a) `int table[][] = new int[ARRAY_SIZE][ARRAY_SIZE];`
 b) Nove.
 c) `for (int x = 0; x < table.length; x++)`
 `for (int y = 0; y < table[x].length; y++)`
 `table[x][y] = x + y;`
- 7.5** As soluções para o Exercício 7.5 são as seguintes:
 a) Erro: atribuir um valor a uma variável constante depois de ela ter sido inicializada.
 Correção: atribua o valor correto à variável constante em uma declaração `final int ARRAY_SIZE` ou criar outra variável.
 b) Erro: fazer referência a um elemento de *array* além dos limites do *array* (`b[10]`).
 Correção: altere o operador `<=` para `<`.
 c) Erro: especificação dos subscritos do *array* feita incorretamente.
 Correção: altere a instrução para `a[1][1] = 5;`.

Exercícios

- 7.6** Preencha as lacunas de cada uma das seguintes sentenças:
- Java armazena listas de valores em _____.
 - Os elementos de um *array* são relacionados pelo fato de eles _____.
 - Ao fazer referência a um elemento de um *array*, o número de posição contido entre os colchetes é chamado de _____.
 - Os nomes dos quatro elementos do *array* `p` são _____, _____ e _____.
 - O processo de nomear um *array*, declarar seu tipo e especificar o número de dimensões no *array* chama-se _____ do *array*.
 - O processo de colocar os elementos de um *array* na ordem crescente ou decrescente chama-se _____.
 - Em um *array* bidimensional, o primeiro subscrito identifica a _____ de um elemento e o segundo subscrito identifica a _____ de um elemento.
 - Um *array* `m` por `n` contém _____ linhas, _____ colunas e _____ elementos.
 - O nome do elemento na linha 3 e na coluna 5 do *array* `d` é _____.
- 7.7** Determine se cada uma das seguintes afirmações é verdadeira ou falsa. Se for falsa, explique por quê.
- Para se referir a uma localização ou a um elemento particular dentro de um *array*, especificamos o nome do *array* e o valor do elemento particular.
 - Uma declaração de *array* reserva espaço para o *array*.

- c) Para indicar que 100 posições devem ser reservadas para o *array* de inteiros **p**, o programador escreve a declaração
`p[100];`
- d) Um programa em Java que inicializa os elementos de um *array* de 15 elementos como zero deve conter pelo menos uma instrução **for**.
- e) Um programa Java que soma os elementos de um *array* de subscrito duplo deve conter instruções aninhadas **for**.

7.8 Escreva instruções Java para realizar cada um das seguintes tarefas:

- a) Exibir o valor do sétimo elemento do *array* de caracteres **f**.
- b) Inicializar cada um dos cinco elementos de um *array* unidimensional de inteiros **g** como **8**.
- c) Somar os elementos de um *array* de ponto flutuante **c** de 100 elementos.
- d) Copiar o *array* **a** de 11 elementos para a primeira parte do *array* **b**, contendo 34 elementos.
- e) Determinar e imprimir os maiores e menores valores contidos no *array* de ponto flutuante **w** de 99 elementos.

7.9 Considere um *array* de inteiros **t** 2 por 3:

- a) Escreva uma instrução que declare e crie **t**.
- b) Quantas linhas tem **t**?
- c) Quantas colunas tem **t**?
- d) Quantos elementos tem **t**?
- e) Escreva os nomes de todos os elementos na segunda linha de **t**.
- f) Escreva os nomes de todos os elementos na terceira coluna de **t**.
- g) Escreva uma única instrução que configura o elemento de **t** na linha 1 e na coluna 2 como zero.
- h) Escreva uma série de instruções que inicializam cada elemento de **t** como zero. Não utilize uma estrutura de repetição.
- i) Escreva uma estrutura aninhada **for** que inicializa cada elemento de **t** como zero.
- j) Escreva uma estrutura aninhada **for** que lê os valores para os elementos de **t** pelo teclado.
- k) Escreva uma série de instruções que determina e imprime o menor valor no *array* **t**.
- l) Escreva uma instrução que exibe os elementos da primeira linha de **t**.
- m) Escreva uma instrução que soma os elementos da quarta coluna de **t**.
- n) Escreva uma série de instruções que imprime o *array* **t** num formato elegante de tabela. Liste os subscritos de coluna como títulos ao longo do topo da tabela e liste os subscritos de linha à esquerda de cada linha.

7.10 Utilize um *array* unidimensional para resolver o seguinte problema: uma empresa paga seu pessoal de vendas com base em comissões. O pessoal de vendas recebe R\$ 200 por semana, mais 9% de suas vendas brutas durante aquela semana. Por exemplo, o vendedor que vende R\$ 5000 brutos em uma semana, recebe R\$ 200 mais 9% de R\$ 5000, ou um total de R\$ 650. Escreva um *applet* (utilizando um *array* de contadores) que determina quantos vendedores ganharam remuneração em cada um dos seguintes intervalos (assuma que o salário de cada vendedor foi truncado para uma quantidade de inteira):

- a) R\$ 200-R\$ 299
- b) R\$ 300-R\$ 399
- c) R\$ 400-R\$ 499
- d) R\$ 500-R\$ 599
- e) R\$ 600-R\$ 699
- f) R\$ 700-R\$ 799
- g) R\$ 800-R\$ 899
- h) R\$ 900-R\$ 999
- i) R\$ 1000 e acima

O applet deve usar as técnicas de GUI apresentadas no Capítulo 6. Exiba os resultados em uma **JTextArea**. Use o método **setText** de **JTextArea** para atualizar os resultados após cada valor digitado pelo usuário ser lido.

7.11 A classificação com o *bubble sort* apresentada na Fig. 7.11 é ineficiente para arrays grandes. Faça as seguintes modificações simples para aprimorar o desempenho da classificação com o *bubble sort*:

- a) Depois da primeira passagem, garante-se que o número maior está no elemento de número mais alto do *array*; após a segunda passagem, os dois números mais altos estão “no lugar”; e assim por diante. Em vez de fazer nove comparações em cada passagem, modifique o *bubble sort* para fazer oito comparações na segunda passagem, sete na terceira passagem, e assim por diante.
- b) Os dados no *array* já podem estar na ordem adequada ou quase na ordem adequada; assim, por que fazer nove passagens se menos seriam suficientes? Modifique a classificação para verificar no fim de cada passa-

gem se alguma troca foi feita. Se nenhuma foi feita, os dados já devem estar na ordem adequada, assim o programa deve terminar. Se foram feitas trocas, é necessária pelo menos mais uma passagem.

7.12 Escreva instruções que realizam as seguintes operações com um *array* unidimensional:

- Inicializar os 10 elementos do *array* de inteiros **counts** com zero.
- Adicionar 1 a cada um dos 15 elementos do *array* de inteiros **bonus**.
- Imprimir os cinco valores do *array* de inteiros **bestScores** em formato de coluna.

7.13 Utilize um *array* unidimensional para resolver o seguinte problema: escrever um *applet* que leia 20 números, cada um deles estando entre 10 e 100, inclusive. À medida que cada número for lido, imprima-o somente se ele não for uma duplicata de um número já lido. Previna-se para o “pior caso” em que todos os 20 números são diferentes. Utilize o menor *array* possível para resolver esse problema. O *applet* deve usar as técnicas GUI apresentadas no Capítulo 6. Exiba os resultados em uma **JTextArea**. Use o método **setText** de **JTextArea** para atualizar os resultados após cada valor digitado pelo usuário ser lido.

7.14 Rotule os elementos do *array* bidimensional 3 por 5 **sales** para indicar a ordem em que são configurados como zero pelo seguinte segmento de programa:

```
for ( int row = 0; row < sales.length; row++ )

    for ( int col = 0; col < sales[ row ].length; col++ )

        sales[ row ][ col ] = 0;
```

7.15 Escreva um *applet* para simular o lançamento de dois dados. O programa deve utilizar **Math.random** para lançar o primeiro dado e deve utilizar **Math.random** novamente para lançar o segundo dado. A soma dos dois valores deve então ser calculada. [Nota: como cada dado pode mostrar um valor de inteiro de 1 a 6, a soma dos valores irá variar de 2 a 12, com 7 sendo a soma mais freqüente e 2 e 12 sendo as somas menos freqüentes. A Fig. 7.21 mostra as 36 combinações possíveis dos dois dados. O programa deve lançar os dados 36.000 vezes. Utilize um *array* unidimensional para contar o número de vezes que cada soma possível aparece. Exiba os resultados em uma **JTextArea**, num formato de tabela. Além disso, determine se os totais são razoáveis (isto é, há seis maneiras de lançar um 7, então aproximadamente um sexto de todos os lançamentos devem dar 7). O *applet* deve usar as técnicas GUI apresentadas no Capítulo 6. Providencie um **JButton** para permitir ao usuário do *applet* lançar os dados mais 36.000 vezes. O *applet* deve reiniciar os elementos do *array* unidimensional com 0 antes de lançar os dados novamente.]

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 7.21 Os 36 resultados possíveis ao se lançar dois dados.

7.16 O que faz o programa da Fig. 7.22?

```
1 // Exercício 7.16: WhatDoesThisDo.java
2
```

Fig. 7.22 Determine o que faz este programa (parte 1 de 2).

```

3 // Pacotes do núcleo de Java
4 import java.awt.*;
5
6 // Pacotes de extensão de Java
7 import javax.swing.*;
8
9 public class WhatDoesThisDo extends JApplet {
10     int result;
11
12     public void init()
13     {
14         int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15
16         result = whatIsThis( array, array.length );
17
18         Container container = getContentPane();
19         JTextArea output = new JTextArea();
20         output.setText( "Result is: " + result );
21         container.add( output );
22     }
23
24     public int whatIsThis( int array2[], int size )
25     {
26         if ( size == 1 )
27             return array2[ 0 ];
28         else
29             return array2[ size - 1 ] +
30                 whatIsThis( array2, size - 1 );
31     }
32 }

```

Fig. 7.22 Determine o que faz este programa (parte 2 de 2).

- 7.17** Escreva um programa que joga 1000 vezes o jogo de dados *craps* (Fig. 6.9) e responda às seguintes perguntas:
- Quantos jogos são ganhos no primeiro lançamento, no segundo lançamento, ..., no vigésimo lançamento e depois do vigésimo lançamento?
 - Quantos jogos são perdidos no primeiro lançamento, no segundo lançamento, ..., no vigésimo lançamento e depois do vigésimo lançamento?
 - Quais são as chances de ganhar no jogo de dados? [Nota: você deve descobrir que o *craps* é um dos jogos de cassino mais honestos. O que você supõe que isso significa?]
 - Qual é a duração média do *craps*?
 - As chances de ganhar aumentam com a duração do jogo?

- 7.18** (*Sistema de reservas de passagens áreas*) Uma pequena companhia aérea acabou de comprar um computador para seu novo sistema automatizado de reservas. Eles pediram para você programar o novo sistema. Você deve escrever um *applet* para atribuir assentos em cada vôo do único avião da companhia aérea (capacidade: 10 assentos).

O programa deve exibir as seguintes alternativas:

```

Please type 1 for "smoking"
Please type 2 for "nonsmoking"

```

Se a pessoa digitar 1, o programa deve atribuir um assento na área de fumantes (assentos 1-5). Se a pessoa digitar 2, o programa deve atribuir um assento na área de não-fumantes (assentos 6-10). O programa, então, deve imprimir um cartão de embarque indicando o número do assento da pessoa e se ela está na área de fumantes ou de não-fumantes do avião.

Utilize um *array* unidimensional do tipo primitivo *boolean* para representar o mapa de assentos do avião. Inicialize todos os elementos do *array* com *false* para indicar que todos os assentos estão vazios. À medida que cada assento for atribuído, configure os elementos correspondentes do *array* com *true* para indicar que o assento não está mais disponível.

Naturalmente, o programa nunca deve atribuir um assento que já foi atribuído. Quando a área de fumantes estiver lotada, seu programa deve solicitar à pessoa se ela aceita ser colocada na área de não-fumantes (e vice-versa). Se for, faça a atribuição apropriada de assento. Se não for, imprima a mensagem "*Next flight leaves in 3 hours.*"

7.19 O que faz o seguinte programa?

```

1 // Exercícios 7.19: WhatDoesThisDo2.java
2
3 // Pacotes do núcleo de Java
4 import java.awt.*;
5
6 // Pacotes de extensão de Java
7 import javax.swing.*;
8
9 public class WhatDoesThisDo2 extends JApplet {
10
11     public void init()
12     {
13         int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14         JTextArea outputArea = new JTextArea();
15
16         someFunction( array, 0, outputArea );
17
18         Container container = getContentPane();
19         container.add( outputArea );
20     }
21
22     public void someFunction( int array2[], int x, JTextArea out )
23     {
24         if ( x < array2.length ) {
25             someFunction( array2, x + 1, out );
26             out.append( array2[ x ] + " " );
27         }
28     }
29 }
```

Fig. 7.23 Determine o que faz este programa.

7.20 Utilize um *array* bidimensional para resolver o seguinte problema. Uma empresa tem quatro vendedores (1 a 4) que vendem cinco produtos diferentes (1 a 5). Uma vez por dia, cada vendedor entrega uma nota de cada tipo de produto diferente vendido. Cada nota contém:

1. O número do vendedor;
2. O número do produto;
3. O valor total desse produto vendido nesse dia, em reais.

Portanto, cada vendedor passa entre 0 e 5 notas de vendas por dia. Suponha que as informações de todas as notas do mês passado estejam disponíveis. Escreva um *applet* que lerá todas essas informações para as vendas do último mês e resumirá as vendas totais por vendedor e produto. Todos os totais devem ser armazenados no *array* bidimensional **sales**. Depois de processar todas as informações referentes ao último mês, exiba os resultados em formato de tabela, com cada uma das colunas representando um vendedor específico e cada uma das linhas representando um produto específico. Coloque um total em cada linha para indicar o total de vendas daquele produto no mês passado; coloque um total em cada coluna para indicar o total de vendas daquele vendedor no mês passado. Sua tabela impressa deve incluir esses totais à direita das linhas totalizadas e na parte inferior das colunas totalizadas. Exiba os resultados em uma **JTextArea**.

7.21 (*Gráfico de Tartaruga*) A linguagem Logo, que é popular entre jovens usuários de computador, tornou famoso o conceito de *gráfico de tartaruga*. Imagine uma tartaruga mecânica que caminha pela sala sob controle de um programa Java. A tartaruga segura uma caneta em uma de duas posições, para cima ou para baixo. Enquanto a caneta estiver para baixo, a tartaruga desenha formas à medida que se move; enquanto a caneta estiver para cima, a tartaruga se move quase livremente sem escrever nada. Nesse problema você simulará a operação da tartaruga e criará também um bloco de rascunho computadorizado.

Utilize um *array* de 20 por 20 **floor** que é inicializado com zeros. Leia os comandos a partir de um *array* que os contenha. Monitore a posição atual da tartaruga durante todo o tempo e se a caneta está no momento para cima ou para baixo. Suponha que a tartaruga sempre inicie na posição 0,0 do chão com a caneta para cima. O conjunto de comandos para a tartaruga que seu programa deve processar é o seguinte:

Comando	Significado
1	Caneta para cima
2	Caneta para baixo
3	Vire para a direita
4	Vire para a esquerda
5,10	Avance 10 espaços (ou um número diferente de 10)
6	Imprima o array 20 por 20
9	Fim dos dados (sentinela)

Fig. 7.24 Comandos para o gráfico de tartaruga.

Suponha que a tartaruga esteja em algum lugar próximo ao centro do chão. O seguinte “programa” desenharia e imprimia um quadrado de 12 por 12 deixando a caneta na posição levantada:

```

2
5,12
3
5,12
3
5,12
3
5,12
1
6
9

```

À medida que a tartaruga se move com a caneta para baixo, configure os elementos apropriados do array **floor** como 1s. Quando o comando **6** (impressão) é dado, onde quer que haja um **1** no array, exiba um asterisco ou algum caractere diferente que você escolher. Onde quer que haja um zero, exiba um caractere de espaço em branco. Escreva um *applet* Java para implementar os recursos para os gráficos de tartaruga discutidos aqui. O applet deve exibir o gráfico de tartaruga em uma **JTextArea**, usando uma fonte de caracteres *Monospaced*. Escreva vários programas de gráfico de tartaruga para desenhar formas interessantes. Adicione outros comandos para aumentar os recursos de sua linguagem de gráfico de tartaruga.

7.22 (Passeio do Cavalo) Um dos problemas mais difíceis e interessantes para os fãs de xadrez é o problema do Passeio do Cavalo, originalmente proposto pelo matemático Euler. A pergunta é: a peça de xadrez chamada cavalo pode mover-se em um tabuleiro vazio e tocar todos os 64 quadrados uma vez e unicamente uma vez? Aqui, estudamos esse intrigante problema em profundidade.

O cavalo move-se em um caminho em forma de L (duas posições em uma direção e então uma em uma direção perpendicular). Portanto, a partir de um quadrado no meio de um tabuleiro vazio, o cavalo pode fazer oito movimentos diferentes (numerados de 0 a 7), como mostrado na Fig. 7.25.

- Desenhe um tabuleiro de 8 por 8 em uma folha de papel e experimente fazer o Passeio do Cavalo à mão. Coloque um **1** no primeiro quadrado para o qual você move o cavalo, um **2** no segundo quadrado, um **3** no terceiro, etc. Antes de iniciar o passeio, estime aonde você imagina chegar, lembrando que o passeio completo consiste em 64 movimentos. Até onde você foi? Isso foi próximo de sua estimativa?
- Agora vamos desenvolver um programa que moverá o cavalo pelo tabuleiro. O tabuleiro é representado por um array bidimensional de 8 por 8 chamado de **board**. Cada um dos quadrados é inicializado como zero. Descrevemos cada um dos oito movimentos possíveis em termos de seus componentes verticais e horizontais. Por exemplo, um movimento do tipo 0 como mostrado na Fig. 7.25 consiste em mover o cavalo horizontalmente dois quadrados para direita e verticalmente um quadrado para cima. O movimento 2 consiste em mover horizontalmente um quadrado para a esquerda e verticalmente dois quadrados para cima. Movimentos horizontais para a esquerda e movimentos verticais para cima são indicados com números negativos. Os oito movimentos podem ser descritos por dois arrays unidimensionais, **horizontal** e **vertical**, como segue:

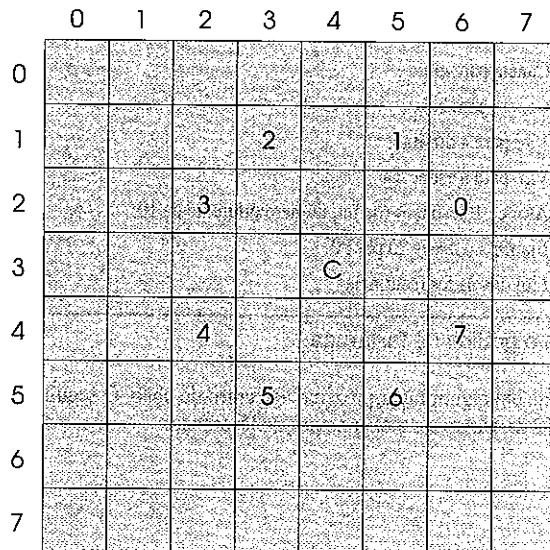


Fig. 7.25 Os oito movimentos possíveis do cavalo.

```

horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 2 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1

```

Faça com que as variáveis `currentRow` e `currentColumn` indiquem a linha e a coluna da posição atual do cavalo. Para fazer um movimento do tipo `moveNumber`, onde `moveNumber` está entre 0 e 7, seu programa utiliza as instruções

```

currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];

```

Mantenha um contador que varie de 1 a 64. Registre a última contagem em cada quadrado para que o cavalo se move. Teste cada movimento potencial para ver se o cavalo já visitou esse quadrado. Teste cada movimento em potencial para assegurar que o cavalo não saia fora do tabuleiro. Escreva um programa para mover o cavalo pelo tabuleiro. Execute o programa. Quantos movimentos o cavalo fez?

- c) Depois de tentar escrever e executar o programa para o Passeio do Cavalo, você provavelmente desenvolveu algumas percepções valiosas. Utilizaremos essas percepções para desenvolver uma *heurística* (ou estratégia) para mover o cavalo. A heurística não garante sucesso, mas uma heurística cuidadosamente desenvolvida aprima significativamente a chance de sucesso. Você pode ter observado que os quadrados exter-

nos são mais problemáticos que os quadrados próximos ao centro do tabuleiro. Na verdade, os quadrados mais problemáticos ou inacessíveis são os quatro cantos.

A intuição pode sugerir que você deva tentar mover o cavalo para os quadrados mais problemáticos primeiro e deixar abertos aqueles que são fáceis de alcançar, de modo que, quando o tabuleiro ficar congestionado próximo do fim, haja uma chance maior de sucesso.

Podemos desenvolver uma “acessibilidade heurística” classificando cada um dos quadrados de acordo com seu grau de acessibilidade e depois sempre movendo o cavalo (utilizando os movimentos em forma de L) para o quadrado mais inacessível. Rotulamos um *array* bidimensional **accessibility** com números que indicam a partir de quantos quadrados cada quadrado particular pode ser acessado. Em um tabuleiro vazio, cada quadrado central tem acessibilidade 8, cada canto tem acessibilidade 2 e os outros quadrados têm números de acessibilidade 3, 4 ou 6, como segue:

2	3	4	4	4	4	3	2
3	4	6	6	6	4	3	
4	6	8	8	8	6	4	
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	4	3	
2	3	4	4	4	3	2	

Escreva uma versão do Passeio do Cavalo utilizando a heurística de acessibilidade. O cavalo sempre deve se mover para o quadrado com o número de acessibilidade mais baixo. Em caso de empate, o cavalo pode mover-se para qualquer um dos quadrados empatados. Portanto, o passeio pode iniciar em qualquer um dos quatro cantos. [Nota: à medida que o cavalo se move pelo tabuleiro de xadrez, o programa deve reduzir os números de acessibilidade à medida que mais quadrados se tornem ocupados. Dessa maneira, a qualquer momento durante o passeio, o número de acessibilidade de cada quadrado disponível permanecerá precisamente igual ao número de quadrados a partir dos quais esse quadrado pode ser alcançado.] Execute essa versão do programa. Você conseguiu um passeio completo? Modifique o programa para executar 64 passeios, um iniciando a partir de cada quadrado do tabuleiro de xadrez. Quantos passeios completos você obteve?

- d) Escreva uma versão do programa Passeio do Cavalo que, diante de um empate entre dois ou mais quadrados, decide qual quadrado escolher olhando adiante para ver quais os quadrados alcançáveis a partir dos quadrados geradores do empate. O programa deve mover-se para o quadrado através do qual seu próximo movimento chegaria ao quadrado com o número de acessibilidade mais baixo.

7.23 (Passeio do Cavalo: abordagens com força bruta) No Exercício 7.22, desenvolvemos uma solução para o problema do Passeio do Cavalo. A abordagem utilizada, denominada “acessibilidade heurística”, gera muitas soluções e roda eficientemente.

À medida que os computadores continuam aumentando a potência, seremos capazes de resolver cada vez mais problemas com a pura capacidade do computador e algoritmos relativamente simples. Vamos chamar essa abordagem de solução de problemas pela “força bruta”.

- a) Utilize a geração de números aleatórios para permitir ao cavalo andar no tabuleiro de xadrez (em seus movimentos válidos em forma de L, naturalmente) de maneira aleatória. O programa deve executar um passeio e imprimir o tabuleiro de xadrez final. Até onde o cavalo chegou?
- b) Muito provavelmente, o programa precedente produziu um passeio relativamente curto. Agora modifique o programa para tentar 1000 passeios. Utilize um *array* unidimensional para monitorar o número de passeios de cada comprimento. Quando o programa terminar de tentar os 1000 passeios, ele deve imprimir essas informações organizadas em forma de tabela. Qual foi o melhor resultado?
- c) Muito provavelmente, o programa precedente deu-lhe alguns passeios “respeitáveis”, mas nenhum passeio completo. Agora “solte todas as amarras” e simplesmente deixe o programa rodar até produzir um passeio completo. (Atenção: essa versão do programa pode levar horas para ser executada em um computador poderoso.) Mais uma vez, mantenha uma tabela do número de passeios de cada comprimento e imprima essa tabela quando o primeiro passeio completo for localizado. Quantos passeios o programa tentou antes de produzir um passeio completo? Quanto tempo ele levou?
- d) Compare a versão de força bruta do Passeio do Cavalo com a versão de acessibilidade heurística. Qual exigiu um estudo mais cuidadoso do problema? Qual algoritmo foi mais difícil de desenvolver? Qual exigiu mais capacidade do computador? Poderíamos ter certeza (com antecedência) de obter um passeio completo com a abordagem de acessibilidade heurística? Poderíamos ter certeza (com antecedência) de obter um

passeio completo com a abordagem de força bruta? Argumente sobre as vantagens e desvantagens de resolver problemas em geral usando força bruta.

7.24 (*Oito Rainhas*) Outro problema difícil para fãs do xadrez é o problema das Oito Rainhas. Eis o problema: é possível colocar oito rainhas em um tabuleiro de xadrez vazio de modo que nenhuma rainha esteja “atacando” qualquer outra, isto é, sem que duas rainhas estejam na mesma linha, na mesma coluna ou na mesma diagonal? Utilize o raciocínio desenvolvido no Exercício 7.22 para formular uma heurística para resolver o problema das Oito Rainhas. Execute seu programa. (*Dica:* é possível atribuir um valor para cada quadrado do tabuleiro de xadrez que indica quantos quadrados de um tabuleiro de xadrez vazio “são eliminados” se uma rainha for colocada nesse quadrado. A cada um dos cantos seria atribuído o valor 22, como na Fig. 7.26.) Depois que esses “números de eliminação” forem colocados em todos os 64 quadrados, uma heurística apropriada talvez seja: coloque a próxima rainha no quadrado com o menor número de eliminação. Por que essa estratégia é intuitivamente atraente?

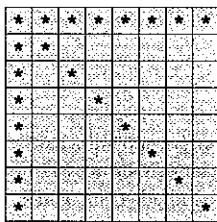


Fig. 7.26 Os 22 quadrados eliminados ao se colocar a rainha no canto esquerdo superior.

7.25 (*Oito Rainhas: abordagens com força bruta*) Nesse exercício, você desenvolverá várias abordagens da força bruta para resolver o problema das Oito Rainhas apresentado no Exercício 7.24.

- Resolva o exercício das Oito Rainhas, utilizando a técnica de força bruta aleatória desenvolvida no Exercício 7.23.
- Utilize uma técnica exaustiva (isto é, tente todas as combinações possíveis de oito rainhas no tabuleiro de xadrez).
- Por que você supõe que a abordagem de força bruta exaustiva pode não ser apropriada para resolver o problema do Passeio do Cavalo?
- Compare e contraste as abordagens de força bruta aleatória e da força bruta exaustiva.

7.26 (*Passeio do Cavalo: teste do passeio fechado*) No Passeio do Cavalo, o passeio completo ocorre quando o cavalo faz 64 movimentos tocando cada quadrado do tabuleiro de xadrez uma e somente uma vez. O passeio fechado ocorre quando o 64º movimento está a um movimento de distância do quadrado em que o cavalo iniciou o passeio. Modifique o programa escrito no Exercício 7.22 para testar se ocorreu um passeio fechado ou um passeio completo.

7.27 (*O Crivo de Eratóstenes*) O inteiro primo é qualquer inteiro que seja divisível exatamente apenas por si mesmo e por 1. O Crivo de Eratóstenes é um método para localizar números primos. Ele opera da seguinte forma:

- Crie um *array* do tipo primitivo `boolean` com todos os elementos inicializados como `true`. Os elementos do *array* com subscritos primos permanecerão como `true`. Todos os outros elementos do *array* acabarão sendo configurados como `false`.
- Iniciando com o subscrito 2 do *array* (o subscrito 1 deve ser primo), determine se um determinado elemento é `true`. Caso seja, percorra com um laço o restante do *array* e configure como `false` cada elemento cujo subscrito é um múltiplo do subscrito para o elemento com valor `true`. Depois, continue o processo com o próximo elemento com valor `true`. Para o subscrito de *array* 2, todos os elementos além de 2 no *array* que são múltiplos de 2 serão configurados como `false` (subscritos 4, 6, 8, 10, etc.); para o subscrito de *array* 3, todos os elementos além de 3 no *array* que são múltiplos de 3 serão configurados como `false` (subscritos 6, 9, 12, 15, etc.); e assim por diante.

Quando esse processo estiver completo, os elementos do *array* que ainda estiverem configurados como `um` indicam que o subscrito é um número primo. Esses subscritos podem então ser exibidos. Escreva um programa que utiliza um *array* de 1000 elementos para determinar e imprimir os números primos entre 1 e 999. Ignore o elemento 0 do *array*.

7.28 (*Classificação com o bucket sort*) A classificação com o *bucket sort* inicia com um *array* unidimensional de inteiros positivos que será classificado e um *array* bidimensional de inteiros com linhas de subscritos de 0 a 9 e colunas de subscritos de 0 a $n - 1$, onde n é o número de valores no *array* que será classificado. Cada linha do *array* bidimensional é um depósito ou *bucket*. Escreva um *applet* que contenha um método **bucketSort** que recebe um *array* de inteiros como argumento e trabalha da seguinte maneira:

- Coloque cada valor do *array* unidimensional em uma linha do *array* de depósitos com base no dígito das unidades do valor. Por exemplo, 97 é colocado na linha 7, 3 é colocado na linha 3 e 100 é colocado na linha 0. Isso se chama “passagem de distribuição”.
- Percorra o *array* de depósito linha por linha e copie os valores de volta para o *array* original. Isso se chama “passagem de coleta”. A nova ordem dos valores precedentes no *array* unidimensional é 100, 3 e 97.
- Repita esse processo para a posição de cada dígito subsequente (dezenas, centenas, milhares, etc.).

Na segunda passagem, 100 é colocado na linha 0, 3 é colocado na linha 0 (porque 3 não tem dígito de dezenas) e 97 é colocado na linha 9. Depois da passagem de coleta, a ordem dos valores no *array* unidimensional é 100, 3 e 97. Na terceira passagem, 100 é colocado na linha 1, 3 é colocado na linha 0 e 97 é colocado na linha 0 (depois do 3). Depois da última passagem de coleta, o *array* original agora está ordenado.

Observe que o *array* bidimensional de depósitos é 10 vezes maior que o *array* de inteiros que está sendo classificado. Essa técnica de classificação fornece um melhor desempenho que uma classificação com o *bubble sort*, mas exige muito mais memória: a classificação com o *bubble sort* exige espaço para apenas um elemento adicional de dados. Esse é um exemplo da relação de compromisso espaço versus tempo: a classificação de depósito utiliza mais memória que a classificação de bolhas, mas seu desempenho é melhor. Essa versão da classificação com o *bucket sort* exige cópia de todos os dados de volta para o *array* original a cada passagem. Outra possibilidade é criar um segundo *array* bidimensional de depósitos e permitir os dados repetidamente entre os dois *arrays* de depósitos.

Exercícios de recusão

7.29 (*Classificação por seleção*) A classificação por seleção pesquisa um *array* procurando o menor elemento no *array*, depois permuta esse elemento com o primeiro elemento do *array*. O processo é repetido para o *subarray* que inicia com o segundo elemento. Cada passagem pelo *array* coloca um elemento em sua posição adequada. Para um *array* de n elementos, devem ser feitas $n - 1$ passagens e, para cada *subarray*, devem ser feitas $n - 1$ comparações para encontrar o menor valor. Quando o *subarray* que está processado contiver um elemento, o *array* está ordenado. Escreva o método recursivo **selectionSort** para executar esse algoritmo.

7.30 (*Palíndromos*) O palíndromo é um *string* que é lido da mesma maneira, da esquerda para a direita e da direita para a esquerda. Alguns exemplos de palíndromos são: “radar” e “socorram-me subi no ônibus em Marrocos”. Escreva um método recursivo **testPalindrome** que devolve o valor booleano **true** se o *string* armazenado no *array* for um palíndromo e **false** caso contrário. O método deve ignorar espaços e pontuação no *string*. [Dica: use o método **toCharArray** da classe **String**, que não recebe argumentos, para obter um *array* do tipo **char** que contém os caracteres do **String**. Depois, passe o *array* para o método **testPalindrome**.]

7.31 (*Pesquisa linear*) Modifique a Fig. 7.12 para utilizar o método recursivo **linearSearch** para realizar uma pesquisa linear do *array*. O método deve receber um *array* de inteiros, o tamanho do *array* e a chave de pesquisa como argumentos. Se a chave de pesquisa for encontrada, retorne o subscrito do *array*; caso contrário, retorne -1.

7.32 (*Pesquisa binária*) Modifique o programa da Fig. 7.13 para utilizar um método recursivo **binarySearch** para realizar a pesquisa binária do *array*. O método deve receber um *array* de inteiros e o subscrito inicial e o subscrito final como argumentos. Se a chave de pesquisa for encontrada, retorne o subscrito do *array*; caso contrário, retorne -1.

7.33 (*Oito Rainhas*) Modifique o programa das Oito Rainhas criado no Exercício 7.24 para resolver o problema recursivamente.

7.34 (*Imprima um array*) Escreva um método recursivo **printArray** que retorna um *array* de valores **int** e o tamanho do *array* como argumentos e não retorna nada. O método deve parar de processar e retornar quando receber um *array* de tamanho zero.

7.35 (*Imprimindo um string de trás para frente*) Escreva um método recursivo **stringReverse** que recebe um *array* de caracteres que contém um *string* como argumento, imprime o *string* de trás para frente e não retorna nada.

7.36 (*Localizando o valor mínimo em um array*) Escreva um método recursivo **recursiveMinimum** que recebe um *array* de inteiros e o tamanho do *array* como argumentos e retorna o menor elemento do *array*. O método deve parar de processar e retornar quando receber um *array* de um elemento.

7.37 (*Classificação rápida – Quicksort*) Nos exemplos e exercícios deste capítulo, discutimos as técnicas de classificação com o *bubble sort*, classificação com o *bucket sort* e a classificação por seleção. Agora apresentamos a técnica de classificação recursiva chamada de *Quicksort*. O algoritmo básico para um *array unidimensional* de valores é o seguinte:

- Passo de particionamento*: pegue o primeiro elemento do *array* não-ordenado e determine sua localização final no *array* ordenado (isto é, todos os valores à esquerda do elemento no *array* são menores que o elemento e todos os valores à direita do elemento no *array* são maiores que o elemento). Agora temos um elemento em sua posição adequada e dois *subarrays* não-ordenados.
- Passo recursivo*: realize o passo 1 em cada *subarray* não-ordenado.

Toda vez que o passo 1 for realizado em um *subarray*, outro elemento é colocado em sua posição final no *array* classificado e dois *subarrays* não-ordenados são criados. Quando um *subarray* consiste em um elemento, ele certamente está ordenado, portanto esse elemento está em sua localização final.

O algoritmo básico parece suficientemente simples, mas como determinarmos a posição final do primeiro elemento de cada *subarray*? Como exemplo, considere o seguinte conjunto de valores (o elemento em negrito é o elemento de particionamento – ele será colocado em sua localização final no *array* classificado) :

37 2 6 4 89 8 10 12 68 45

- Iniciando a partir do elemento mais à direita do *array*, compare cada elemento com **37** até um elemento menor que **37** ser encontrado, depois permuta **37** e esse elemento. O primeiro elemento menor que **37** é 12, depois **37** e 12 são permutados. O novo *array* é

12 2 6 4 89 8 10 **37** 68 45

- O elemento 12 está em itálico para indicar que acabou ser permutado com **37**.
- Iniciando da esquerda do *array*, mas começando com o elemento depois de 12, compare cada elemento com **37** até um elemento maior que **37** ser encontrado, depois permuta **37** e esse elemento. O primeiro elemento maior que **37** é 89, então **37** e 89 foram permutados. O novo *array* é

12 2 6 4 **37** 8 10 89 68 45

- Iniciando da direita mas começando com o elemento antes de 89, compare cada elemento com **37** até um elemento menor que **37** ser encontrado, depois permuta **37** e esse elemento. O primeiro elemento menor que **37** é 10, então **37** e 10 foram permutados. O novo *array* é

12 2 6 4 10 8 **37** 89 68 45

- Iniciando da esquerda mas começando com o elemento depois de 10, compare cada elemento com **37** até um elemento maior que **37** ser encontrado, depois permuta **37** e esse elemento. Não há mais elementos maiores que **37**; assim, quando compararmos **37** com ele mesmo, sabemos que **37** foi colocado na sua posição final do *array* ordenado.

Uma vez que foi feito o particionamento no *array* anterior, há dois *subarrays* não classificados. O *subarray* com valores menores que **37** contém 12, 2, 6, 4, 10 e 8. O *subarray* com valores maiores que **37** contém 89, 68 e 45. A classificação continua com ambos os *subarrays* sendo particionados da mesma maneira que o *array* original.

Com base na discussão precedente, escreva um método recursivo **quickSort** para classificar um *array unidimensional* de inteiros. O método deve receber como argumentos um *array* de inteiros, um subscrito inicial e um subscrito final. O método **partition** deve ser chamado por **quickSort** para realizar o passo de partição.

7.38 (*Percorrendo um labirinto*) A seguinte grade de #s e pontos (.) é uma representação de um labirinto na forma de um *array bidimensional*.

```
# # # # # # # # # #
# . . . # . . . . .
. # . # . # . . # .
# # # . # . . . # .
# . . . # # # . # .
# # # # . # . # . #
# . . . . . # . . .
# # # # # # . # .
# . . . . . # . . .
# # # # # # # # #
```

No *array* bidimensional precedente, os `#`s representam as paredes do labirinto e os pontos representam quadrados ou posições nos caminhos possíveis pelo labirinto. Os movimentos são permitidos apenas para as posições do *array* que contêm um ponto.

Há um algoritmo simples para percorrer um labirinto que garante a localização da saída (pressupondo que exista uma saída). Se não houver saída, você chegará à localização inicial novamente. Coloque a mão direita na parede à sua direita e comece a andar para frente. Nunca tire a mão da parede. Se o labirinto virar para a direita, siga a parede à direta. Contanto que você não remova a mão da parede, você acabará chegando à saída do labirinto. É possível que haja um caminho mais curto do que o que você tomou, mas a saída do labirinto é garantida se você seguir o algoritmo.

Escreva o método recursivo `mazeTraverse` para percorrer o labirinto. O método deve receber como argumentos um *array* de caracteres 12 por 12 que representa o labirinto e a posição de entrada no labirinto. À medida que `mazeTraverse` tenta localizar a saída do labirinto, ele deve colocar o caractere `X` em cada quadrado no caminho. O método deve exibir o labirinto depois de cada movimento de modo que o usuário possa observar enquanto o problema da saída do labirinto é resolvido.

7.39 (*Gerando labirintos aleatoriamente*) Escreva um método `mazeGenerator` que recebe como argumento um *array* bidimensional de caracteres de 12 por 12 e produza aleatoriamente um labirinto. O método também deve fornecer as posições inicial e final do labirinto. Teste o método `mazeTraverse` do Exercício 7.38 utilizando vários labirintos gerados aleatoriamente.

7.40 (*Labirintos de qualquer tamanho*) Generalize os métodos `mazeTraverse` e `mazeGenerator` dos Exercícios 7.38 e 7.39 para processar labirintos de qualquer largura e altura.

7.41 (*Simulação: a lebre e a tartaruga*) Nesse problema, você recriará uma das mais belas histórias folclóricas, a saber, a clássica competição entre a lebre e a tartaruga. Você utilizará geração de números aleatórios para desenvolver uma simulação desse memorável evento.

Nossos competidores começam a corrida no “quadrado 1” de 70 quadrados. Cada quadrado representa uma posição possível ao longo do percurso da competição. A linha de chegada está no quadrado 70. O primeiro competidor a alcançar ou passar o quadrado 70 é recompensado com um cesto de cenouras frescas e alface. O percurso envolve subir uma montanha escorregadia, de modo que, ocasionalmente, os competidores perdem terreno.

Há um relógio que bate uma vez por segundo. A cada batida do relógio, seu *applet* deve ajustar a posição dos animais de acordo com as regras da Fig. 7.27.

Utilize variáveis para monitorar a posição dos animais (isto é, os números de posição são 1–70). Inicie com cada animal na posição 1 (isto é, a “partida”). Se um animal escorrega para a esquerda antes do quadrado 1, move o animal de volta para quadrado 1.

Gere as porcentagens na tabela precedente produzindo um inteiro aleatório, i , no intervalo $1 \leq i \leq 10$. Para a tartaruga, realize uma “caminhada rápida” quando $1 \leq i \leq 5$, um “escorregão” quando $6 \leq i \leq 7$ ou uma “caminhada lenta” quando $8 \leq i \leq 10$. Utilize uma técnica semelhante para mover a lebre.

Inicie a corrida imprimindo

```
BANG !!!!!  
E ELES PARTIRAM !!!!!
```

Então, para cada batida do relógio (isto é, para cada repetição de um laço), imprima uma linha 70 de posições mostrando a letra `T` na posição da tartaruga e a letra `L` na posição da lebre. De vez em quando, os competidores aterrissarão no mesmo quadrado. Nesse caso, a tartaruga morde a lebre e o programa deve imprimir `AI!!!` começando nessa posição. Todas as outras posições diferentes de `T`, `L` ou `AI!!!` (no caso de um empate na mesma posição) devem estar em branco.

Depois que cada linha é impressa, teste se o animal alcançou ou passou o quadrado 70. Se isso ocorreu, imprima o vencedor e encerre a simulação. Se a tartaruga ganhar, imprima `A TARTARUGA VENCE!!! EH!!!` Se a lebre ganhar, imprima `A LEBRE GANHA. OH.` Se ambos os animais ganharem na mesma batida do relógio, você pode querer favorecer a tartaruga ou imprimir `TEMOS UM EMPATE`. Se nenhum animal ganhar, realize o laço novamente para simular a próxima batida do relógio. Quando você estiver pronto para executar o programa, monte um grupo de fãs para observar a corrida. Você se surpreenderá com como a sua audiência ficará empolgada!

Mais adiante no livro, apresentaremos vários recursos de Java, como gráficos, imagens, animação, som e *multithreading*. Ao estudar esses recursos, você pode se divertir aprimorando a simulação da competição entre a lebre e a tartaruga.

Animal	Tipo de Movimento	Porcentagem do tempo	Movimento real
Tartaruga	Caminhada rápida	50%	3 quadrados à direita
	Escoregão	20%	6 quadrados à esquerda
	Caminhada lenta	30%	1 quadrado à direita
Lebre	Dormindo	20%	Sem nenhum movimento
	Salto grande	20%	9 quadrados à direita
	Escoregão grande	10%	12 quadrados à esquerda
	Salto pequeno	30%	1 quadrado à direita
	Escoregão pequeno	20%	2 quadrados à esquerda

Fig. 7.27 Regras para ajustar as posições da tartaruga e da lebre.

Seção especial: construindo seu próprio computador

Nos próximos problemas, fazemos um desvio temporário para fora do mundo da programação em linguagem de alto nível. Vamos “abrir” um computador e examinar sua estrutura interna. Apresentamos programação em linguagem de máquina e escrevemos vários programas em linguagem de máquina. Para torná-la essa uma experiência especialmente valiosa, nós então construímos um computador (pela técnica de *simulação baseada em software*) no qual é possível executar seus programas em linguagem de máquina!

7.42 (*Programação em linguagem de máquina*) Vamos criar um computador que chamaremos de Simpletron. Como seu nome indica, é uma máquina simples, mas, como logo veremos, uma máquina poderosa também. O Simpletron executa programas escritos na única linguagem que ele entende diretamente, isto é, *Simpletron Machine Language* ou, abreviadamente, SML.

O Simpletron contém um *acumulador* – um “registrar especial” em que as informações são colocadas antes de o Simpletron utilizar essas informações em cálculos ou examiná-las de várias maneiras. Todas as informações no Simpletron são tratadas em termos de *palavras*. A palavra é um número decimal de quatro dígitos com um sinal como +3364, -1293, +0007, -0001, etc. O Simpletron é equipado com uma memória de 100 palavras e essas palavras são mencionadas por seus números de posição 00, 01, ..., 99.

Antes de executar um programa de SML, devemos *carregar* ou colocar o programa na memória. A primeira instrução de cada programa de SML é sempre colocada na posição 00. O simulador começará a executar as instruções a partir dessa posição.

Cada instrução escrita em SML ocupa uma palavra da memória do Simpletron (em consequência, as instruções são números decimais de quatro dígitos com sinal). Assumiremos que o sinal de uma instrução da SML é sempre mais, mas o sinal de uma palavra de dados pode ser mais ou menos. Cada posição na memória do Simpletron pode conter uma instrução, um valor de dados utilizado por um programa ou uma área de memória não utilizada (e portanto indefinida). Os primeiros dois dígitos de cada instrução da SML são os *códigos de operação* que especificam a operação a ser realizada. Os códigos de operação da SML estão resumidos na Fig. 7.28.

Os últimos dois dígitos de uma instrução da SML são os *operandos* – o endereço da posição da memória que contém a palavra à qual a operação se aplica. Vamos considerar vários programas simples em SML.

O primeiro programa em SML (Fig. 7.29) lê dois números do teclado e calcula e imprime sua soma. A instrução +1007 lê o primeiro número do teclado e o coloca na posição 07 (que foi inicializada com zero). Então, a instrução +1008 lê o próximo número para a posição 08. A instrução *load*, +2007, coloca o primeiro número no acumulador e a instrução *add*, +3008, adiciona o segundo número ao número no acumulador. *Todas as instruções aritméticas da SML deixam seus resultados no acumulador*. A instrução *store*, +2109, coloca o resultado de volta na posição 09 da memória de onde a instrução *write*, +1109, pega o número e o imprime (como um número decimal de quatro dígitos com sinal). A instrução *halt*, +4300, termina a execução.

Código de operação	Significado
Operações de entrada/saída: <code>final int READ = 10;</code> <code>final int WRITE = 11;</code>	Lê uma palavra do teclado para uma posição específica da memória. Escreve na tela uma palavra de uma posição específica da memória.
Operações de carga/armazenamento: <code>final int LOAD = 20;</code> <code>final int STORE = 21;</code>	Carrega uma palavra de uma posição específica na memória para o acumulador. Armazena uma palavra do acumulador em uma posição específica na memória.
Operações aritméticas: <code>final int ADD = 30;</code> <code>final int SUBTRACT = 31;</code> <code>final int DIVIDE = 32;</code> <code>final int MULTIPLY = 33;</code>	Adiciona uma palavra de uma posição específica na memória à palavra no acumulador (deixa o resultado no acumulador). Subtrai uma palavra de uma posição específica na memória da palavra no acumulador (deixa o resultado no acumulador). Divide a palavra que está no acumulador por uma palavra de uma posição específica na memória (deixa o resultado no acumulador). Multiplica uma palavra de uma posição específica na memória pela palavra no acumulador (deixa o resultado no acumulador).
Operações de transferência de controle: <code>final int BRANCH = 40;</code> <code>final int BRANCHNEG = 41;</code> <code>final int BRANCHZERO = 42;</code> <code>final int HALT = 43;</code>	Desvia para uma posição específica na memória. Desvia para uma posição específica na memória se o acumulador for negativo. Desvia para uma posição específica na memória se o acumulador for zero. Pára – o programa completou sua tarefa.

Fig. 7.28 Códigos de operação da Simpletron Machine Language (SML).

Posição	Número	Instrução
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Fig. 7.29 Programa em SML que lê dois inteiros e calcula sua soma.

O segundo programa em SML (Fig. 7.30) lê dois números do teclado, determina e imprime o maior valor. Observe o uso da instrução `+4107` como transferência de controle condicional, muito parecida com a instrução `if` de Java.

Agora escreva programas em SML para realizar cada uma das seguintes tarefas.

- Utilizar um laço controlado por sentinelas para ler 10 números positivos. Calcular e imprimir sua soma.
- Utilizar um laço controlado por contador para ler sete números, alguns positivos e alguns negativos, e calcular e imprimir sua média.
- Ler uma série de números e determinar e imprimir o maior número. O primeiro número lido indica quantos números devem ser processados.

Posição	Número	Instrução
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Fig. 7.30 Programa em SML que lê dois inteiros e determina qual é o maior.

7.43 (*Um simulador de computador*) À primeira vista, pode parecer loucura, mas nesse problema você vai construir seu próprio computador. Não, você não irá soldar componentes. Em vez disso, você utilizará a poderosa técnica de *simulação baseada em software* para criar um *modelo em software* orientado a objetos do Simpletron. Você não se decepcionará. O simulador de Simpletron transformará o computador que você está utilizando em um Simpletron e você realmente será capaz de executar, testar e depurar os programas de SML escritos no Exercício 7.42. O Simpletron será um *applet* baseado em eventos – você clicará em um botão para executar cada instrução de SML e será capaz de ver a instrução “em ação”.

Quando você executa o simulador de Simpletron, ele deve começar exibindo:

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction      ***
*** (or data word) at a time into the input        ***
*** text field. I will display the location       ***
*** number and a question mark (?). You then      ***
*** type the word for that location. Press the     ***
*** Done button to stop entering your program.    ***
```

[Bem-vindo ao Simpletron! Por favor insira em seu programa uma instrução (ou palavra de dados) por vez no campo de texto de entrada. Exibiréi o número da posição e um ponto de interrogação (?). Digite então a palavra para essa posição. Pressione o botão **Done** para interromper a digitação do seu programa.]

O programa deve exibir um **JTextField** **input** em que o usuário digitará cada instrução, uma por vez, e um botão **Done** para o usuário clicar quando o programa completo de SML tiver sido digitado. Simule a memória do Simpletron com um array unidimensional **memory** que tem 100 elementos. Agora suponha que o simulador esteja sendo executado e vamos examinar o diálogo à medida que inserimos o programa da Fig. 7.30 (Exercício 7.42):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
```

```

05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000

```

O programa deve utilizar um **JTextField** para exibir a posição da memória seguida por um ponto de interrogação. Cada um dos valores à direita de um ponto de interrogação é digitado pelo usuário no **JTextField input**. Quando se clicar no botão **Done**, o programa deve exibir:

```

*** Program loading completed ***
*** Program execution begins ***

```

O programa de SML agora foi colocado (ou carregado) no array **memory**. O Simpletron deve fornecer um botão “**Execute next instruction**” em que o usuário pode clicar para executar cada instrução de seu programa em SML. A execução inicia com a instrução na posição **00** e, como Java, continua seqüencialmente, a menos que seja desviada para alguma outra parte do programa por uma transferência de controle.

Utilize a variável **accumulator** para representar o registrador acumulador. Utilize a variável **instructionCounter** para monitorar a posição na memória que contém a instrução que está sendo executada. Utilize a variável **operationCode** para indicar a operação que está sendo atualmente executada (isto é, os dois dígitos da esquerda da palavra de instrução). Utilize a variável **operand** para indicar a posição da memória sobre a qual a instrução atual opera. Portanto, **operand** são os dois dígitos mais à direita da instrução que está sendo atualmente executada. Não execute instruções diretamente de memória. Mais precisamente, transfira a próxima instrução que será realizada da memória para uma variável chamada **instructionRegister**. Então “pegue” os dois dígitos da esquerda e os coloque em **operationCode** e “pegue” os dois dígitos direitos e coloque-os em **operand**. Cada um dos registradores precedentes devem ter um correspondente **JTextField** em que seu valor atual pode ser exibido o tempo todo. Quando o Simpletron começa a execução, os registradores especiais são todos inicializados com 0.

Agora vamos “percorrer” a execução da primeira instrução de SML, **+1009** na posição da memória **00**. Isso se chama *ciclo de execução de instrução*.

O **instructionCounter** nos informa a posição da próxima instrução que será executada. Realizamos uma *busca (fetch)* do conteúdo dessa posição a partir de **memory** utilizando a instrução Java

```
instructionRegister = memory[ instructionCounter ];
```

O código de operação e o operando são extraídos do registrador de instrução pelas instruções

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Agora o Simpletron deve determinar que o código de operação é, na verdade, uma instrução *read* (e não uma *write*, uma *load*, etc.). A estrutura **switch** diferencia entre as 12 operações de SML.

Na estrutura **switch**, o comportamento de várias instruções de SML é simulado como mostrado na Fig. 7.31. Discutimos as instruções de desvio brevemente e deixamos as outras para o leitor.

Quando o programa de SML completar a execução, o nome e o conteúdo de cada registrador bem como o conteúdo completo da memória devem ser exibidos. Esse tipo de impressão é freqüentemente chamado de *dump de memória*. Para ajudá-lo a programar seu método de *dump*, um exemplo de formato de *dump* é mostrado na Fig. 7.32. Observe que um *dump*, depois de executar um programa Simpletron, mostra os valores reais das instruções e os valores dos dados no momento em que a execução terminou. O exemplo de *dump* supõe que a saída será enviada para a tela do vídeo com uma série de chamadas para os métodos **System.out.print** e **System.out.println**. Entretanto, incentivamo-lo a fazer experiência com uma versão que possa ser exibida no *applet* com **JTextArea** ou com um array de objetos **JTextField**.

Vamos prosseguir com a execução da primeira instrução de nosso programa, a saber, o **+1009** na posição **00**. Como indicamos, a instrução **switch** simula isso pedindo que o usuário digite um valor no diálogo de entrada, lendo o valor, convertendo o valor em um inteiro e armazenando-o na posição de memória **memory[operand]**. Uma vez que seu Simpletron se baseie em eventos, espere o usuário digitar um valor no **JTextField input** e pressionar a tecla *Enter*. O valor então é lido para a posição **09**.

Nesse ponto, a simulação da primeira instrução é concluída. Tudo que resta é preparar o Simpletron para executar a próxima instrução. Como a instrução recém-executada não foi uma transferência de controle, precisamos meramente incrementar o registro do contador de instruções como segue:

```
++instructionCounter;
```

Instrução	Descrição
<i>read</i> :	Exibe um diálogo de entrada com o <i>prompt</i> "Enter an integer". Converte o valor digitado em um inteiro e armazena-o na posição <code>memory[operand]</code> .
<i>load</i> :	<code>accumulator = memory[operand];</code>
<i>add</i> :	<code>accumulator += memory[operand];</code>

Fig. 7.31 Comportamento de diversas instruções SML no Simpletron.

REGISTERS:										
<code>accumulator</code>										
	+0000									
<code>instructionCounter</code>										
	00									
<code>instructionRegister</code>										
	+0000									
<code>operationCode</code>										
	00									
<code>operand</code>										
	00									
MEMORY:										
0	1	2	3	4	5	6	7	8	9	
0 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	
10 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	
20 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	
30 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	
40 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	
50 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	
60 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	
70 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	
80 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	
90 +0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	

Fig. 7.32 Um exemplo de *dump*.

Isso completa a execução simulada da primeira instrução. Quando o usuário clica no botão **Execute next instruction**, o processo inteiro (isto é, o ciclo de execução de instrução) inicia novamente com a busca da próxima instrução a ser executada.

Agora vamos analisar como as instruções de desvio – as transferências de controle – são simuladas. Tudo que precisamos fazer é ajustar o valor no contador de instrução apropriadamente. Portanto, a instrução de desvio incondicional (40) é simulada dentro do `switch` como

```
instructionCounter = operand;
```

A instrução condicional “desvie se acumulador for zero” é simulada como

```
if ( accumulator == 0 )
    instructionCounter = operand;
```

Nesse ponto, você deve implementar o simulador de Simpletron e executar cada um dos programas de SML que você escreveu no Exercício 7.42. Você pode sofisticar a SML com recursos adicionais e oferecer-ló no seu simulador.

O simulador deve verificar vários tipos de erros. Durante a fase de carga do programa, por exemplo, cada número que o usuário digita na memória do Simpletron deve estar no intervalo de -9999 a +9999. O simulador deve testar se cada número digitado está nesse intervalo, e, se não estiver, continuar solicitando que o usuário digite novamente o número até que o usuário digite um número correto.

Durante a fase de execução, o simulador deve verificar vários erros sérios, como tentativas de divisão por zero, tentativas de execução de códigos de operação inválidos, estouros de acumulador (isto é, operações aritméticas que resultam em valores maiores que +9999 ou menores que -9999) e assim por diante. Os erros sérios são chamados de *erros fatais*. Quando um erro fatal é detectado, o simulador deve imprimir uma mensagem de erro como:

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

e deve imprimir um *dump* de computador completo no formato discutido anteriormente. Isso ajudará o usuário a localizar o erro no programa.

7.44 (*Modificações para o simulador de Simpletron*) No Exercício 7.43, você escreveu uma simulação em *software* de um computador que executa programas escritos na Simpletron Machine Language (SML). Nesse exercício, são propostas várias modificações e aprimoramentos para o simulador de Simpletron. Nos Exercícios 19.26 e 19.27, propomos a construção de um compilador que converte os programas escritos em uma linguagem de programação de alto nível (uma variação do Basic) para a Simpletron Machine Language. Algumas das seguintes modificações e melhorias podem ser necessárias para executar os programas produzidos pelo compilador.

- Estender a memória do Simulador de Simpletron para conter 1000 posições da memória, a fim de permitir que o Simpletron trate programas maiores.
- Permitir que o simulador execute cálculos em módulo. Isso exige uma instrução adicional na Simpletron Machine Language.
- Permitir que o simulador execute cálculos de potenciação. Isso exige uma instrução adicional na Simpletron Machine Language.
- Modificar o simulador para utilizar valores hexadecimais em vez de valores inteiros para representar instruções da Simpletron Machine Language.
- Modificar o simulador para permitir enviar para a saída um caractere nova linha. Isso exige uma instrução adicional na Simpletron Machine Language.
- Modificar o simulador para processar valores de ponto flutuante além de valores inteiros.
- Modificar o simulador para tratar entrada de *strings*. [Dica: cada palavra do Simpletron pode ser dividida em dois grupos, cada um armazenando um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o equivalente ASCII decimal de um caractere. Adicione uma instrução de linguagem de máquina que lerá um *string* e armazenará o *string* a partir de uma posição de memória específica do Simpletron. A primeira metade da palavra nessa posição será uma contagem do número de caracteres no *string* (isto é, o comprimento do *string*). Cada meia palavra seguinte contém um caractere ASCII expresso como dois dígitos decimais. A instrução de linguagem de máquina converte cada caractere em seu equivalente ASCII e o atribui a uma “meia-palavra”.]
- Modificar o simulador para tratar saída de *strings* armazenados no formato da parte g). [Dica: adicione uma instrução de linguagem de máquina que imprimirá um *string* iniciando em uma certa posição da memória do Simpletron. A primeira metade da palavra nessa posição é uma contagem do número de caracteres do *string* (isto é, o comprimento do *string*). Cada meia palavra seguinte contém um caractere de ASCII expresso como dois dígitos decimais. A instrução de linguagem de máquina verifica o comprimento e imprime o *string* traduzindo cada número de dois dígitos para seu caractere equivalente.]

7.45 A série de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inicia com os termos 0 e 1 e tem a propriedade de que cada termo sucessivo é a soma dos dois termos precedentes.

- Escreva um método `fibonacci(n)` não-recursivo que calcula o n -ésimo número de Fibonacci. Incorpore esse método a um *applet* que permita ao usuário digitar o valor de n .
- Determine o maior número de Fibonacci que pode ser impresso no sistema.
- Modifique o programa da parte a) para utilizar `double` em vez de `int` para calcular e retornar números de Fibonacci e utilize esse programa modificado para repetir a parte b).

8

Programação baseada em objetos

Objetivos

- Entender encapsulamento e ocultamento de dados.
- Entender as noções de abstração de dados e tipos abstratos de dados (ADT).
- Criar ADTs Java, isto é, classes.
- Ser capaz de criar, utilizar e destruir objetos.
- Ser capaz de controlar o acesso a variáveis de instância e métodos de objetos.
- Apreciar o valor da orientação a objetos.
- Entender o uso da referência `this`.
- Entender variáveis de classes e métodos de classe.

My object all sublime

I shall achieve in time.

W. S. Gilbert

*É este um mundo em que devemos esconder nossas
virtudes?*

William Shakespeare, *Twelfth Night*

Your public servants serve you right.

Adlai Stevenson

*But what, to serve our private ends,
Forbids the cheating of our friends?*

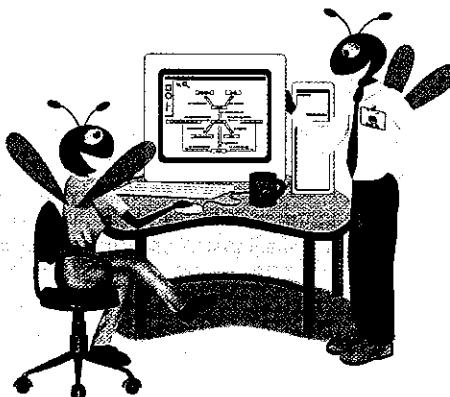
Charles Churchill

Sobretudo sê fiel e verdadeiro contigo mesmo.

William Shakespeare, *Hamlet*

Não tenha amigos não iguais a você.

Confúcio



Sumário do capítulo

- 8.1 Introdução
- 8.2 Implementando um tipo abstrato de dados *Time* com uma classe
- 8.3 Escopo de classe
- 8.4 Controlando o acesso a membros
- 8.5 Criando pacotes
- 8.6 Inicializando objetos de classe: construtores
- 8.7 Utilizando construtores sobrecarregados
- 8.8 Utilizando os métodos *set* e *get*
 - 8.8.1 Executando um *applet* que usa pacotes definidos pelo programador
- 8.9 Reutilização de *software*
- 8.10 Variáveis de instância final
- 8.11 Composição: objetos como variáveis de instância de outras classes
- 8.12 Acesso de pacote
- 8.13 Utilizando a referência *this*
- 8.14 Finalizadores
- 8.15 Membros de classe static
- 8.16 Abstração de dados e encapsulamento
 - 8.16.1 Exemplo: tipo abstrato de dados *Fila*
- 8.17 (Estudo de caso opcional) Pensando em objetos: começando a programar as classes para a simulação do elevador

Resumo • *Terminologia* • *Exercícios de auto-revisão* • *Respostas aos exercícios de auto-revisão*
• *Exercícios*

8.1 Introdução

Agora, investigamos a orientação a objetos em Java em maior profundidade. Por que adiamos isso até agora? Primeiro, os objetos que construiremos serão compostos, em parte, por fragmentos de programas estruturados, portanto precisávamos estabelecer uma base em programação estruturada com estruturas de controle. Segundo, nossa intenção era estudar métodos em profundidade. Terceiro, pretendíamos familiarizar o leitor com *arrays* que são objetos Java.

Através de nossas discussões sobre programas Java orientados a objetos nos Capítulos 2 a 7, apresentamos muitos conceitos básicos (isto é, “pensar em objetos”) e terminologia (isto é, “falar em objetos”) de programação orientada a objetos em Java. Além disso, discutimos nossa metodologia de desenvolvimento de programas: analisamos muitos problemas típicos que exigiram que um programa – um *applet* Java ou um aplicativo Java – fosse construído, determinamos as classes da Java API necessárias para implementar o programa, determinamos as variáveis de instância necessárias, determinamos os métodos necessários e especificamos como um objeto de nossa classe colaborava com objetos de classes da Java API para atingir os objetivos globais do programa.

Vamos revisar resumidamente alguns conceitos fundamentais e a terminologia de orientação a objeto. A OOP *encapsula* dados (*atributos*) e métodos (*comportamentos*) em *objetos*; os dados e métodos de um objeto estão intimamente amarrados entre si. Os objetos têm a propriedade de *ocultar informações*. Significa que, embora os objetos possam saber se comunicar uns com os outros através de *interfaces* bem-definidas, os objetos normalmente não têm permissão para conhecer como os outros objetos são implementados – os detalhes da implementação ficam escondidos dentro dos próprios objetos. Seguramente é possível dirigir bem um carro sem conhecer os detalhes sobre como os sistemas de motores, transmissões e escapamento funcionam internamente. Veremos por que o ocultamento de informações é tão crucial para uma boa engenharia de *software*.

Em C e em outras *linguagens de programação procedurais*, a programação tende ser *orientada a ações*. Em Java, a programação é *orientada a objetos*. Em C, a unidade de programação é a *função* (chamada de *método* em Java). Em Java, a unidade de programação é a *classe* a partir da qual os objetos são *instanciados* (isto é, criados) em algum momento. As funções não desapareceram em Java; em vez disso, elas estão encapsuladas como métodos, com os dados que elas processam, dentro das “paredes” das classes.

Os programadores de C concentram-se em escrever funções. Os grupos de ações que realizam alguma tarefa são usados para formar funções, e as funções são agrupadas para formar programas. Os dados são certamente importantes em C, mas a idéia é que os dados existem principalmente para suportar as ações que as funções realizam. Os *verbos* em um documento de requisitos de sistema ajudam o programador de C a determinar o conjunto de funções que trabalharão juntas para implementar o sistema.

Os programadores de Java concentram-se em criar seus próprios *tipos definidos pelo usuário*, chamados de *classes*. As classes também são conhecidas como *tipos definidos pelo programador*. Cada classe contém dados e o conjunto de métodos que manipulam os dados. Os componentes de dados de uma classe são chamados de *variáveis de instância* (estas são chamadas de *membros de dados* em C++). Assim como uma instância de um tipo predefinido como `int` é chamado de *variável*, a instância de um tipo definido pelo usuário (isto é, uma classe) é chamada de *objeto*. Em Java, o foco da atenção está nos objetos e não nos métodos. Os *substantivos* em um documento de requisitos de sistema ajudam o programador de Java a determinar um conjunto inicial de classes com o qual começar o processo do projeto. Essas classes são então utilizadas para instanciar objetos que trabalharão juntos para implementar o sistema.

Este capítulo explica como criar e utilizar objetos, um assunto denominado de *programação baseada em objetos* (*OBP – object-based programming*). O Capítulo 9 apresenta *herança* e *polimorfismo* – duas tecnologias fundamentais que permitem a verdadeira *programação orientada a objetos* (*OOP – object-oriented programming*). Embora a herança só seja discutida em detalhes no Capítulo 9, ela faz parte de cada definição de classe Java.



Dica de desempenho 8.1

Quando se passa um objeto para um método em Java, só uma referência para o objeto é passada, não uma cópia de um objeto possivelmente grande (como é o caso em uma passagem por valor).



Observação de engenharia de software 8.1

É importante escrever programas que sejam compreensíveis e fáceis de manter. A mudança é a regra, e não a exceção. Os programadores devem prever que o código será modificado. Como veremos, as classes facilitam a modificação do programa.

8.2 Implementando um tipo abstrato de dados Time com uma classe

O próximo exemplo consiste em duas classes – `Time1` (Fig. 8.1) e `TimeTest` (Fig. 8.2). A classe `Time1` é definida no arquivo `Time1.java`. A classe `TimeTest` é definida em um arquivo separado, denominado `TimeTest.java`. É importante observar que essas classes *devem* ser definidas em arquivos separados, porque as duas são classes [Nota: a saída produzida por este programa aparece na Fig. 8.2.].

```

1 // Fig. 8.1: Time1.java
2 // Definição da classe Time1
3
4 // Pacotes do núcleo de Java
5 import java.text.DecimalFormat;
6
7 public class Time1 extends Object {
8     private int hour;      // 0 - 23
9     private int minute;    // 0 - 59
10    private int second;   // 0 - 59
11
12    // O construtor Time1 inicializa cada variável de instância
13    // com zero. Assegura que cada objeto Time1 inicia em um
14    // estado consistente.

```

Fig. 8.1 Implementação do tipo abstrato de dados `Time1` como uma classe (parte 1 de 2).

```

15  public Time1()
16  {
17      setTime( 0, 0, 0 );
18  }
19
20 // Configura um novo valor de hora usando hora universal. Verifica a
21 // validade dos dados. Configura os valores inválidos como zero.
22 public void setTime( int h, int m, int s )
23 {
24     hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
25     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
26     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
27 }
28
29 // Converte para String no formato de hora padrão
30 public String toUniversalString()
31 {
32     DecimalFormat twoDigits = new DecimalFormat( "00" );
33
34     return twoDigits.format( hour ) + ":" +
35         twoDigits.format( minute ) + ":" +
36         twoDigits.format( second );
37 }
38
39 // Converte para String no formato de hora padrão
40 public String toString()
41 {
42     DecimalFormat twoDigits = new DecimalFormat( "00" );
43
44     return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
45         ":" + twoDigits.format( minute ) +
46         ":" + twoDigits.format( second ) +
47         ( hour < 12 ? " AM" : " PM" );
48 }
49
50 } // fim da classe Time1

```

Fig. 8.1 Implementação do tipo abstrato de dados **Time1** como uma classe (parte 2 de 2).



Observação de engenharia de software 8.2

As definições de classe que iniciam com a palavra-chave **public** devem ser armazenadas em um arquivo que tenha exatamente o mesmo nome que a classe e termine com a extensão de nome de arquivo **.java**.



Erro comum de programação 8.1

Definir mais de uma classe **public** no mesmo arquivo é um erro de sintaxe.

A Fig. 8.1 contém uma definição simples para a classe **Time1**. Nossa definição de classe **Time1** inicia com a linha 6, que indica que a classe **Time1** estende (**extends**) a classe **Object** (do pacote **java.lang**). Lembre-se de que você nunca cria uma definição de classe “a partir do zero”. Na verdade, quando você cria uma definição de classe, você sempre utiliza fragmentos de uma definição de classe existente. Java utiliza a *herança* para criar novas classes a partir de definições de classes existentes. A palavra-chave **extends** seguida do nome de classe **Object** indica a classe (nesse caso **Object**) da qual nossa nova classe herda pedaços existentes. Nesse relacionamento de herança, **Object** chama-se *superclasse* ou *classe básica* e **Time1** chama-se *subclasse* ou *classe derivada*. Utilizar herança resulta em uma nova definição de classe que tem os *atributos* (dados) e *comportamentos* (métodos) da classe **Object**, além de novos recursos que adicionamos em nossa definição de classe **Time1**. Todas as classes em Java são (direta ou indiretamente) subclasses de **Object**. Portanto, cada classe herda os 11 métodos definidos pela classe **Object**. O método-chave de **Object** é **toString**, discutido mais adiante nesta seção. Ou-

tros métodos da classe `Object` são discutidos conforme necessário ao longo de todo o texto. Para ver uma lista completa dos métodos da classe `Object`, veja a documentação *on-line* da API em

java.sun.com/j2se/1.3/docs/api/index.html



Observação de engenharia de software 8.3

Todas as classes definidas em Java devem estender outra classe. Se uma classe não utiliza explicitamente a palavra-chave `extends` em sua definição, a classe implicitamente estende (`extends`) `Object`.

O corpo da definição de classe é delineado com as chaves esquerda e direita (`{` e `}`) nas linhas 7 e 50. A classe `Time1` contém três variáveis de instância do tipo inteiro – `hour`, `minute` e `second` – que representam a hora no formato de *hora universal* (formato de *relógio de 24 horas*).

As palavras-chave `public` e `private` são *modificadores de acesso a membros*. As variáveis de instância ou os métodos declarados com o modificador de acesso a membros `public` são acessíveis onde quer que o programa tenha uma referência a um objeto `Time1`. As variáveis de instância ou os métodos declarados com modificador de acesso a membros `private` são acessíveis *somente* aos métodos da classe na qual eles são definidos. Cada variável de instância ou definição de método deve ser precedida por um modificador de acesso a membro.

As três variáveis de instância do tipo inteiro `hour`, `minute` e `second` são declaradas (linhas 8 a 10) com o modificador de acesso a membros `private`, o que indica que as variáveis de instância dessa classe somente são acessíveis aos métodos da classe. Quando um programa cria (instancia) um objeto da classe, tais variáveis de instância são encapsuladas no objeto e podem ser acessadas somente através de métodos da classe daquele objeto (normalmente pelos métodos `public` da classe). As variáveis de instância normalmente são declaradas `private` e os métodos normalmente são declarados `public`. É possível ter métodos `private` e dados `public`, como veremos mais tarde. Os métodos `private` chamam-se *métodos utilitários* ou *métodos auxiliares*, uma vez que só podem ser chamados por outros métodos daquela classe e são utilizados para suportar a operação desses métodos. Utilizar dados `public` é incomum e é uma prática de programação arriscada.



Observação de engenharia de software 8.4

Os métodos tendem a dividir-se em várias categorias diferentes: os métodos que obtêm os valores de variáveis de instância `private`; os métodos que configuram valores de variáveis de instância `private`; os métodos que implementam serviços da classe; e os métodos que realizam várias tarefas mecânicas para a classe, como inicializar objetos de classe, atribuir objetos de classe e converter entre classes e tipos primitivos ou entre classes e outras classes.

Os métodos de acesso podem ler ou exibir dados. Outra utilização comum dos métodos de acesso é testar a verdade ou a falsidade de condições – esses métodos geralmente se chamam *métodos de predicado*. Um exemplo de um método de predicado seria um método `isEmpty` para qualquer classe de contêiner – uma classe capaz de armazenar muitos objetos – como uma lista encadeada, uma pilha ou uma fila (essas estruturas de dados são discutidas em profundidade nos Capítulos 19, 20 e 21). O programa poderia testar `isEmpty` antes de tentar ler outro item do objeto contêiner. O programa poderia testar `isFull` antes de tentar inserir outro item em um objeto de contêiner.

A classe `Time1` contém os seguintes métodos `public` – `Time1` (linhas 15 a 18), `setTime` (linhas 22 a 27), `toUniversalString` (linhas 30 a 37) e `toString` (linhas 40 a 48). Esses são os *métodos public*, *serviços public* ou *interface public* da classe. Esses métodos são utilizados por *clientes* da classe (isto é, partes de um programa que são usuários de uma classe) para manipular os dados armazenados em objetos da classe.

Os clientes de uma classe utilizam referências para interagir com um objeto da classe. Por exemplo, o método `paint` em um *applet* é um cliente da classe `Graphics`. O método `paint` utiliza uma referência a um objeto `Graphics` (como `g`) que ele recebe como argumento para desenhar no *applet* chamando métodos que são os serviços `public` da classe `Graphics` (como `drawString`, `drawLine`, `drawOval` e `drawRect`).

Repare no método com o mesmo nome que a classe (linhas 15 a 18); ele é o método *construtor* dessa classe. O construtor é um método especial que inicializa as variáveis de instância de um objeto de classe. Java chama o método construtor de uma classe quando o programa instancia um objeto dessa classe. Nesse exemplo, o construtor simplesmente chama o método `setTime` da classe (discutido em breve), com os valores de hora, minuto e segundos especificados como 0.

É comum ter vários construtores para uma classe; isso é realizado por *sobre carga de método* (como veremos na Fig. 8.6). Os construtores podem receber argumentos, mas não podem especificar um tipo de dado devolvido.

Uma diferença importante entre os construtores e outros métodos é que os construtores *não têm permissão para especificar um tipo de dados de retorno* (nem mesmo `void`). Normalmente, os construtores são métodos `public` de uma classe. Os métodos `não-public` são discutidos mais tarde.



Erro comum de programação 8.2

Tentar declarar um tipo de retorno para um construtor e/ou tentar devolver um valor a partir de um construtor é um erro de lógica. Java permite que outros métodos da classe tenham o mesmo nome que a classe e especifiquem o tipo de retorno. Tais métodos não são construtores e não serão chamados quando um objeto da classe for instanciado.

O método `setTime` (linhas 22 a 27) é um método `public` que recebe três argumentos do tipo inteiro e os utiliza para configurar a hora. Cada argumento é testado em uma expressão condicional que determina se o valor está num intervalo válido. Por exemplo, o valor `hour` deve ser maior que ou igual a 0 e menor que 24, porque representamos a hora em formato de hora universal (0–23 para a hora, 0–59 para o minuto e 0–59 para o segundo). Qualquer valor fora desse intervalo é um valor inválido e é configurado como zero – assegurando que um objeto `Time1` sempre contenha dados válidos. Isso também é conhecido como *manter o objeto em um estado consistente* ou manter a integridade do objeto. Nos casos em que os dados inválidos são fornecidos para `setTime`, o programa pode querer indicar que uma configuração de hora inválida foi tentada. Exploramos essa possibilidade nos exercícios.



Boa prática de programação 8.1

Sempre defina uma classe de modo que suas variáveis de instância sejam mantidas em um estado consistente.

O método `toUniversalString` (linhas 30 a 37) não recebe nenhum argumento e retorna um `String`. Esse método produz um *string* no formato de hora universal que consiste em seis dígitos – dois para a hora, dois para o minuto e dois para o segundo. Por exemplo, 13:30:07 representa 1:30:07 p.m. A linha 32 cria uma instância da classe `DecimalFormat` (do pacote `java.text` importado na linha 3) para ajudar a formatar a hora universal. O objeto `twoDigits` é inicializado com o *string de controle de formato* "00", que indica que o formato de número deve consistir em dois dígitos – cada 0 é um marcador de lugar para um dígito. Se o número que está formatado for constituído de um único dígito, ele é automaticamente precedido por um 0 inicial (isto é, 8 é formatado como 08). A instrução `return` nas linhas 34 a 36 utiliza o método `format` (que retorna um `String` formatado que contém o número) do objeto `twoDigits` para formatar os valores `hour`, `minute` e `second` em *strings* de dois dígitos. Esses *strings* são concatenados com o operador + (separadas por dois-pontos) e devolvidos do método `toUniversalString`.

O método `toString` (linhas 40 a 48) não recebe nenhum argumento e retorna um `String`. Esse método produz um *string* no formato de hora padrão que consiste em valores `hour`, `minute` e `second` separados por dois-pontos e um indicador AM ou PM, como em 1:27:06 PM. Esse método utiliza as mesmas técnicas `DecimalFormat` que o método `toUniversalString` para garantir que cada um dos valores `minute` e `second` apareça com dois dígitos. O método `toString` é especial no sentido de que herdamos da classe `Object` um método `toString` com exatamente a mesma primeira linha que nosso `toString` na linha 40. O método `toString` original da classe `Object` é uma versão genérica que é utilizada principalmente como marcador de lugar que pode ser redefinido por uma subclasse (de maneira semelhante aos métodos `init`, `start` e `paint` da classe `JApplet`). Nossa versão substitui a versão que herdamos para fornecer um método `toString` que é mais apropriado para nossa classe. Isso é conhecido como *sobrescrever (override)* a definição original de método (discutido em detalhes no Capítulo 9).

Uma vez que a classe foi definida, ela pode ser utilizada como um tipo em declarações, como

```
Time1 sunset,           // referência a um objeto do tipo Time1
timeArray[],           // referência a um array de objetos Time1
```

O nome da classe é um novo especificador de tipo. Pode haver muitos objetos de uma classe, da mesma forma que pode haver muitas variáveis de um tipo primitivo de dados como `int`. O programador pode criar novos tipos de classe conforme necessário; essa é uma das razões pelas quais Java é conhecida como uma *linguagem extensível*.

O aplicativo **TimeTest1** da Fig. 8.2 utiliza a classe **Time1**. O método **main** da classe **TimeTest1** declara e inicializa uma instância da classe **Time1** chamada **time** na linha 12. Quando o objeto é instanciado, o operador **new** aloca a memória na qual o objeto **Time1** será armazenado, depois **new** chama o construtor **Time1** para inicializar as variáveis de instância do novo objeto **Time1**. O construtor invoca o método **setTime** para inicializar explicitamente cada variável de instância privada com 0. O operador **new** então retorna uma referência para o novo objeto e essa referência é atribuída a **time**. De maneira semelhante, a linha 32 na classe **Time1** (Fig. 8.1) utiliza **new** para alocar a memória para um objeto **DecimalFormat**, depois chama o construtor **DecimalFormat** com o argumento "00" para indicar o *string* de controle de formato de número.



Observação de engenharia de software 8.5

Toda vez que new cria um objeto de uma classe, o construtor dessa classe é chamado para inicializar as variáveis de instância do novo objeto.

Observe que a classe **Time1** não foi **importada** no arquivo **TimeTest1.java**. Na verdade, cada classe em Java faz parte de um *pacote* (como as classes da Java API). Se o programador não especifica o pacote para uma classe, a classe automaticamente é colocada no *pacote default*, que inclui as classes compiladas no diretório atual. Se uma classe está no mesmo pacote que a classe que a utiliza, não é necessária uma instrução **import**. Importamos classes da Java API porque seus arquivos **.class** não estão no mesmo pacote com cada programa que escrevemos. A Seção 8.5 ilustra como definir seus próprios pacotes de classes para reutilizar.

```

1 // Fig. 8.2: TimeTest1.java
2 // Classe TimeTest1 para exercitar a classe Time1
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 public class TimeTest {
8
9     // cria objeto Time1 e o manipula
10    public static void main( String args[] )
11    {
12        Time1 time = new Time1();    // chama o construtor Time1
13
14        // acrescenta a versão String da hora ao string de saída
15        String output = "The initial universal time is: " +
16            time.toUniversalString() +
17            "\nThe initial standard time is: " + time.toString() +
18            "\nImplicit toString() call: " + time;
19
20        // muda a hora e acrescenta a versão String da hora a output
21        time.setTime( 13, 27, 6 );
22        output += "\n\nUniversal time after setTime is: " +
23            time.toUniversalString() +
24            "\nStandard time after setTime is: " + time.toString();
25
26        // usa valores inválidos para mudar a hora e acrescenta
27        // versão String da hora a output
28        time.setTime( 99, 99, 99 );
29        output += "\n\nAfter attempting invalid settings: " +
30            "\nUniversal time: " + time.toUniversalString() +
31            "\nStandard time: " + time.toString();
32
33        JOptionPane.showMessageDialog( null, output,
34            "Testing Class Time1",
35            JOptionPane.INFORMATION_MESSAGE );

```

Fig. 8.2 Usando um objeto da classe **Time1** em um programa (parte 1 de 2).

```

36
37     System.exit( 0 );
38 }
39
40 } // fim da classe TimeTest1

```

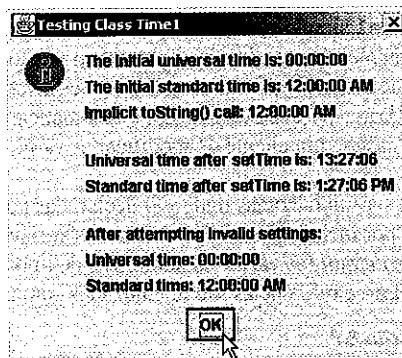


Fig. 8.2 Usando um objeto da classe `Time1` em um programa (parte 2 de 2).

A instrução nas linhas 15 a 18 define uma referência a `String output` que armazena o `string` que contém os resultados que serão exibidos em uma caixa de mensagem. Inicialmente, o programa atribui a `output` a hora no formato de hora universal (enviando a mensagem `toUniversalString` para o objeto ao qual `time` faz referência) e no formato de hora padrão (enviando a mensagem `toString` para o objeto ao qual `time` faz referência) para confirmar que os dados foram inicializados adequadamente. Repare que a linha 18 utiliza um recurso especial de concatenação de `strings` de Java. Concatenar um `String` com qualquer objeto resulta em uma chamada implícita ao método `toString` do objeto para converter o objeto em um `String`; assim, os `Strings` são concatenados. As linhas 17 e 18 mostram que você pode chamar `toString` tanto explícita quanto implicitamente em uma operação de concatenação de `String`.

A linha 21 envia a mensagem `setTime` para o objeto ao qual `t` faz referência para mudar a hora. A seguir, as linhas 22 a 24 novamente acrescentam a hora a `output` em ambos os formatos para confirmar que a hora foi configurada corretamente.

Para ilustrar que o método `setTime` valida os valores passados a ele, a linha 28 chama o método `setTime` e tenta configurar as variáveis de instância com valores inválidos. Então as linhas 29 a 31 acrescentam novamente a hora a `output` em ambos os formatos para confirmar que `setTime` validou os dados. As linhas 33 a 35 exibem uma caixa de mensagem com os resultados de nosso programa. Repare nas duas últimas linhas da janela de saída que a hora está configurada como meia-noite – o valor `default` de um objeto `Time1`.

Agora que vimos nossa primeira classe que não é um aplicativo nem um *applet*, vamos analisar diversos aspectos do projeto de classes.

Novamente, observe que as variáveis de instância `hour`, `minute` e `second` são todas declaradas `private`. As variáveis de instância declaradas `private` não estão acessíveis fora da classe em que são definidas. A filosofia aqui é que a representação real de dados utilizada dentro da classe não tem nenhuma importância para os clientes de classe. Por exemplo, seria perfeitamente razoável para a classe representar a hora internamente como o número de segundos decorridos desde meia-noite. Os clientes poderiam utilizar os mesmos métodos `public` e obter os mesmos resultados sem estarem cientes disso. Nesse sentido, dizemos que a implementação de uma classe é *oculta* de seus clientes. O Exercício 8.18 pede para você fazer precisamente essa modificação na classe `Time1` da Fig. 8.1 e mostrar que não há alteração visível para os clientes da classe.

Observação de engenharia de software 8.6



O ocultamento de informações facilita a modificação de programas e simplifica a percepção de uma classe por parte de cliente. O cliente não deve precisar conhecer a implementação de uma classe (conhecido como conhecimento da implementação) para ser capaz de reutilizar uma classe.



Observação de engenharia de software 8.7

Os clientes de uma classe podem (e devem) utilizar a classe sem conhecer os detalhes internos de como a classe é implementada. Se a implementação de classe é alterada (para aprimorar o desempenho, por exemplo), desde que a interface da classe permaneça constante, o código-fonte dos clientes da classe não precisa ser alterado. Isso torna muito mais fácil modificar sistemas.

Nesse programa, o construtor `Time1` simplesmente inicializa as variáveis de instância como 0 (isto é, o do horário universal equivalente a 12 AM, ou meia-noite). Isso assegura que o objeto seja criado em um *estado consistente* (isto é, todos os valores da variável de instância são válidos). Os valores inválidos não podem ser armazenados nas variáveis de instância de um objeto `Time1` porque o construtor é automaticamente chamado quando o objeto `Time1` é criado e as tentativas subsequentes de um cliente de modificar as variáveis de instância são escrutinadas pelo método `setTime`.

As variáveis de instância podem ser inicializadas onde são declaradas no corpo da classe, pelo construtor da classe ou ter valores atribuídos a elas por métodos “*set*”. Lembre-se: as variáveis de instância que não são inicializadas explicitamente recebem valores *default* (as variáveis numéricas de tipos primitivos são inicializadas com 0, `booleans` são inicializadas com `false` e referências são inicializadas com `null`).



Boa prática de programação 8.3

Inicialize as variáveis de instância de uma classe no construtor dessa classe.

Cada classe pode incluir um método *finalizador* denominado `finalize` que faz “a limpeza de término” em cada objeto de classe antes de a memória do objeto ser coletada como lixo pelo sistema. Discutiremos a coleta de lixo e os finalizadores em detalhes nas Seções 8.14 e 8.15.

É interessante que os métodos `toUniversalString` e `toString` não recebem nenhum argumento. Isso acontece porque esses métodos sabem implicitamente que eles irão manipular variáveis de instância do objeto `Time1` particular para o qual eles são invocados. Isso torna as chamadas de método mais concisas que as chamadas de função convencionais na programação procedural. Também reduz a probabilidade de se passar argumentos errados, argumentos de tipos errados e/ou a quantidade errada de argumentos, como acontece com frequência em chamadas de função em C.



Observação de engenharia de software 8.8

Freqüentemente, utilizar uma abordagem de programação orientada a objetos pode simplificar as chamadas de métodos reduzindo a quantidade de parâmetros a serem passados. Esse benefício da programação orientada a objetos deriva do fato de que o encapsulamento de métodos e as variáveis de instância dentro de um objeto dá aos métodos o direito de acessar as variáveis de instância.

As classes simplificam a programação porque o cliente (ou usuário do objeto de classe) precisa preocupar-se apenas com as operações `public` encapsuladas no objeto. Tais operações normalmente são projetadas para serem orientadas ao cliente, em vez de orientadas à implementação. Os clientes não precisam se preocupar com a implementação de uma classe. As interfaces mudam, mas menos freqüentemente que as implementações. Quando uma implementação muda, o código dependente de implementação deve ser alterado de acordo com a mudança. Ocultando a implementação, eliminamos a possibilidade de outras partes do programa se tornarem dependentes dos detalhes da implementação da classe.

Freqüentemente, as classes não precisam ser criadas “a partir do zero”. Em vez disso, elas podem ser *derivadas* de outras classes que fornecem operações que as novas classes podem utilizar ou as classes podem incluir objetos de outras classes como membros. Essa reutilização de software pode aprimorar significativamente a produtividade dos programadores. O processo de derivar novas classes de classes existentes chama-se *herança* – um recurso que distingue a programação baseada em objetos da programação orientada a objetos – e é discutido em detalhes no Capítulo 9. Incluir objetos de classe como membros de outras classes é um processo que se chama *composição* ou *agregação* e é discutido mais adiante neste capítulo.

8.3 Escopo de classe

As variáveis de instância e métodos de uma classe pertencem ao *escopo dessa classe*. Dentro do escopo de uma classe, os membros da classe estão acessíveis para todos os métodos dessa classe e podem ser chamados simplesmente pelo nome. Fora do escopo de uma classe, os membros da classe não podem ser chamados diretamente pelo nome. Aqueles membros da classe que são visíveis (como os membros **public**) podem ser acessados apenas através de um “handle” (isto é, membros do tipo primitivo de dados podem ser chamados por *nomeDaReferênciaDoObjeto.nomeDaVariávelPrimitiva* e membros-objeto podem ser chamados por *nomeDaReferênciaDoObjeto.nomeDoObjetoMembro*). Por exemplo, um programa pode determinar o número de elementos em um objeto *array* acessando o membro **public length** do *array*, como em *nomeDoArray.length*.

As variáveis definidas em um método são conhecidas somente por esse método (isto é, são variáveis locais a esse método). Dizemos que tais variáveis têm escopo de bloco. Se um método define uma variável com o mesmo nome que uma variável com escopo de classe (isto é, uma variável de instância), a variável de escopo de classe é oculta pela variável de escopo de método no escopo do método. A variável de instância oculta pode ser acessada no método precedendo seu nome com a palavra-chave **this** e o operador ponto, como em **this.nomeDeVariável**. A palavra-chave **this** é discutida na Seção 8.13.

8.4 Controlando o acesso a membros

Os modificadores de acesso a membro **public** e **private** controlam o acesso às variáveis de instância e aos métodos de uma classe (no Capítulo 9, apresentaremos o modificador de acesso adicional **protected**). Como mencionamos anteriormente, o principal propósito dos métodos **public** é apresentar aos clientes da classe uma visualização dos *serviços* que a classe fornece (isto é, a interface pública da classe). Os clientes da classe não precisam se preocupar com a maneira como a classe realiza suas tarefas. Por essa razão, as variáveis de instância **private** e os métodos **private** de uma classe (isto é, os detalhes da implementação da classe) não são acessíveis aos clientes de uma classe. Restringir o acesso a membros de uma classe através da palavra-chave **private** é chamado de *encapsulamento*.



Erro comum de programação 8.3

Tentar acessar um membro **private** de uma classe por um método que não seja membro desta classe particular é um erro de sintaxe.

A Fig. 8.3 demonstra que os membros de classe **private** não são acessíveis por nome fora da classe. A linha 10 tenta acessar diretamente a variável de instância **private hour** do objeto **Time1** ao qual **time** faz referência. Quando esse programa é compilado, o compilador gera um erro informando que o membro **private hour** não está acessível. [Nota: esse programa pressupõe que a classe **Time1** da Fig. 8.1 é utilizada.]



Boa prática de programação 8.3

Nossa preferência é listar primeiro as variáveis de instância **private** de uma classe, de modo que, ao ler o código, você veja os nomes e tipos das variáveis de instância antes de serem utilizados nos métodos da classe.



Observação de engenharia de software 8.9

Mantenha todas as variáveis de instância de uma classe **private**. Quando necessário, forneça os métodos **public** para configurar os valores das variáveis de instância **private** e obter os valores de variáveis de instância **private**. Essa arquitetura ajuda a ocultar a implementação de uma classe de seus clientes, o que reduz defeitos e torna mais fácil fazer modificações no programa.

O acesso aos dados **private** deve ser cuidadosamente controlado pelos métodos da classe. Por exemplo, para permitir aos clientes ler o valor de dados **private**, a classe pode fornecer um método “get” (também chamado de *método de acesso*). Para permitir que os clientes modifiquem os dados **private**, a classe pode fornecer um método “set” (também chamado de *método modificador*). Essa modificação pareceria violar a noção de dados **private**, mas um método *set* pode fornecer recursos para validação de dados (como verificação de intervalo) para assegurar que o valor seja adequadamente configurado. O método *set* também pode traduzir os dados da forma utilizada na interface para a forma utilizada na implementação. O método *get* não precisa expor os dados no formato “cru”; em vez disso, o método *get* pode editar os dados e limitar a visualização dos dados que o cliente terá.



Observação de engenharia de software 8.10

Os projetistas de classe utilizam dados **private** e métodos **public** para garantir a noção do ocultamento de informações e o princípio do menor privilégio. Se o cliente de uma classe precisa acessar dados na classe, forneça acesso por métodos **public** da classe. Fazendo isso, o programador da classe controla como os dados da classe são manipulados (por exemplo, a verificação da validade dos dados pode impedir que os dados inválidos sejam armazenados em um objeto). Este é o princípio do encapsulamento de dados.

```

1 // Fig. 8.3: TimeTest2.java
2 // Demonstra erros que resultam de tentativas
3 // de acessar membros private da classe.
4 public class TimeTest2 {
5
6     public static void main( String args[] )
7     {
8         Timel time = new Timel();
9
10        time.hour = 7;
11    }
12 }
```

```

TimeTest2.java:10: hour has private access in Timel
    time.hour = 7;
               ^
1 error
```

Fig. 8.3 Tentativa incorreta de acessar membros privados da classe **Timel**.



Observação de engenharia de software 8.11

O projetista de classe não precisa fornecer os métodos **set** e/**ou** **get** para cada membro de dados **private**; esses recursos devem ser fornecidas só quando fizer sentido, e depois de o projetista da classe pensar cuidadosamente sobre isso.



Dica de teste e depuração 8.1

Tornar **private** as variáveis de instância de uma classe e **public** os métodos da classe facilita a depuração, uma vez que os problemas com manipulações de dados estão localizados nos métodos da classe.

8.5 Criando pacotes

Como vimos em quase todos os exemplos do texto, classes e *interfaces* (discutidas no Capítulo 9) de bibliotecas preexistentes, como a Java API, podem ser importadas para um programa Java. Cada classe e cada interface na Java API pertence a um pacote específico que contém um grupo de classes e interfaces relacionadas. À medida que os aplicativos se tornam mais complexos, os pacotes ajudam os programadores a administrar a complexidade dos componentes do aplicativo. Os pacotes também facilitam a reutilização de *software* permitindo que os programas importem classes de outros pacotes (como fizemos em quase todos os exemplos até este ponto). Outro benefício dos pacotes é que eles fornecem uma convenção para *nomes de classe únicos*. Com centenas de milhares de programadores de Java em todo o mundo, há uma boa probabilidade de que os nomes que você escolhe para suas classes entrem em conflito com os nomes que outros programadores escolheram para as classes deles. Esta seção mostra como criar seus próprios pacotes e discute o mecanismo de distribuição padrão para os pacotes.

O aplicativo das Figs. 8.4 e 8.5 ilustra como criar seu próprio pacote e utilizar uma classe desse pacote em um programa. Os passos para criar uma classe reutilizável são:

1. Definir uma classe **public**. Se a classe não for **public**, ela pode ser utilizada somente por outras classes no mesmo pacote.

2. Escolher um nome de pacote e adicionar uma instrução **package** ao arquivo de código-fonte para a definição da classe reutilizável. [Nota: pode haver somente uma instrução **package** em um arquivo de código-fonte Java.]
3. Compilar a classe de modo que ela seja colocada na estrutura de diretórios de pacotes apropriada.
4. Importar a classe reutilizável para dentro de um programa e utilizar a classe.

Erro comum de programação 8.4



Ocorre um erro de sintaxe se houver qualquer código antes da instrução **package** (se existir uma) em um arquivo Java.

Optamos por demonstrar o *Passo 1* modificando a classe **public Time1** definida na Fig. 8.1. A nova versão é mostrada na Fig. 8.4. Nenhuma modificação foi feita na implementação da classe, então não discutiremos novamente aqui os detalhes da implementação da classe.

Para satisfazer o *Passo 2*, adicionamos uma instrução **package** ao início do arquivo. A linha 3 utiliza uma *instrução package* para definir um **package** denominado **com.deitel.jhttp4.ch08**. Colocar uma instrução **package** no início de um arquivo-fonte Java indica que a classe definida no arquivo faz parte do pacote especificado. Os únicos tipos de instruções em Java que podem aparecer fora das chaves de uma definição de classe são as instruções **package** e as instruções **import**.

```

1 // Fig. 8.4: Time1.java
2 // Definição da classe Time1 em um pacote
3 package com.deitel.jhttp4.ch08;
4
5 // Pacotes do núcleo de Java
6 import java.text.DecimalFormat;
7
8 public class Time1 extends Object {
9     private int hour;      // 0 - 23
10    private int minute;    // 0 - 59
11    private int second;    // 0 - 59
12
13    // O construtor Time1 inicializa cada variável de instância
14    // com zero. Assegura que cada objeto Time1 inicia em um
15    // estado consistente.
16    public Time1()
17    {
18        setTime( 0, 0, 0 );
19    }
20
21    // Configura um novo valor de hora usando hora universal. Verifica a
22    // validade dos dados. Configura os valores inválidos como zero.
23    public void setTime( int h, int m, int s )
24    {
25        hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
26        minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
27        second = ( ( s >= 0 && s < 60 ) ? s : 0 );
28    }
29
30    // Converte para String no formato de hora padrão
31    public String toUniversalString()
32    {
33        DecimalFormat twoDigits = new DecimalFormat( "00" );
34
35        return twoDigits.format( hour ) + ":" +
36            twoDigits.format( minute ) + ":" +
37            twoDigits.format( second );

```

Fig. 8.4 Colocando a classe Time1 em um pacote para reutilização (parte 1 de 2).

```

38     }
39
40     // Converte para String no formato de hora padrão
41     public String toString()
42     {
43         DecimalFormat twoDigits = new DecimalFormat( "00" );
44
45         return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
46             ":" + twoDigits.format( minute ) +
47             ":" + twoDigits.format( second ) +
48             ( hour < 12 ? " AM" : " PM" );
49     }
50
51 } // fim da classe Time1

```

Fig. 8.4 Colocando a classe `Time1` em um pacote para reutilização (parte 2 de 2).



Observação de engenharia de software 8.12

Um arquivo de código-fonte Java tem a seguinte ordem: uma instrução `package` (se houver), zero ou mais instruções `import`, e então definições de classe. Somente uma das definições de classe em um dado arquivo pode ser `public`. Outras classes no arquivo também são colocadas no pacote, mas são reutilizáveis somente a partir de outras classes naquele pacote – elas não podem ser importadas para classes em um outro pacote. Elas estão no pacote para suportar a classe reutilizável no arquivo.

Em um esforço para fornecer nomes únicos para cada pacote, a Sun Microsystems especifica uma convenção para nomes de pacotes à que todos os programadores Java devem obedecer. Cada nome de pacote deve iniciar com seu nome de domínio na Internet na ordem inversa. Por exemplo, nosso nome de domínio na Internet é `deitel.com`; assim, iniciamos nosso nome de pacote com `com.deitel`. Se seu nome de domínio é `suauniversidade.edu`, o nome de pacote que você utilizaria é `edu.suauniversidade`. Depois que o nome de domínio é invertido, você pode escolher qualquer outro nome para seu pacote. Se você faz parte de uma empresa com muitas divisões ou uma universidade com muitas faculdades, você pode querer utilizar o nome de sua divisão ou faculdade como o próximo nome no pacote. Escolhemos utilizar `jhttp4` como o próximo nome em nosso nome de pacote para indicar que essa classe é do livro *Java Como Programar: Quarta Edição*. O último nome em nosso nome de pacote especifica que esse pacote é para o Capítulo 8 (`ch08`). [Nota: utilizamos nossos próprios pacotes várias vezes em todo o livro. Você pode determinar o capítulo em que uma de nossas classes reutilizáveis é definida olhando-se a última parte do nome do pacote na instrução `import`. Isso aparece antes do nome da classe que está sendo importada ou antes do `*`, se não for especificada nenhuma classe em particular.]

O Passo 3 é compilar a classe para que seja armazenado no pacote apropriado. Quando um arquivo Java que contém uma instrução `package` é compilado, o arquivo `.class` resultante é colocado no diretório especificado pela instrução `package`. A instrução `package` precedente indica que a classe `Time1` deve ser colocada no diretório `ch08`. Os outros nomes – `com`, `deitel` e `jhttp4` – também são diretórios. Os nomes de diretório na instrução `package` especificam a posição exata das classes no pacote. Se esses diretórios não existirem antes de a classe ser compilada, o compilador os cria.

Ao compilar uma classe em um pacote, há uma opção extra (`-d`) que deve ser passada para o compilador `javac`. Esta opção especifica onde criar (ou localizar) os diretórios na instrução `package`. Por exemplo, utilizamos o comando de compilação

```
javac -d . Time1.java
```

para especificar que o primeiro diretório especificado em nosso nome de pacote deve ser colocado no diretório corrente. O `.` após o `-d` no comando precedente representa o diretório corrente nos sistemas operacionais Windows, UNIX e Linux. Depois de executado o comando de compilação, o diretório corrente contém um diretório chamado `com`; `com` contém um diretório chamado `deitel`; `deitel` contém um diretório chamado `jhttp4`, e `jhttp4` contém um diretório chamado `ch08`. No diretório `ch08`, você pode localizar o arquivo `Time1.class`.

Os nomes de diretório em `package` tornam-se parte do nome de classe quando a classe é compilada. O nome de classe nesse exemplo é, na verdade, `com.deitel.jhttp3.ch08.Time1` depois que a classe é compilada.

Você pode utilizar esse nome *completamente qualificado* em seus programas ou pode **importar** a classe e utilizar seu nome abreviado (`Time1`) no programa. Se outro pacote também contiver uma classe `Time1`, os nomes de classe completamente qualificados podem ser utilizados para distinguir entre as classes no programa e evitar um conflito de nomes (também chamado de colisão de nomes).

Uma vez que a classe é compilada e armazenada em seu pacote, ela pode ser importada para programas (*Passo 4*). No aplicativo `TimeTest3` da Fig. 8.5, a linha 8 especifica que a classe `Time1` deve ser **importada** para utilização na classe `TimeTest3`.

Durante a compilação da classe `TimeTest3`, `javac` precisa localizar o arquivo `.class` que contém `Time1`, de modo que `javac` possa assegurar que a classe `TimeTest3` usa a classe `Time1` corretamente. O compilador segue uma ordem de pesquisa específica para localizar as classes de que ele precisa. Ele começa procurando as classes-padrão de Java que estão incluídas no J2SDK. A seguir, ele procura as *classes de extensão*. Java 2 oferece um *mecanismo de extensão* que permite que novos pacotes sejam adicionados a Java para fins de desenvolvimento e execução. [Nota: o mecanismo de extensão está além do escopo deste livro. Para obter mais informações, visite java.sun.com/j2se/1.3/docs/guide/extensions.] Se a classe não for encontrada entre as classes-padrão de Java nem nas classes de extensão, o compilador pesquisa no *caminho para as classes*. Por *default*, o caminho para as classes consiste apenas no diretório corrente. Entretanto, o caminho para as classes pode ser modificado:

```

1 // Fig. 8.5: TimeTest3.java
2 // Classe TimeTest3 que usa a classe importada Time1
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 // Deitel packages
8 import com.deitel.jhtp4.ch08.Time1; // importa a classe Time1
9
10 public class TimeTest3 {
11
12     // cria um objeto da classe Time1 e o manipula
13     public static void main( String args[] )
14     {
15         Time1 time = new Time1(); // cria o objeto Time1
16
17         time.setTime( 13, 27, 06 ); // set new time
18         String output =
19             "Universal time is: " + time.toUniversalString() +
20             "\nStandard time is: " + time.toString();
21
22         JOptionPane.showMessageDialog( null, output,
23             "Packaging Class Time1 for Reuse",
24             JOptionPane.INFORMATION_MESSAGE );
25
26         System.exit( 0 );
27     }
28
29 } // end class TimeTest3

```

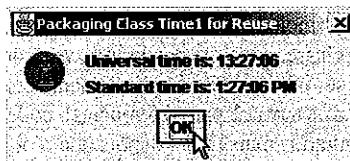


Fig 8.5 Usando a classe definida pelo programador `Time1` num pacote.

1. fornecendo a opção `-classpath` para o compilador `javac`, ou
2. configurando a variável de ambiente `CLASSPATH` (uma variável especial que você define e o sistema operacional mantém para que os aplicativos possam procurar as classes nas posições especificadas).

Em cada caso, o caminho para as classes consiste em uma lista de diretórios e/ou arquivos *compactados* separados por ponto-e-vírgulas (;). Arquivos compactados são arquivos individuais que contêm diretórios de outros arquivos, geralmente em formato comprimido. Por exemplo, as classes-padrão de Java estão contidas no arquivo compactado `rt.jar` que é instalado com o J2SDK. Os arquivos compactados normalmente terminam com as extensões de nome `.jar` ou `.zip`. Os diretórios e arquivos compactados especificados no caminho para as classes contêm as classes que você deseja tornar disponíveis para o compilador Java. Para obter mais informações sobre o caminho para as classes, visite o endereço Java.sun.com/J2se/1.3/docs/tooldocs/win32/classpath.html para Windows ou o endereço Java.sun.com/J2se/1.3/docs/tooldocs/solaris/classpath.html para Solaris/Linux. [Nota: discutimos arquivos *archive* em mais detalhes na Seção 8.8.]



Erro comum de programação 8.5

Especificar um caminho explícito para as classes elimina o diretório corrente do caminho para as classes. Isto impede que as classes do diretório corrente sejam carregadas apropriadamente. Se há necessidade de carregar classes do diretório corrente, inclua o diretório corrente (.) no caminho explícito para as classes.



Observação de engenharia de software 8.13

Em geral, é uma prática melhor usar a opção `-classpath` do compilador, em vez de usar a variável de ambiente `CLASSPATH`, para especificar o caminho para as classes de um programa. Isso permite que cada aplicativo tenha seu próprio caminho para as classes.



Dica de teste e depuração 8.2

Especificar o caminho para as classes com a variável de ambiente `CLASSPATH` pode provocar erros sutis e difíceis de localizar em programas que usam versões diferentes dos mesmos pacotes.

Para o exemplo das Figuras 8.4 e 8.5, não especificamos um caminho específico para as classes. Assim, para encontrar as classes no pacote `com.deitel.jhttp4.ch08` deste exemplo, o compilador procura no diretório corrente pelo primeiro nome do pacote – `com`. A seguir, o compilador navega pela estrutura de diretórios. O diretório `com` contém o subdiretório `deitel`. O diretório `deitel` contém o subdiretório `jhttp4`. Finalmente, o diretório `jhttp4` contém o subdiretório `ch08`. No diretório `ch08` está o arquivo `Time1.class`, que é carregado pelo compilador para assegurar que a classe é usada apropriadamente em nosso programa.

Localizar as classes para executar o programa é semelhante a localizar as classes para compilar o programa. Assim como o compilador, o interpretador `java` procura primeiro nas classes-padrão e classes de extensão, em seguida procura no caminho para as classes (que por *default* é o diretório corrente). O caminho para as classes para o interpretador pode ser especificado explicitamente usando-se qualquer uma das técnicas discutidas para o compilador. Assim como para o compilador, é melhor especificar o caminho para as classes de um programa individual através de opções de linha de comando para o interpretador. Você pode especificar o caminho para as classes para o interpretador `java` através das opções de linha de comando `-classpath` ou `-cp`, seguidas por uma lista de diretórios e/ou arquivos compactados separados por ponto-e-vírgulas (;).

8.6 Inicializando objetos de classe: construtores

Quando um objeto é criado, seus membros podem ser inicializados por um método *construtor*. O construtor é um método com o mesmo nome que a classe (incluindo distinção entre maiúsculas e minúsculas). O programador de uma classe pode definir o construtor da classe, que é invocado toda vez que o programa instancia um objeto dessa classe. As variáveis de instância podem ser inicializadas implicitamente com seus valores *default* (0 para tipos numéricos primitivos, `false` para `booleans` e `null` para referências), podem ser inicializadas em um construtor da classe ou seus valores podem ser configurados mais tarde, depois que o objeto for criado. Os construtores não podem especificar tipos devolvidos nem valores devolvidos. A classe pode conter construtores sobrecarregados para oferecer diversas maneiras de inicializar os objetos dessa classe.

Quando um programa instancia um objeto de uma classe, o programa pode fornecer *inicializadores* entre parênteses à direita do nome de classe. Esses inicializadores são passados como argumentos para o construtor da classe. Essa técnica é demonstrada no exemplo da Seção 8.7. Também vimos essa técnica várias vezes anteriormente,

quando criamos novos objetos de classes como `DecimalFormat`, `JLabel`, `JTextField`, `JTextArea` e `JButton`. Para cada uma dessas classes, vimos instruções da forma

```
ref = new NomeDaClasse( argumentos );
```

em que `ref` é uma referência do tipo de dados apropriado, `new` indica que um novo objeto está sendo criado, `NomeDaClasse` indica o tipo do novo objeto e `argumentos` especificam os valores utilizados pelo construtor da classe para inicializar o objeto.

Se nenhum construtor é definido para uma classe, o compilador cria um *construtor default* que não recebe nenhum argumento (também chamado de *construtor sem argumentos*). O construtor *default* para uma classe chama o construtor *default* para a sua superclasse (a classe que ela estende), então prossegue para inicializar as variáveis de instância da maneira como discutimos antes. Se a classe que essa classe estende não tem um construtor *default*, o compilador emite uma mensagem de erro. Além disso, o programador pode fornecer um construtor sem argumentos, como mostramos na classe `Time1` e como você verá no próximo exemplo. Se algum construtor é definido pelo programador para uma classe, Java não criará um construtor *default* para a classe.



Boa prática de programação 8.4

Quando apropriado (quase sempre), forneça um construtor para assegurar que cada objeto seja inicializado com valores significativos.



Erro comum de programação 8.6

Se forem fornecidos construtores para uma classe, mas nenhum dos construtores public for um construtor sem argumentos, e se tentar chamar um construtor sem argumentos para inicializar um objeto da classe, ocorre um erro de sintaxe. Um construtor pode ser chamado sem argumentos somente se não houver nenhum construtor para a classe (o construtor default é chamado) ou se houver um construtor sem argumentos.

8.7 Utilizando construtores sobrecarregados

Os métodos de uma classe podem ser *sobre carregados* (isto é, vários métodos em uma classe podem ter exatamente o mesmo nome, como definido no Capítulo 6). Para sobre carregar um método de uma classe, simplesmente forneça uma definição separada de método com o mesmo nome para cada versão do método. Lembre-se de que os métodos sobre carregados *devem* ter listas de parâmetro diferentes.



Erro comum de programação 8.7

Tentar sobre carregar um método de uma classe com outro método que tem exatamente a mesma assinatura (nome e parâmetros) é um erro de sintaxe.

O construtor `Time1` na Fig. 8.1 inicializou `hour`, `minute` e `second` com 0 (isto é, meia-noite no horário universal) com uma chamada ao método `setTime` da classe. A Fig. 8.6 sobre carrega o método construtor para oferecer uma variedade conveniente de maneiras de inicializar objetos da nova classe `Time2`. Os construtores garantem que cada objeto inicia sua existência em um estado consistente. Nesse programa, cada construtor chama o método `setTime` com os valores passados para o construtor, para assegurar que o valor fornecido para `hour` esteja no intervalo 0 a 23 e que os valores para `minute` e `second` estejam no intervalo 0 a 59. Se um valor estiver fora do intervalo, ele será configurado como zero por `setTime` (mais uma vez, assegurando que cada variável de instância permaneça em um estado consistente). O construtor apropriado é invocado comparando a quantidade, os tipos e a ordem dos argumentos especificados na chamada do construtor com a quantidade, os tipos e a ordem dos parâmetros especificados na definição de cada construtor. O construtor correspondente é chamado automaticamente. A Fig. 8.7 usa a classe `Time2` para demonstrar seus construtores.

```
1 // Fig. 8.6: Time2.java
2 // Definição da classe Time2 com construtores sobre carregados
3 package com.deitel.jhtp4.ch08;
4
5 // Pacotes do núcleo de Java
6 import java.text.DecimalFormat;
```

Fig. 8.6 Classe Time2 com construtores sobre carregados (parte 1 de 3).

```

7
8 public class Time2 extends Object {
9     private int hour;      // 0 - 23
10    private int minute;    // 0 - 59
11    private int second;    // 0 - 59
12
13    // Construtor de Time2 inicializa cada variável de instância
14    // com zero. Assegura que o objeto Time inicia em um
15    // estado consistente.
16    public Time2()
17    {
18        setTime( 0, 0, 0 );
19    }
20
21    // Construtor de Time2: a hora é informada,
22    // minuto e segundo são colocados em zero por default
23    public Time2( int h )
24    {
25        setTime( h, 0, 0 );
26    }
27
28    // Construtor de Time2: hora e minuto informados,
29    // segundo colocado em zero por default
30    public Time2( int h, int m )
31    {
32        setTime( h, m, 0 );
33    }
34
35    // Construtor de Time2: hora, minuto e segundo informados
36    public Time2( int h, int m, int s )
37    {
38        setTime( h, m, s );
39    }
40
41    // Construtor de Time2: outro objeto Time2 fornecido
42    public Time2( Time2 time )
43    {
44        setTime( time.hour, time.minute, time.second );
45    }
46
47    // Configura um novo valor de hora, usando formato de hora universal.
48    // Faz teste de validade dos dados. Configura valores inválidos como zero.
49    public void setTime( int h, int m, int s )
50    {
51        hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
52        minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
53        second = ( ( s >= 0 && s < 60 ) ? s : 0 );
54    }
55
56    // converte para String no formato de hora universal
57    public String toUniversalString()
58    {
59        DecimalFormat twoDigits = new DecimalFormat( "00" );
60
61        return twoDigits.format( hour ) + ":" +
62            twoDigits.format( minute ) + ":" +
63            twoDigits.format( second );
64    }
65

```

Fig. 8.6 Classe Time2 com construtores sobrecarregados (parte 2 de 3).

```

66     // converte para String no formato de hora padrão
67     public String toString()
68     {
69         DecimalFormat twoDigits = new DecimalFormat( "00" );
70
71         return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
72             ":" + twoDigits.format( minute ) +
73             ":" + twoDigits.format( second ) +
74             ( hour < 12 ? " AM" : " PM" );
75     }
76
77 } // fim da classe Time2

```

Fig. 8.6 Classe Time2 com construtores sobrecarregados (parte 3 de 3).

A maior parte do código na classe **Time2** é idêntica àquela da classe **Time1**, então aqui nos concentramos apenas nos novos recursos (isto é, os construtores). As linhas 16 a 19 definem o construtor sem argumentos (padrão). As linhas 23 a 26 definem um construtor **Time2** que recebe um único argumento **int** que representa **hour**. As linhas 30 a 33 definem um construtor **Time2** que recebe dois argumentos **int** que representam **hour** e **minute**. As linhas 36 a 39 definem um construtor **Time2** que recebe três argumentos **int** que representam **hour**, **minute** e **second**. As linhas 42 a 45 definem um construtor **Time2** que recebe uma referência **Time2** para outro objeto **Time2**. Nesse caso, os valores do argumento **Time2** são utilizados para inicializar **hour**, **minute** e **second**. Repare que nenhum dos construtores especifica um tipo de dados retornado (lembre-se, isso não é permitido aos construtores). Além disso, repare que todos os construtores recebem quantidades diferentes de argumentos e/ou tipos de argumentos diferentes. Embora apenas dois dos construtores recebam valores para **hour**, **minute** e **second**, todos os construtores chamam **setTime** com valores para **hour**, **minute** e **second** e substituem os valores ausentes por zero para satisfazer o requisito de três argumentos do **setTime**.

Repare principalmente no construtor nas linhas 42 a 45, que utiliza os valores **hour**, **minute** e **second** de seu argumento **time** para inicializar o novo objeto **Time2**. Mesmo sabendo que **hour**, **minute** e **second** são declarados como variáveis **private** da classe **Time2**, somos capazes de acessar diretamente esses valores a partir dos objetos a que **time** faz referência utilizando as expressões **time.hour**, **time.minute** e **time.second**. Isso se deve a um relacionamento especial entre objetos da mesma classe. Quando um objeto de uma classe tem uma referência para outro objeto da mesma classe, o primeiro objeto pode acessar *todos* os métodos e dados do segundo objeto.

A classe **TimeTest4** (Fig. 8.7) cria seis objetos **Time2** (linhas 17 a 22) para demonstrar como invocar os diferentes construtores da classe. A linha 17 mostra que o construtor sem argumentos é invocado colocando-se um conjunto vazio de parênteses depois do nome da classe ao alocar um objeto **Time2** com **new**. As linhas 18 a 22 do programa demonstram a passagem de argumentos para os construtores **Time2**. Lembre-se de que o construtor apropriado é invocado comparando-se a quantidade, os tipos e a ordem dos argumentos especificados na chamada do construtor com a quantidade, os tipos e a ordem dos parâmetros especificados em cada definição de método. Assim, a linha 18 invoca o construtor na linha 23 da Fig. 8.6. A linha 19 invoca o construtor na linha 30 da Fig. 8.6. As linhas 20 e 21 invocam o construtor na linha 36 da Fig. 8.6. A linha 22 invoca o construtor na linha 42 da Fig. 8.6.

Observe que cada construtor **Time2** na Fig. 8.6 poderia ter sido escrito incluindo-se uma cópia das instruções apropriadas do método **setTime**. Isso poderia ser ligeiramente mais eficiente porque a chamada extra a **setTime** é eliminada. Entretanto, considere a alteração da representação da hora de três valores **int** (exigindo 12 bytes de memória) para um único valor **int** representando o número total de segundos decorridos no dia (exigindo 4 bytes de memória). Codificar os construtores **Time2** e o método **setTime** igualmente torna a alteração nessa definição de classe mais difícil. Se a implementação de método **setTime** mudar, a implementação dos construtores **Time2** precisaria ser alterada de acordo. Fazer os construtores de **Time2** chamar **setTime** diretamente exige que qualquer alteração à implementação de **setTime** seja feita somente uma vez. Isso reduz a probabilidade de um erro de programação ao alterar a implementação.

Observação de engenharia de software 8.14



Se um método de uma classe já fornece toda ou parte da funcionalidade exigida por um construtor (ou outro método) da classe, chame esse método a partir do construtor (ou do outro método). Isso simplifica a manutenção do código.

digo e reduz a probabilidade de erro se a implementação do código for modificada. Também é um exemplo efetivo de reutilização.

```

1 // Fig. 8.7: TimeTest4.java
2 // Usando construtores sobrecarregados
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 // Pacotes Deitel
8 import com.deitel.jhttp4.ch08.Time2;
9
10 public class TimeTest4 {
11
12     // testa construtores da classe Time2
13     public static void main( String args[] )
14     {
15         Time2 t1, t2, t3, t4, t5, t6;
16
17         t1 = new Time2();                      // 00:00:00
18         t2 = new Time2( 2 );                  // 02:00:00
19         t3 = new Time2( 21, 34 );            // 21:34:00
20         t4 = new Time2( 12, 25, 42 );        // 12:25:42
21         t5 = new Time2( 27, 74, 99 );        // 00:00:00
22         t6 = new Time2( t4 );              // 12:25:42
23
24         String output = "Constructed with: " +
25             "\nt1: all arguments defaulted" +
26             "\n      " + t1.toUniversalString() +
27             "\n      " + t1.toString();
28
29         output += "\nt2: hour specified; minute and " +
30             "second defaulted" +
31             "\n      " + t2.toUniversalString() +
32             "\n      " + t2.toString();
33
34         output += "\nt3: hour and minute specified; " +
35             "second defaulted" +
36             "\n      " + t3.toUniversalString() +
37             "\n      " + t3.toString();
38
39         output += "\nt4: hour, minute, and second specified" +
40             "\n      " + t4.toUniversalString() +
41             "\n      " + t4.toString();
42
43         output += "\nt5: all invalid values specified" +
44             "\n      " + t5.toUniversalString() +
45             "\n      " + t5.toString();
46
47         output += "\nt6: Time2 object t4 specified" +
48             "\n      " + t6.toUniversalString() +
49             "\n      " + t6.toString();
50
51         JOptionPane.showMessageDialog( null, output,
52             "Demonstrating Overloaded Constructors",
53             JOptionPane.INFORMATION_MESSAGE );
54
55         System.exit( 0 );
56     }

```

Fig. 8.7 Usando construtores sobrecarregados para inicializar objetos da classe Time2 (parte 1 de 2).

```

57
58 } // fim da classe TimeTest4

```

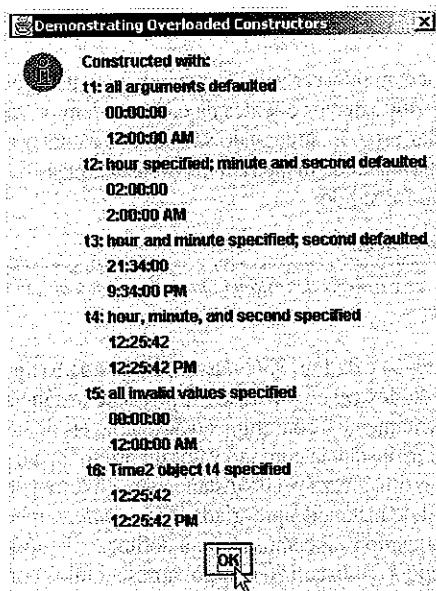


Fig. 8.7 Usando construtores sobrecarregados para inicializar objetos da classe Time2 (parte 2 de 2).

8.8 Utilizando os métodos *set* e *get*

As variáveis de instância privadas podem ser manipuladas somente por métodos da classe. Uma manipulação típica poderia ser o ajuste do saldo bancário de um cliente (por exemplo, uma variável de instância **private** de uma classe **BankAccount**) por um método **computeInterest**.

As classes freqüentemente fornecem métodos **public** para permitir aos clientes da classe configurar (*set*, isto é, atribuir valores a) ou obter (*get*, isto é, obter os valores de) variáveis de instância **private**. Esses métodos não precisam se chamar *set* e *get*, mas isso ocorre freqüentemente.

Como exemplo de atribuição de nome, geralmente um método que configura uma variável de instância **interestRate** seria chamado de **setInterestRate** e, em geral, um método que obtém **interestRate** seria chamado de **getInterestRate**. Os métodos *get* são comumente chamados de *métodos de acesso* ou *métodos de consulta*. Os métodos *set* também são comumente chamados de *métodos modificadores* (uma vez que eles em geral alteram um valor).

Poderia parecer que fornecer as capacidades de *set* e *get* é essencialmente o mesmo que tornar **public** as variáveis de instância. Essa é outra sutileza de Java que torna a linguagem tão desejável para engenharia de *software*. Se uma variável de instância for **public**, ela pode ser lida ou escrita à vontade por qualquer método no programa. Se uma variável de instância for **private**, o método *get public* certamente parece permitir que outros métodos leiam os dados à vontade, mas o método *get* controla a formatação e exibição dos dados. O método *set public* pode – e muito provável irá – escrutinar cuidadosamente as tentativas de modificar o valor da variável de instância. Isso assegura que o novo valor seja apropriado para esse item de dados. Por exemplo, a tentativa de configurar (*set*) o dia do mês com uma data como 37 seria rejeitada, a tentativa de configurar (*set*) o peso de uma pessoa com um valor negativo seria rejeitada, e assim por diante. Então, embora os métodos *set* e *get* possam fornecer acesso a dados **private**, o acesso é restrinido pela implementação dos métodos feita pelo programador.

Os benefícios da integridade dos dados não surgem automaticamente apenas porque as variáveis de instância são tornadas privadas (**private**) – o programador deve providenciar a verificação de validade. Java fornece uma estrutura em que os programadores podem projetar programas melhores de uma maneira conveniente.



Observação de engenharia de software 8.15

Os métodos que configuram os valores de dados **private** devem verificar se os novos valores pretendidos são adequados; se eles não forem, os métodos *set* devem colocar variáveis de instância **private** em um estado consistente apropriado.

Os métodos *set* de uma classe podem retornar valores que indicam que foram feitas tentativas de atribuir dados inválidos a objetos da classe. Isso permite que os clientes da classe testem os valores retornados pelos métodos *set* para determinar se os objetos que eles estão manipulando são válidos e adotar a ação apropriada se os objetos não forem válidos. No Capítulo 14, ilustramos uma maneira mais robusta através da qual os clientes de uma classe podem ser notificados se um objeto não for válido.



Boa prática de programação 8.5

Cada método que modifica as variáveis de instância privadas de um objeto deve assegurar que os dados permanecem em um estado consistente.

O applet da Fig. 8.8 (classe **Time3**) e da Fig. 8.9 (classe **TimeTest5**) aprimoraram nossa classe **Time** (agora chamada de **Time3**) incluindo os métodos *get* e *set* para as variáveis de instância **private hour**, **minute** e **second**. Os métodos *set* controlam estritamente a configuração das variáveis de instância para valores válidos. Tentativas de configurar qualquer variável de instância com um valor incorreto fazem com que a variável de instância seja configurada com zero (deixando, assim, a variável de instância em um estado consistente). Cada método *get* simplesmente retorna o valor da variável de instância apropriada. Esse applet também apresenta técnicas aprimoradas de tratamento de eventos de GUI enquanto avançamos rumo à definição de nosso primeiro aplicativo completo de janelas e com todos os recursos. Depois de discutir o código, apresentamos como configurar um *applet* para usar classes em pacotes definidos pelo programador.

```

1 // Fig. 8.8: Time3.java
2 // Definição da classe Time3 com métodos set e get
3 package com.deitel.jhtp4.ch08;
4
5 // Pacotes do núcleo de Java
6 import java.text.DecimalFormat;
7
8 public class Time3 extends Object {
9     private int hour;      // 0 - 23
10    private int minute;    // 0 - 59
11    private int second;    // 0 - 59
12
13    // Construtor Time3 inicializa cada variável de instância
14    // com zero. Assegura que o objeto Time inicie em um
15    // estado consistente.
16    public Time3()
17    {
18        setTime( 0, 0, 0 );
19    }
20
21    // Construtor Time3: hora informada, minuto e
22    // segundo colocados em 0 por default
23    public Time3( int h )
24    {
25        setTime( h, 0, 0 );
26    }
27
28    // Construtor Time3: hora e minuto informados,
29    // segundo colocado em 0 por default
30    public Time3( int h, int m )
31    {

```

Fig. 8.8 Classe **Time3** com métodos *set* e *get* (parte 1 de 3).

```

32     setTime( h, m, 0 );
33 }
34
35 // Construtor Time3: hora, minuto e segundo informados
36 public Time3( int h, int m, int s )
37 {
38     setTime( h, m, s );
39 }
40
41 // Construtor Time3: outro objeto Time3 fornecido
42 public Time3( Time3 time )
43 {
44     setTime( time.getHour(), time.getMinute(),
45             time.getSecond() );
46 }
47
48 // Métodos Set
49 // Configura um novo valor de hora usando o formato universal. Faz
50 // verificação de validade nos dados. Configura valores inválidos com zero.
51 public void setTime( int h, int m, int s )
52 {
53     setHour( h ); // configura a hora
54     setMinute( m ); // configura o minuto
55     setSecond( s ); // configura o segundo
56 }
57
58 // valida e configura a hora
59 public void setHour( int h )
60 {
61     hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
62 }
63
64 // valida e configura o minuto
65 public void setMinute( int m )
66 {
67     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
68 }
69
70 // valida e configura o segundo
71 public void setSecond( int s )
72 {
73     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
74 }
75
76 // Métodos Get
77 // obtém valor da hora
78 public int getHour()
79 {
80     return hour;
81 }
82
83 // obtém valor do minuto
84 public int getMinute()
85 {
86     return minute;
87 }
88
89 // obtém valor do segundo
90 public int getSecond()
91 {

```

Fig. 8.8 Classe Time3 com métodos set e get (parte 2 de 3).

```

92     return second;
93 }
94
95 // converte para String no formato de hora universal
96 public String toUniversalString()
97 {
98     DecimalFormat twoDigits = new DecimalFormat( "00" );
99
100    return twoDigits.format( getHour() ) + ":" +
101        twoDigits.format( getMinute() ) + ":" +
102        twoDigits.format( getSecond() );
103 }
104
105 // converte para String no formato de hora padrão
106 public String toString()
107 {
108     DecimalFormat twoDigits = new DecimalFormat( "00" );
109
110    return ( ( getHour() == 12 || getHour() == 0 ) ?
111        12 : getHour() % 12 ) + ":" +
112        twoDigits.format( getMinute() ) + ":" +
113        twoDigits.format( getSecond() ) +
114        ( getHour() < 12 ? " AM" : " PM" );
115 }
116
117 } // fim da classe Time3

```

Fig. 8.8 Classe Time3 com métodos *set* e *get* (parte 3 de 3).

Os novos métodos *set* da classe são definidos na Fig. 8.8, nas linhas 59 a 62, 65 a 68 e 71 a 74, respectivamente. Repare que cada método executa a mesma instrução condicional que estava previamente no método **setTime** para configurar **hour**, **minute** ou **second**. A adição desses métodos fez com que redefiníssemos o corpo do método **setTime** para seguir a *Observação de engenharia de software 8.14* – se um método de uma classe já fornece toda ou parte da funcionalidade exigida por outro método da classe, chame aquele método a partir do outro método. Repare que **setTime** (linhas 51 a 56) agora chama os métodos **setHour**, **setMinute** e **setSecond** – cada um dos quais executa parte da tarefa de **setTime**.

Os novos métodos *get* da classe são definidos nas linhas 78 a 81, 84 a 87 e 90 a 93, respectivamente. Repare que cada método simplesmente devolve o valor **hour**, **minute** ou **second** (devolve-se uma cópia de cada valor, porque essas são todas variáveis de tipos primitivos de dados). A adição desses métodos fez com que redefiníssemos os corpos dos métodos **toUniversalString** (linhas 96 a 103) e **toString** (linhas 106 a 115) para seguir a *Observação de engenharia de software 8.14*. Em ambos os casos, cada utilização das variáveis de instância **hour**, **minute** e **second** é substituída por uma chamada para **getHour**, **getMinute** e **getSecond**.

Devido às alterações na classe Time3 recém-descritas, minimizamos as alterações que terão de ocorrer na definição de classe se a representação de dados for alterada de **hour**, **minute** e **second** para outra representação (como o total de segundos decorridos no dia). Somente o corpo dos novos métodos *set* e *get* terá de ser alterado. Isso permite alterar a implementação da classe sem afetar os clientes da classe (contanto que todos os métodos **public** da classe ainda sejam chamados da mesma maneira).

O applet **TimeTest5** (Fig. 8.9) fornece uma interface gráfica com o usuário que permite exercitar os métodos da classe Time3. O usuário pode configurar o valor de hora, minuto ou segundo digitando um valor no JTextField apropriado e pressionando a tecla *Enter*. Além disso, o usuário pode clicar o botão “**Add 1 to second**” para incrementar o tempo em um segundo. Os eventos de JTextField e JButton nesse applet são todos processados no método **actionPerformed** (linhas 66 a 94). Repare que as linhas 34, 41, 48 e 59 chamam **addActionListener** para indicar que o applet deve começar a escutar eventos dos JTextFields **hourField**, **minuteField**, **secondField** e do JButton **tickButton**, respectivamente. Repare também que todas as quatro chamadas utilizam **this** como argumento, para indicar que o objeto de nossa classe de applet **TimeTest5** tem seu método **actionPerformed** invocado para cada interação do usuário com esses quatro componentes GUI. Isso impõe uma pergunta interessante – como determinamos o componente GUI com que o usuário interagiu?

Em `actionPerformed`, observe o uso de `actionEvent.getSource()` para determinar qual componente GUI gerou o evento. Por exemplo, a linha 69 determina se o usuário clicou em `tickButton`. Se foi, o corpo da estrutura `if` é executado. Caso contrário, o programa testa a condição na estrutura `if` na linha 73, e assim por diante. Cada evento tem uma *origem* – o componente GUI com o qual o usuário interagiu para sinalizar que o programa devia fazer uma tarefa. O parâmetro `ActionEvent` contém uma referência para a origem do evento. A condição na linha 69 simplesmente pergunta: “A *origem* do evento é o `tickButton`?” Essa condição compara as referências em cada lado do operador `==` para determinar se elas se referem ao mesmo objeto. Nesse caso, se ambas se referirem ao `JButton`, então o programa sabe que o usuário pressionou o botão. Lembre-se de que a origem do evento chama `actionPerformed` em resposta à interação do usuário.

Após cada operação, o tempo resultante é exibido como um *string* na barra de estado do *applet*. As janelas de saída na Fig. 8.9 ilustram o *applet* antes e depois das seguintes operações: configurar a hora como 23, configurar o minuto como 59, configurar o segundo como 58 e incrementar o segundo duas vezes com botão “**Add 1 to second**”.

```

1 // Fig. 8.9: TimeTest5.java
2 // Demonstrando os métodos set e get da classe Time3.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 // Pacotes Deitel
12 import com.deitel.jhtp4.ch08.Time3;
13
14 public class TimeTest5 extends JApplet
15     implements ActionListener {
16
17     private Time3 time;
18     private JLabel hourLabel, minuteLabel, secondLabel;
19     private JTextField hourField, minuteField,
20         secondField, displayField;
21     private JButton tickButton;
22
23     // Cria objeto Time3 e configura a GUI
24     public void init()
25     {
26         time = new Time3();
27
28         Container container = getContentPane();
29         container.setLayout( new FlowLayout() );
30
31         // configura hourLabel e hourField
32         hourLabel = new JLabel( "Set Hour" );
33         hourField = new JTextField( 10 );
34         hourField.addActionListener( this );
35         container.add( hourLabel );
36         container.add( hourField );
37
38         // configura minuteLabel e minuteField
39         minuteLabel = new JLabel( "Set minute" );
40         minuteField = new JTextField( 10 );
41         minuteField.addActionListener( this );
42         container.add( minuteLabel );
43         container.add( minuteField );
44

```

Fig. 8.9 Usando os métodos `set` e `get` da classe `Time3` (parte 1 de 4).

```

45     // configura secondLabel e secondField
46     secondLabel = new JLabel( "Set Second" );
47     secondField = new JTextField( 10 );
48     secondField.addActionListener( this );
49     container.add( secondLabel );
50     container.add( secondField );
51
52     // configura displayField
53     displayField = new JTextField( 30 );
54     displayField.setEditable( false );
55     container.add( displayField );
56
57     // configura tickButton
58     tickButton = new JButton( "Add 1 to Second" );
59     tickButton.addActionListener( this );
60     container.add( tickButton );
61
62     updateDisplay();    // atualiza o texto em displayField
63 }
64
65 // trata eventos do botão e do campo de texto
66 public void actionPerformed( ActionEvent actionEvent )
67 {
68     // processa o evento tickButton
69     if ( actionEvent.getSource() == tickButton )
70         tick();
71
72     // processa o evento hourField
73     else if ( actionEvent.getSource() == hourField ) {
74         time.setHour(
75             Integer.parseInt( actionEvent.getActionCommand() ) );
76         hourField.setText( "" );
77     }
78
79     // processa o evento minuteField
80     else if ( actionEvent.getSource() == minuteField ) {
81         time.setMinute(
82             Integer.parseInt( actionEvent.getActionCommand() ) );
83         minuteField.setText( "" );
84     }
85
86     // processa o evento secondField
87     else if ( actionEvent.getSource() == secondField ) {
88         time.setSecond(
89             Integer.parseInt( actionEvent.getActionCommand() ) );
90         secondField.setText( "" );
91     }
92
93     updateDisplay();    // atualiza displayField e a barra de estado
94 }
95
96 // atualiza displayField e a barra de estado do contêiner de applets
97 public void updateDisplay()
98 {
99     displayField.setText( "Hour: " + time.getHour() +
100         "; Minute: " + time.getMinute() +
101         "; Second: " + time.getSecond() );
102
103     showStatus( "Standard time is: " + time.toString() +
104         "; Universal time is: " + time.toUniversalString() );

```

Fig. 8.9 Usando os métodos *set* e *get* da classe Time3 (parte 2 de 4).

```

105    }
106
107    // soma um ao segundo e atualiza hora e minuto se necessário
108    public void tick()
109    {
110        time.setSecond( ( time.getSecond() + 1 ) % 60 );
111
112        if ( time.getSecond() == 0 )
113            time.setMinute( ( time.getMinute() + 1 ) % 60 );
114
115        if ( time.getMinute() == 0 )
116            time.setHour( ( time.getHour() + 1 ) % 24 );
117    }
118
119
120 } // fim da classe TimeTest5

```

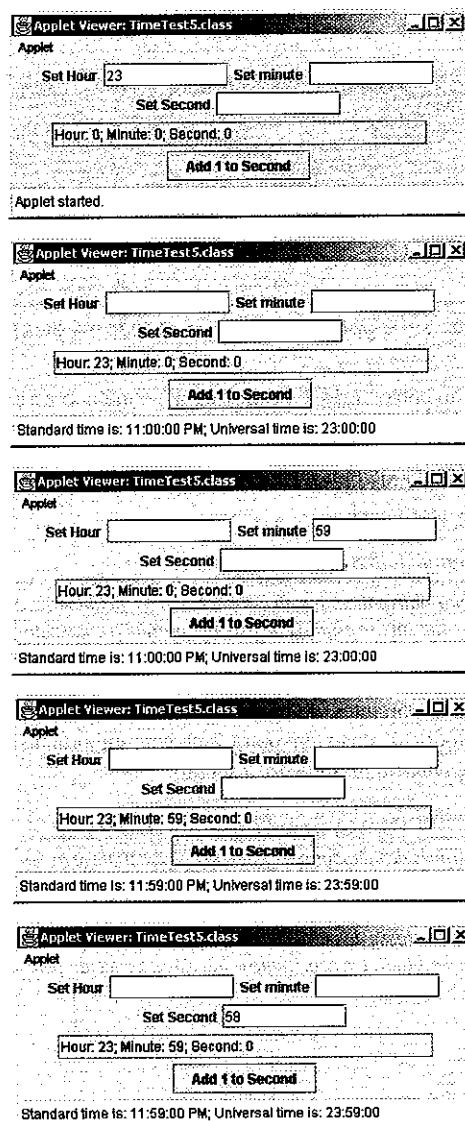


Fig. 8.9 Usando os métodos *set* e *get* da classe Time3 (parte 3 de 4).

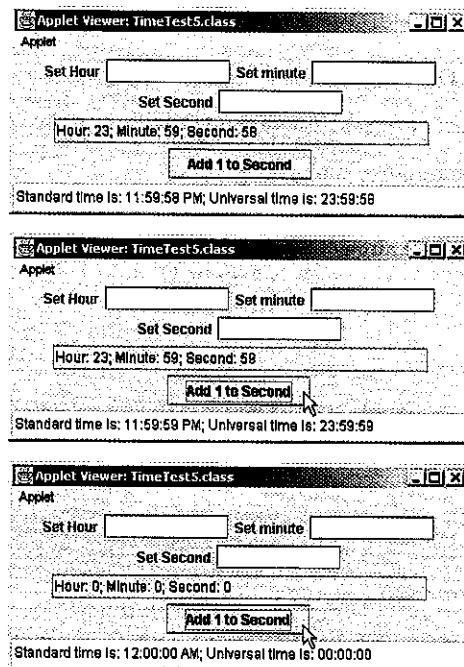


Fig. 8.9 Usando os métodos *set* e *get* da classe *Time3* (parte 4 de 4).

Observe que, quando se clica no botão “**Add 1 to second**”, o método `actionPerformed` chama o método `tick` do *applet* (definido nas linhas 108 a 118). O método `tick` utiliza todos os novos métodos *set* e *get* para incrementar o segundo adequadamente. Embora funcione, isso incorre em um problema de desempenho pelo fato de fazer múltiplas chamadas de método. Na Seção 8.12, discutiremos a noção de acesso de pacote como uma maneira de eliminar esse problema de desempenho.



Erro comum de programação 8.8

*Um construtor pode chamar outros métodos da classe, como os métodos *set* ou *get*. Entretanto, as variáveis de instância ainda podem não estar em um estado consistente, porque o construtor está inicializando o objeto. Usar variáveis de instância antes de elas serem adequadamente inicializadas é um erro de lógica.*

Os métodos *set* certamente são importantes, do ponto de vista da engenharia de *software*, porque podem realizar verificação de validade. Os métodos *set* e *get* têm outra vantagem importante para a engenharia de *software* como discutido na *Observação de engenharia de software* a seguir.



Observação de engenharia de software 8.16

*Acessar dados privados através de métodos *set* e *get* não apenas protege as variáveis de instância de receberem valores inválidos, como também isola os clientes da classe da representação das variáveis de instância. Portanto, se a representação dos dados se altera (em geral, para reduzir a quantidade de armazenamento necessário, para melhorar o desempenho ou aprimorar a classe de outras maneiras), só as implementações dos métodos precisam ser alteradas – os clientes não precisam ser alterados quanto a interface fornecida pelos métodos permaneça a mesma.*

8.8.1 Executando um *applet* que usa pacotes definidos pelo programador

Depois de compilar as classes na Fig. 8.8 e na Fig. 8.9, você pode executar o *applet* a partir de uma janela de comando, com o comando

```
appletviewer TimeTest5.html
```

Como discutimos quando apresentamos os pacotes antes, neste capítulo, o interpretador pode localizar classes de pacotes no diretório corrente. O **appletviewer** é um aplicativo Java que executa um *applet* Java. Como o interpretador, o **appletviewer** pode carregar classes-padrão de Java e classes de extensão instaladas no computador local. Entretanto, o **appletviewer** não usa o caminho para as classes para localizar classes em pacotes definidos pelo programador. Para um *applet*, tais classes devem estar incluídas com a classe do *applet* em um arquivo compactado denominado arquivo *Java Archive (JAR)*. Lembre-se de que os *applets* normalmente são baixados da Internet para um navegador da Web (ver Capítulo 3 para obter mais informações). Reunir as classes e os pacotes que compõem um *applet* permite que o *applet* e as classes que o suportam sejam baixados como uma unidade e depois executados no navegador (ou através do *Java Plug-In* para os navegadores que não suportam Java 2).

Para reunir as classes das Figs. 8.8 e 8.9, abra uma janela de comando e mude de diretório até a posição na qual **TimeTest5.class** está armazenado. Neste mesmo diretório deve estar o diretório **com** que começa a estrutura de diretório de pacotes para a classe **Time3**. Neste diretório, digite o seguinte comando

```
jar cf TimeTest5.jar TimeTest5.class com\*.*
```

para criar o arquivo JAR. [Nota: esse comando usa **** como o separador de diretórios do *prompt* do MS-DOS. UNIX usaria **/** como o separador de diretórios.] No comando precedente, **jar** é o *utilitário para Java archive* usado para criar arquivos JAR. Em seguida estão as opções para o utilitário **jar - cf**. A letra **c** indica que estamos criando um arquivo JAR. A letra **f** indica que o argumento seguinte na linha de comando (**TimeTest5.jar**) é o nome do arquivo JAR a criar. Seguindo as opções e o nome do arquivo JAR estão os nomes dos arquivos que serão incluídos no arquivo JAR. Especificamos **TimeTest5.class** e **com*.***, indicando que **TimeTest5.class** e todos os arquivos no diretório **com** devem ser incluídos no arquivo JAR. O diretório **com** inicia o pacote que contém o arquivo **.class** para a **Time3**. [Nota: você pode incluir arquivos selecionados especificando o caminho e o nome de arquivo para cada arquivo individual.] É importante que a estrutura de diretório no arquivo JAR coincida com a estrutura de diretório para as classes empacotadas. Portanto, executamos o comando **jar** a partir do diretório no qual **com** está localizado.

Para confirmar que os arquivos foram arquivados diretamente, você pode digitar o comando

```
jar tvf TimeTest5.jar
```

que produz a listagem da Fig. 8.10. No comando precedente, as opções para o utilitário **jar** são **tvf**. A letra **t** indica que o índice do JAR deve ser listado. A letra **v** indica que a saída deve ser prolixia (saída prolixia inclui o tamanho do arquivo em *bytes* e a data e a hora em que cada arquivo foi criado, além da estrutura do diretório e o nome do arquivo). A letra **f** especifica que o argumento seguinte na linha de comando é o arquivo JAR a ser usado.

A única questão restante é especificar o arquivo como parte do arquivo HTML do *applet*. Em exemplos anteriores as marcas **<applet>** tinham a forma

```
<applet code = "NomeDaClasse.class" width = "largura" height = "altura">
</applet>
```

0	Fri May 25 14:13:14 EDT 2001	META-INF/
71	Fri May 25 14:13:14 EDT 2001	META-INF/MANIFEST.MF
2959	Fri May 25 13:42:32 EDT 2001	TimeTest5.class
0	Fri May 18 17:35:18 EDT 2001	com/deitel/
0	Fri May 18 17:35:18 EDT 2001	com/deitel/jhttp4/
0	Fri May 18 17:35:18 EDT 2001	com/deitel/jhttp4/ch08/
1765	Fri May 18 17:35:18 EDT 2001	com/deitel/jhttp4/ch08/Time3.class

Fig. 8.10 Conteúdo de **TimeTest5.jar**.

Para especificar que as classes do *applet* estão localizadas em um arquivo JAR, use uma marca <**applet**> no formato:

```
<applet code = "NomeDaClasse.class" archive = "listaDeArquivos"
        width = "largura" height = "altura">
</applet>
```

O atributo **archive** pode especificar uma lista de arquivos compactados separados por vírgulas para uso em um *applet*. Cada arquivo na lista de **archive** será baixado pelo navegador quando ele encontrar as marcas <**applet**> no documento HTML. Para o *applet* TimeTest5, a marca <**applet**> seria

```
<applet code = "TimeTest5.class" archive = "TimeTest5.jar"
        width = "400" height = "115">
</applet>
```

Tente carregar este *applet* em seu navegador da Web. Lembre-se de que você deve ter um navegador que suporte Java 2 (como o Netscape Navigator 6) ou converter o arquivo HTML para uso com o *Java Plug-In* (como discutido no Capítulo 3).

8.9 Reutilização de software

Os programadores de Java concentram-se na elaboração de novas classes e na reutilização de classes existentes. Existem muitas *bibliotecas de classe* e outras estão sendo desenvolvidas em todo o mundo. O *software* é, então, construído a partir de componentes existentes, bem-definidos, cuidadosamente testados, portáveis e amplamente disponíveis. Esse tipo de reutilização de *software* acelera o desenvolvimento de *softwares* poderosos e de alta qualidade. O *desenvolvimento rápido de aplicações (RAD – rapid applications development)* é de grande interesse atualmente.

Os programadores de Java agora podem escolher entre os milhares de classes da Java API para ajudá-los a implementar os programas Java. Definitivamente, Java não é apenas uma linguagem de programação. É um arcabouço a partir do qual os desenvolvedores Java podem trabalhar para alcançar a reutilização e o desenvolvimento rápido de aplicativos. Os programadores de Java podem se concentrar na tarefa que está em suas mãos quando estão desenvolvendo seus programas e deixar detalhes de baixo nível para as classes da Java API. Por exemplo, para escrever um programa que faça desenhos, um programador de Java não precisa ter conhecimentos de gráficos em todas as plataformas de computação nas quais o programa será executado. Em vez disso, o programador de Java se concentra em aprender os recursos gráficos de Java (que são bastante substanciais e continuam crescendo) e escreve um programa Java que desenha os gráficos, usando classes da Java API como **Graphics**. Quando o programa é executado em um dado computador, é trabalho do interpretador traduzir os comandos Java para os comandos que o computador local possa entender.

As classes da Java API permitem que os programadores de Java tragam novos aplicativos para o mercado mais depressa pelo uso de componentes preexistentes e já testados. Isso não apenas reduz o tempo de desenvolvimento, mas também melhora a capacidade de depurar e manter aplicativos dos programadores. Para tirar proveito dos inúmeros recursos de Java, é essencial que os programadores dediquem tempo para familiarizar-se com a grande variedade de pacotes e classes da Java API. Existem muitos recursos baseados na Web em java.sun.com para ajudá-lo nesta tarefa. O recurso básico para aprender sobre a Java API é a documentação da Java API, que pode ser encontrada em

java.sun.com/j2se/docs/api/index.html

Além disso, java.sun.com oferece muitos outros recursos, incluindo tutoriais, artigos e *sites* específicos para tópicos individuais de Java. Os desenvolvedores de Java também devem se registrar (gratuitamente) na Java Developer Connection

developer.java.sun.com

Este *site* oferece recursos adicionais que os desenvolvedores de Java acharão úteis, incluindo mais tutoriais e artigos e *links* para outros recursos de Java. Consulte o Apêndice B para ver uma lista mais completa de recursos relacionados com Java na Internet e na World Wide Web.



Boa prática de programação 8.6

Evite reinventar a roda. Estude os recursos da Java API. Se a API já contiver uma classe que atende às necessidades de seu programa, use aquela classe em vez de criar a sua própria.

Em geral, para perceber o potencial completo da reutilização de *software*, precisamos aprimorar os esquemas de catalogação, os esquemas de licenciamento, os mecanismos de proteção que asseguram que as cópias mestras das classes não sejam corrompidas, os esquemas de descrição que os projetistas de sistema utilizam para determinar se os objetos existentes atendem às suas necessidades, os mecanismos de navegação que determinam as classes que estão disponíveis e em que grau elas atendem aos requisitos do desenvolvedor de *software*, e assim por diante. Muitos problemas interessantes de pesquisas e desenvolvimento foram solucionados e muitos outros necessitam ser resolvidos; esses problemas acabarão sendo resolvidos porque o valor potencial da reutilização de *software* é enorme.

8.10 Variáveis de instância final

Temos enfatizado repetidamente o *princípio do menor privilégio* como um dos princípios mais fundamentais da boa engenharia de *software*. Vejamos agora uma maneira em que esse princípio se aplica às variáveis de instância.

Algumas variáveis de instância precisam ser modificáveis e algumas não. O programador pode utilizar a palavra-chave **final** para especificar que uma variável não é modificável e que qualquer tentativa de modificar a variável é um erro. Por exemplo,

```
private final int INCREMENT = 5;
```

declara uma variável de instância constante **INCREMENT** do tipo **int** e a inicializa com 5.



Observação de engenharia de software 8.17

Declarar uma variável de instância como **final** ajuda a garantir o princípio do menor privilégio. Se uma variável de instância não deve ser modificada, declare-a como **final** para proibir expressamente a modificação.



Dica de teste e depuração 8.3

Tentativas acidentais de modificar uma variável de instância **final** são capturadas durante a compilação em vez de causar erros em tempo de execução. É sempre preferível eliminar os erros em tempo de compilação, se possível, em vez de permitir que eles escapem para a execução (quando estudos revelaram que o custo de reparo é freqüentemente até 10 vezes mais caro).



Erro comum de programação 8.9

Tentar modificar uma variável de instância **final** depois que é ela inicializada é um erro de sintaxe.

O applet da Fig. 8.11 cria uma variável de instância **final** **INCREMENT** do tipo **int** e inicializa como 5 na sua declaração (linha 15). A variável **final** não pode ser modificada por atribuição depois que ela for inicializada. Essa variável pode ser inicializada em sua declaração ou em cada construtor da classe.



Erro comum de programação 8.10

É um erro de sintaxe não inicializar uma variável de instância **final** em sua declaração ou em cada construtor da classe.

```

1 // Fig. 8.11: Increment.java
2 // Inicializando uma variável final
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class Increment extends JApplet
12     implements ActionListener {

```

Fig. 8.11 Inicializando uma variável **final** (parte 1 de 2).

```

13
14     private int count = 0, total = 0;
15     private final int INCREMENT = 5;    // variável com valor constante
16
17     private JButton button;
18
19     // configura a GUI
20     public void init()
21     {
22         Container container = getContentPane();
23
24         button = new JButton( "Click to increment" );
25         button.addActionListener( this );
26         container.add( button );
27     }
28
29     // adiciona INCREMENT ao total quando o usuário clica no botão
30     public void actionPerformed( ActionEvent actionEvent )
31     {
32         total += INCREMENT;
33         count++;
34         showStatus( "After increment " + count +
35             ": total = " + total );
36     }
37
38 } // fim da classe Increment

```

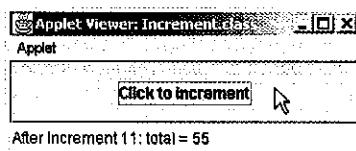


Fig. 8.11 Inicializando uma variável **final** (parte 2 de 2).

A Fig. 8.12 ilustra os erros do compilador produzidos para o programa da Fig. 8.11 se a variável de instância **INCREMENT** é declarada **final**, mas não é inicializada na declaração.

```

Increment.java:11: variable INCREMENT might not have been
    initialized
public class Increment extends JApplet
^
1 error

```

Fig. 8.12 Mensagem de erro do compilador como resultado da não-inicialização de **INCREMENT**.

8.11 Composição: objetos como variáveis de instância de outras classes

Um objeto da classe **AlarmClock** precisa saber quando ele deve soar o alarme, então por que ele não inclui uma referência a um objeto **Time** como membro do objeto **AlarmClock**? Essa capacidade se chama *composição*. A classe pode ter referências a objetos de outras classes como membros.



Observação de engenharia de software 8.18

Uma forma de reutilização de software é a composição, na qual uma classe tem referências a objetos de outras classes como membros.

O próximo programa contém três classes – **Date** (Fig. 8.13), **Employee** (Fig. 8.14) e **EmployeeTest** (Fig. 8.15). A classe **Employee** contém as variáveis de instância **firstName**, **lastName**, **birthDate** e **hireDate**. Os membros **birthDate** e **hireDate** são referências para **Dates** que contêm as variáveis de instância **month**, **day** e **year**. Isso demonstra que uma classe pode conter referências para objetos de outras classes. A classe **EmployeeTest** instancia um **Employee** e inicializa e exibe suas variáveis de instância. O construtor **Employee** (Fig. 8.14, linhas 12 a 20) recebe oito argumentos – **first**, **last**, **birthMonth**, **birthDay**, **birthYear**, **hireMonth**, **hireDay** e **hireYear**. Os argumentos **birthMonth**, **birthDay** e **birthYear** são passados para o construtor **Date** (Fig. 8.13, linhas 13 a 28) para inicializar o objeto **birthDate**, e **hireMonth**, **hireDay** e **hireYear** são passados para o construtor **Date**, para inicializar o objeto **hireDate**.

O objeto-membro não precisa ser inicializado imediatamente com argumentos de construtor. Se uma lista de argumentos vazia for fornecida quando se cria um objeto-membro, o construtor *default* do objeto (ou o construtor sem argumentos, se houver um disponível) será chamado automaticamente. Os valores, se houver algum, estabelecidos pelo construtor *default* (ou pelo construtor sem argumentos) podem então ser substituídos por métodos *set*.

Dica de desempenho 8.2



Inicie os objetos-membros explicitamente durante a construção, passando argumentos apropriados para os construtores dos objetos-membros. Isso elimina a sobrecarga de se inicializar duas vezes os objetos-membros – uma vez quando o construtor default do objeto-membro é chamado e, novamente, quando os métodos set são utilizados para fornecer valores iniciais para o objeto-membro.

Observe que tanto a classe **Date** (Fig. 8.13) como a classe **Employee** (Fig. 8.14) são definidas como parte do pacote **com.deitel.jhtp4.ch08** como especificado na linha 3 de cada arquivo. Uma vez que elas estão no mesmo pacote (isto é, no mesmo diretório), a classe **Employee** não precisa importar a classe **Date** para utilizá-la. Quando o compilador procura o arquivo **Date.class**, o compilador sabe pesquisar o diretório onde **Employee.class** está localizado. As classes em um pacote nunca precisam importar outras classes do mesmo pacote.

```

1 // Fig. 8.13: Date.java
2 // Declaração da classe Date
3 package com.deitel.jhtp4.ch08;
4
5 public class Date extends Object {
6     private int month; // 1 a 12
7     private int day; // 1 a 31, de acordo com o mês
8     private int year; // qualquer ano
9
10    // Construtor: confirma valor adequado para mês;
11    // chama o método checkDay para confirmar
12    // valor adequado para o dia.
13    public Date( int theMonth, int theDay, int theYear )
14    {
15        if ( theMonth > 0 && theMonth <= 12 ) // valida o mês
16            month = theMonth;
17        else {
18            month = 1;
19            System.out.println( "Month " + theMonth +
20                " invalid. Set to month 1." );
21        }
22
23        year = theYear; // poderia validar o ano
24        day = checkDay( theDay ); // valida o dia
25
26        System.out.println(
27            "Date object constructor for date " + toString() );
28    }
29

```

Fig. 8.13 Classe Date (parte 1 de 2).

```

30 // Método utilitário para confirmar se o valor do dia
31 // é adequado, com base no mês e ano.
32 private int checkDay( int testDay )
33 {
34     int daysPerMonth[] =
35         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 31 };
36
37     // verifica se o dia está no intervalo válido para o mês
38     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
39         return testDay;
40
41     // verifica se há anos bissextos
42     if ( month == 2 && testDay == 29 &&
43         ( year % 400 == 0 ||
44          ( year % 4 == 0 && year % 100 != 0 ) ) )
45         return testDay;
46
47     System.out.println( "Day " + testDay +
48         " invalid. Set to day 1." );
49
50     return 1; // deixa o objeto com uma data consistente
51 }
52
53 // Cria um String no formato mês/dia/ano
54 public String toString()
55 {
56     return month + "/" + day + "/" + year;
57 }
58
59 } // fim da classe Date

```

Fig. 8.13 Classe Date (parte 2 de 2).

```

1 // Fig. 8.14: Employee.java
2 // Definição da classe Employee
3 package com.deitel.jhtp4.ch08;
4
5 public class Employee extends Object {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11    // construtor para inicializar nome, data de nascimento e data de contratação
12    public Employee( String first, String last,
13        int birthMonth, int birthDay, int birthYear,
14        int hireMonth, int hireDay, int hireYear )
15    {
16        firstName = first;
17        lastName = last;
18        birthDate = new Date( birthMonth, birthDay, birthYear );
19        hireDate = new Date( hireMonth, hireDay, hireYear );
20    }
21
22    // converte Employee para o formato de String
23    public String toString()
24    {

```

Fig. 8.14 Classe Employee com referências para objetos-membros (parte 1 de 2).

```

25     return lastName + ", " + firstName +
26     " Hired: " + hireDate.toString() +
27     " Birthday: " + birthDate.toString();
28   }
29
30 } // fim da classe Employee

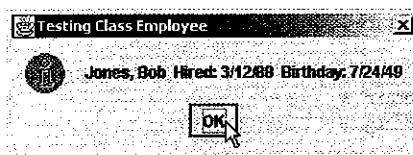
```

Fig. 8.14 Classe Employee com referências para objetos-membros (parte 2 de 2).

```

1 // Fig. 8.15: EmployeeTest.java
2 // Demonstrando um objeto com um objeto-membro
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 // Pacotes Deitel
8 import com.deitel.jhtp4.ch08.Employee;
9
10 public class EmployeeTest {
11
12   // testa a classe Employee
13   public static void main( String args[] )
14   {
15     Employee employee = new Employee( "Bob", "Jones",
16         7, 24, 49, 3, 12, 88 );
17
18     JOptionPane.showMessageDialog( null,
19         employee.toString(), "Testing Class Employee",
20         JOptionPane.INFORMATION_MESSAGE );
21
22     System.exit( 0 );
23   }
24
25 } // fim da classe EmployeeTest

```



Date object constructor for date 7/24/1949
 Date object constructor for date 3/12/1988

Fig. 8.15 Demonstrando um objeto com uma referência para objetos-membros.

8.12 Acesso de pacote

Quando nenhum membro modificador de acesso é oferecido para um método ou uma variável quando estão definidos em uma classe, o método ou a variável são considerados como se tivessem *acesso de pacote*. Se o programa consiste em uma definição de classe, isso não tem nenhum efeito específico no programa. Entretanto, se o programa utiliza múltiplas classes do mesmo pacote (isto é, um grupo de classes relacionadas), essas classes podem acessar todos os métodos de acesso e dados com acesso de pacoteumas das outras diretamente, através de uma referência a um objeto.

*Dica de desempenho 8.3*

O acesso de pacote permite aos objetos de classes diferentes interagir sem a necessidade de métodos set e get que forneçam acesso a dados, eliminando, assim, parte do custo de chamada de métodos.

Vamos considerar um exemplo mecânico de acesso de pacote. O aplicativo da Fig. 8.16 contém duas classes – a classe de aplicativo **PackageDataTest** (linhas 8 a 34) e a classe **PackageData** (linhas 37 a 54). Na definição da classe **PackageData**, as linhas 38 e 39 declaram as variáveis de instância **number** e **string** sem modificadores de acesso a membro; portanto, são variáveis de instância com acesso de pacote. O método **main** do aplicativo **PackageDataTest** cria uma instância da classe **PackageData** (linha 13) para demonstrar a capacidade de modificar as variáveis de instância de **PackageData** diretamente (como mostrado nas linhas 20 e 21). Os resultados da modificação podem ser vistos na janela de saída.

Quando você compila esse programa, o compilador produz dois arquivos separados – um arquivo **.class** para a classe **PackageData** e um arquivo **.class** para a classe **PackageDataTest**. Cada classe Java tem seu próprio arquivo **.class**. Esses dois arquivos **.class** são colocados automaticamente no mesmo diretório pelo compilador e são considerados parte do mesmo pacote (eles certamente estão relacionados porque estão no mesmo arquivo). Uma vez que fazem parte do mesmo pacote, a classe **PackageDataTest** está autorizada a modificar os dados de objetos da classe **PackageData** com acesso de pacote.

```

1 // Fig. 8.16: PackageDataTest.java
2 // As classes no mesmo pacote (i.e., o mesmo diretório) podem usar
3 // dados com acesso de pacote de outras classes do mesmo pacote.
4
5 // Pacotes de extensão de Java
6 import javax.swing.JOptionPane;
7
8 public class PackageDataTest {
9
10    // Pacotes de extensão de Java
11    public static void main( String args[] ) {
12        PackageData packageData = new PackageData();
13
14        // acrescenta representação como String de packageData à saída
15        String output =
16            "After instantiation:\n" + packageData.toString();
17
18        // muda dados com acesso de pacote no objeto packageData
19        packageData.number = 77;
20        packageData.string = "Good bye";
21
22        // acrescenta representação como String de packageData à saída
23        output += "\nAfter changing values:\n" +
24            packageData.toString();
25
26        JOptionPane.showMessageDialog( null, output,
27            "Demonstrating Package Access",
28            JOptionPane.INFORMATION_MESSAGE );
29
30        System.exit( 0 );
31    }
32
33
34 } // fim da classe PackageDataTest
35
36 // classe com variáveis de instância com acesso de pacote
37 class PackageData {
38     int number;      // variável de instância com acesso de pacote
39     String string;  // variável de instância com acesso de pacote

```

Fig. 8.16 Acesso de pacote a membros de uma classe (parte 1 de 2).

```

40
41     // construtor
42     public PackageData()
43     {
44         number = 0;
45         string = "Hello";
46     }
47
48     // converte objeto PackageData para representação como String
49     public String toString()
50     {
51         return "number: " + number + "    string: " + string;
52     }
53
54 } // fim da classe PackageData

```

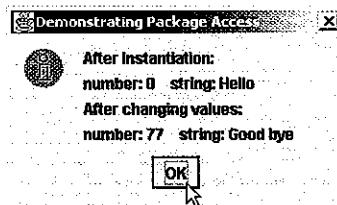


Fig. 8.16 Acesso de pacote a membros de uma classe (parte 2 de 2).



Observação de engenharia de software 8.19

Algumas pessoas na comunidade OOP acreditam que o acesso de pacote corrompe o ocultamento das informações e enfraquece o valor da abordagem do projeto orientado a objetos, porque o programador precisa assumir a responsabilidade pela verificação de erros e pela validação dos dados em qualquer código que manipule os membros de dados com acesso de pacote.

8.13 Utilizando a referência `this`

Quando um método de uma classe faz referência a outro membro dessa classe para um objeto específico dessa classe, como Java assegura que o objeto adequado recebe referência? A resposta é que cada objeto tem acesso a uma referência a ele próprio – chamada de referência `this`.

Utiliza-se a referência `this` implicitamente para fazer referências às variáveis de instância e aos métodos de um objeto. Começamos com um exemplo simples de uso da referência `this` explicitamente; em um exemplo subsequente, mostraremos alguns exemplos substanciais e sutis de utilização de `this`.



Dica de desempenho 8.4

Java economiza espaço de armazenamento na memória mantendo somente uma cópia de cada método por classe; esse método é invocado por todos os objetos dessa classe. Cada objeto, por outro lado, tem sua própria cópia das variáveis de instância da classe.

O aplicativo da Fig. 8.17 demonstra a utilização explícita e implícita da referência `this` para permitir ao método `main` da classe `ThisTest` exibir os dados `private` de um objeto `SimpleTime`.

```

1 // Fig. 8.17: ThisTest.java
2 // Usando a referência this para se referir
3 // a variáveis de instância e métodos.
4

```

Fig. 8.17 Usando a referência `this` implícita e explicitamente (parte 1 de 3).

```

5 // Pacotes do núcleo de Java
6 import java.text.DecimalFormat;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class ThisTest {
12
13     // testa a classe SimpleTime
14     public static void main( String args[] )
15     {
16         SimpleTime time = new SimpleTime( 12, 30, 19 );
17
18         JOptionPane.showMessageDialog( null, time.buildString(),
19             "Demonstrating the \"this\" Reference",
20             JOptionPane.INFORMATION_MESSAGE );
21
22         System.exit( 0 );
23     }
24
25 } // fim da classe ThisTest
26
27 // classe SimpleTime demonstra a referência "this"
28 class SimpleTime {
29     private int hour, minute, second;
30
31     // construtor usa nomes de parâmetros idênticos a nomes de variáveis
32     // de instância, de modo que a referência "this" é necessária para
33     // distinguir variáveis de instância e parâmetros
34     public SimpleTime( int hour, int minute, int second )
35     {
36         this.hour = hour;      // configura a hora deste ("this") objeto
37         this.minute = minute; // configura o minuto deste ("this") objeto
38         this.second = second; // configura o segundo deste ("this") objeto
39     }
40
41     // chama toString explicitamente através da referência "this", explicitamente
42     // através de uma referência "this" implícita, implicitamente através de "this"
43     public String buildString()
44     {
45         return "this.toString(): " + this.toString() +
46             "\ntoString(): " + toString() +
47             "\nthis (with implicit toString() call): " + this;
48     }
49
50     // converte SimpleTime para formato de String
51     public String toString()
52     {
53         DecimalFormat twoDigits = new DecimalFormat( "00" );
54
55         // "this" não é necessário porque toString não tem
56         // variáveis locais com o mesmo nome que as variáveis de instância
57         return twoDigits.format( this.hour ) + ":" +
58             twoDigits.format( this.minute ) + ":" +
59             twoDigits.format( this.second );
60     }
61
62 } // fim da classe SimpleTime

```

Fig. 8.17 Usando a referência `this` implicitamente e explicitamente (parte 2 de 3).

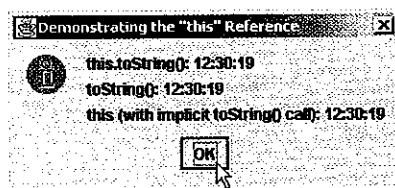


Fig. 8.17 Usando a referência `this` implícita e explicitamente (parte 3 de 3).

A classe `SimpleTime` (linhas 8 a 60) define três variáveis de instância `private` – `hour`, `minute` e `second`. O construtor (linhas 34 a 39) recebe três argumentos `int` para inicializar um objeto `SimpleTime`. Repare que os nomes de parâmetro para o construtor são os mesmos que os nomes das variáveis de instância. Lembre-se de que uma variável local de um método com o mesmo nome que uma variável de instância de uma classe oculta a variável de instância no escopo do método. Por essa razão, utilizamos a referência `this` explicitamente para fazer referência às variáveis de instância nas linhas 36 a 38.



Erro comum de programação 8.11

Em um método em que um parâmetro de método tem o mesmo nome que um dos membros da classe, utilize `this` explicitamente se você quiser acessar o membro da classe; caso contrário, você incorretamente fará referência ao parâmetro do método.



Boa prática de programação 8.7

Evite utilizar nomes de parâmetros de método que entrem em conflito com nomes de membros da classe.

O método `buildString` (linhas 43 a 48) devolve um `String` criado com uma instrução que utiliza a referência `this` de três maneiras. A linha 45 invoca explicitamente o método `toString` da classe através de `this.toString()`. A linha 46 utiliza implicitamente a referência `this` para realizar a mesma tarefa. A linha 47 acrescenta `this` ao `string` que será devolvido. Lembre-se de que a referência `this` é uma referência a um objeto – o objeto `SimpleTime` que está sendo manipulado atualmente. Como antes, qualquer referência adicionada a um `String` resulta em uma chamada ao método `toString` para o objeto mencionado. O método `buildString` é invocado na linha 18 para exibir os resultados das três chamadas a `toString`. Observe que a mesma hora é exibida em todas as três linhas da saída, porque todas as três chamadas a `toString` são para o mesmo objeto.

Outra utilização da referência `this` é permitir *chamadas de método concatenadas* (também conhecidas como *chamadas de método em cascata* ou *encadeamento de chamadas*). A Fig. 8.18 ilustra a devolução de uma referência a um objeto `Time4` para permitir que as chamadas aos métodos da classe `Time4` sejam concatenadas. Os métodos `setTime` (linhas 50 a 57), `setHour` (linhas 60 a 65), `setMinute` (linhas 68 a 74) e `setSecond` (linhas 77 a 83) têm um tipo de retorno do `Time4` e como sua última instrução

return this;

para indicar que uma referência para o objeto `Time4` que está sendo manipulado deve ser retornada para o chamar do método.

```

1 // Fig. 8.18: Time4.java
2 // Definição da classe Time4
3 package com.deitel.jhtp4.ch08;
4
5 // Pacotes do núcleo de Java
6 import java.text.DecimalFormat;
7
8 public class Time4 extends Object {
9     private int hour;      // 0 a 23

```

Fig. 8.18 Classe `Time4` usando `this` para permitir chamadas de métodos encadeadas (parte 1 de 3).

```

10  private int minute;    // 0 a 59
11  private int second;   // 0 a 59
12
13  // construtor Time4 inicializa cada variável de instância
14  // com zero. Assegura que o objeto Time inicie em um
15  // estado consistente.
16  public Time4()
17  {
18      this.setTime( 0, 0, 0 );
19  }
20
21  // construtor Time4: hora informada, minuto e
22  // segundo colocados em 0 por default
23  public Time4( int hour )
24  {
25      this.setTime( hour, 0, 0 );
26  }
27
28  // construtor Time4: hora e minuto informados,
29  // segundo colocado em 0 por default
30  public Time4( int hour, int minute )
31  {
32      this.setTime( hour, minute, 0 );
33  }
34
35  // construtor Time4: hora, minuto e segundo informados
36  public Time4( int hour, int minute, int second )
37  {
38      this.setTime( hour, minute, second );
39  }
40
41  // construtor Time4: outro objeto Time4 é fornecido.
42  public Time4( Time4 time )
43  {
44      this.setTime( time.getHour(), time.getMinute(),
45                  time.getSecond() );
46  }
47
48  // Métodos set
49  // configuram um novo valor Time usando o tempo universal
50  public Time4 setTime( int hour, int minute, int second )
51  {
52      this.setHour( hour );           // configura a hora
53      this.setMinute( minute );     // configura o minuto
54      this.setSecond( second );     // configura o segundo
55
56      return this; // permite encadeamento
57  }
58
59  // valida e configura a hora
60  public Time4 setHour( int hour )
61  {
62      this.hour = ( hour >= 0 && hour < 24 ? hour : 0 );
63
64      return this; // permite encadeamento
65  }
66
67  // valida e configura o minuto
68  public Time4 setMinute( int minute )
69  {

```

Fig. 8.18 Classe Time4 usando this para permitir chamadas de métodos encadeadas (parte 2 de 3).

```

70     this.minute =
71         ( minute >= 0 && minute < 60 ) ? minute : 0;
72
73     return this; // permite encadeamento
74 }
75
76 // valida e configura o segundo
77 public Time4 setSecond( int second )
78 {
79     this.second =
80         ( second >= 0 && second < 60 ) ? second : 0;
81
82     return this; // permite encadeamento
83 }
84
85 // Métodos get
86 // obtém valor de hora
87 public int getHour() { return this.hour; }
88
89 // obtém valor de minuto
90 public int getMinute() { return this.minute; }
91
92 // obtém valor de segundo
93 public int getSecond() { return this.second; }
94
95 // converte para String no formato de hora universal
96 public String toUniversalString()
97 {
98     DecimalFormat twoDigits = new DecimalFormat( "00" );
99
100    return twoDigits.format( this.getHour() ) + ":" +
101        twoDigits.format( this.getMinute() ) + ":" +
102        twoDigits.format( this.getSecond() );
103 }
104
105 // converte para String no formato de hora padrão
106 public String toString()
107 {
108     DecimalFormat twoDigits = new DecimalFormat( "00" );
109
110    return ( this.getHour() == 12 || this.getHour() == 0 ?
111        12 : this.getHour() % 12 ) + ":" +
112        twoDigits.format( this.getMinute() ) + ":" +
113        twoDigits.format( this.getSecond() ) +
114        ( this.getHour() < 12 ? " AM" : " PM" );
115 }
116
117 } // fim da classe Time4

```

Fig. 8.18 Classe `Time4` usando `this` para permitir chamadas de métodos encadeadas (parte 3 de 3).

O exemplo demonstra novamente a utilização explícita da referência `this` dentro do corpo de uma classe. Na classe `Time4`, cada utilização de uma variável de instância da classe e cada chamada a outro método na classe `Time4` utiliza a referência `this` explicitamente. A maioria dos programadores prefere não utilizar a referência `this` a menos que ela seja exigida ou ajude a esclarecer uma parte do código.



Boa prática de programação 8.8

Utilizar `this` explicitamente pode aumentar a clareza do programa em alguns contextos em que `this` é opcional.

Na classe de aplicativo `TimeTest6` (Fig. 8.19), as linhas 18 e 26 demonstram o encadeamento de chamadas de método. Por que a técnica de retornar a referência `this` funciona? Vamos discutir a linha 18. O operador ponto (`.`) associa da esquerda para a direita, de modo que a expressão

```
time.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
```

primeiro avalia `time.setHour(18)` e depois retorna uma referência ao objeto `time` como resultado dessa chamada de método. Sempre que você tiver uma referência em um programa (mesmo como o resultado de uma chamada de método), a referência pode ser seguida por um operador ponto e uma chamada a um dos métodos do tipo da referência. Portanto, a expressão restante é interpretada como

```
time.setMinute( 30 ).setSecond( 22 );
```

A chamada `time.setMinute(30)` é executada e retorna uma referência para `time`. A expressão restante é interpretada como

```
time.setSecond( 22 );
```

Quando a instrução está completa, a hora é 18 para `hour`, 30 para `minute` e 22 para `second`.

Observe que as chamadas na linha 26

```
time.setTime( 20, 20, 20 ).toString();
```

também utilizam o recurso da concatenação. Essas chamadas de método devem aparecer nessa ordem nessa expressão, porque `toString`, como definido na classe, não retorna uma referência a um objeto `Time4`. Colocar a chamada para `toString` antes da chamada a `setTime` causa um erro de sintaxe. Observe que `toString` retorna uma referência a um objeto `String`. Portanto, um método de classe `String` poderia ser concatenado ao final da linha 26.

```

1 // Fig. 8.19: TimeTest6.java
2 // Encadeando chamadas a métodos usando a referência this
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 // Pacotes Deitel
8 import com.deitel.jhtp4.ch08.Time4;
9
10 public class TimeTest6 {
11
12     // testa encadeamento de chamadas a métodos com objeto da classe Time4
13     public static void main( String args[] )
14     {
15         Time4 time = new Time4();
16
17         // encadeia chamadas a setHour, setMinute e setSecond
18         time.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
19
20         // usa encadeamento de chamadas de métodos para configurar
21         // nova hora e obter representação como String da nova hora
22         String output =
23             "Universal time: " + time.toUniversalString() +
24             "\nStandard time: " + time.toString() +
25             "\n\nNew standard time: " +
26             time.setTime( 20, 20, 20 ).toString();
27
28         JOptionPane.showMessageDialog( null, output,
29             "Chaining Method Calls",
30             JOptionPane.INFORMATION_MESSAGE );
31
32         System.exit( 0 );
33     }

```

Fig. 8.19 Concatenando chamadas a métodos (parte 1 de 2).

```

34
35 } // fim da classe TimeTest6

```

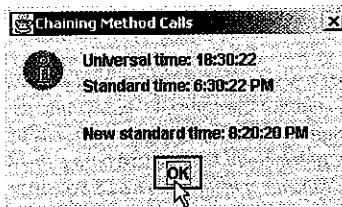


Fig. 8.19 Concatenando chamadas a métodos (parte 2 de 2).

Observe que o objetivo do exemplo nas Figuras 8.18 e 8.19 é demonstrar o mecanismo de chamadas a métodos concatenados. Muitos métodos em Java devolvem referências para objetos que podem ser usadas da maneira mostrada aqui. É importante entender chamadas a métodos concatenadas porque elas aparecem freqüentemente em programas Java.



Boa prática de programação 8.9

Para obter maior clareza nos programas, evite o uso de chamadas a métodos concatenadas.

8.14 Finalizadores

Vimos que os métodos construtores são capazes de inicializar dados em um objeto de uma classe quando a classe é criada. Freqüentemente, os construtores adquirem vários recursos de sistema, como a memória (quando o operador `new` é utilizado). Precisamos de uma maneira disciplinada de devolver recursos para o sistema quando eles não são mais necessários, para evitar desperdício de recursos. O recurso mais comum adquirido pelos construtores é a memória. Java realiza *coleta de lixo* automática da memória para ajudar a devolver a memória para o sistema. Quando um objeto não é mais utilizado no programa (isto é, não há referências para o objeto), o objeto é *marcado para coleta de lixo*. A memória para esse objeto pode ser reivindicada quando o *coletor de lixo* é executado. Portanto, desperdícios de memória, que são comuns em outras linguagens como C e C++ (uma vez que a memória não é reivindicada automaticamente nessas linguagens), são menos prováveis de ocorrer em Java. Entretanto, podem ocorrer outros desperdícios de recursos.



Dica de desempenho 8.5

O uso extensivo de variáveis locais que fazem referência para objetos pode degradar o desempenho. Quando uma variável local é a única referência para um objeto, o objeto é marcado para coleta de lixo quando a variável local fica fora de escopo. Se isto acontece freqüentemente em períodos de tempo curtos, grandes quantidades de objetos podem ser marcados para coleta de lixo, resultando em um ônus para o desempenho do coletor de lixo.

Cada classe em Java pode ter um *método finalizador* que retorna recursos para o sistema. É garantido que o método finalizador para um objeto é chamado para realizar uma *limpeza de terminação* no objeto imediatamente antes de o coletor de lixo reivindicar a memória para o objeto. O método finalizador de uma classe sempre tem o nome `finalize`, não recebe parâmetros e não retorna nenhum valor (isto é, seu tipo de retorno é `void`). Uma classe deve ter apenas um método `finalize`, que não retorna argumentos. O método `finalize` é originalmente definido na classe `Object` como um marcador de lugar que não faz nada. Isso assegura que cada classe tenha um método `finalize` para ser chamado pelo coletor de lixo.



Boa prática de programação 8.10

A última instrução em um método `finalize` deve sempre ser `super.finalize()`; para assegurar que o método `finalize` da superclasse seja chamado.



Observação de Engenharia de Software 8.20

Não é garantido que o coletor de lixo seja executado; portanto, não há garantia de que o método `finalize` de um objeto seja chamado. Você não deve projetar classes que confiem na chamada do método `finalize` de um objeto pelo coletor de lixo para liberar recursos do sistema.

Não foram fornecidos finalizadores para as classes apresentadas até agora. Na verdade, os finalizadores são raramente utilizados em aplicativos Java nas empresas. Veremos um exemplo de método `finalize` e discutiremos mais sobre o coletor de lixo adiante na Fig. 8.20.



Dica de teste e depuração 8.4

Diversos desenvolvedores profissionais de Java que revisaram este livro indicaram que o método `finalize` não é útil em aplicações de Java nas empresas, porque não há garantia de que ele seja chamado. Por este motivo, você deve procurar qualquer ocorrência do método `finalize` em seus programas Java para assegurar que o programa não dependa de chamadas para o método `finalize` para a liberação adequada de recursos. Na verdade, você pode pensar em remover inteiramente o método `finalize` e usar outras técnicas para assegurar a liberação de recursos apropriada. Apresentamos uma destas técnicas no Capítulo 14.

8.15 Membros de classe static

Cada objeto de uma classe tem sua própria cópia de todas as variáveis de instância da classe. Em certos casos, apenas uma cópia de uma variável particular deve ser compartilhada por todos os objetos de uma classe. Utiliza-se uma variável de classe `static` por essas e outras razões. A variável de classe `static` representa *informações de escopo de classe* – todos os objetos da classe compartilham a mesma parte dos dados. A declaração de um membro `static` inicia com a palavra-chave `static`.

Vamos motivar a necessidade para dados `static` com escopo de classe com um exemplo de vídeo game. Suponha que tivéssemos um vídeo game com `Marcianos` e outras criaturas do espaço. Cada `Marciano` tende a ser corajoso e disposto a atacar outras criaturas espaciais quando está ciente de que há pelo menos cinco `Marcianos` presentes. Se houver menos que cinco `Marcianos` presentes, todos os `Marcianos` tornam-se covardes. Assim, cada `Marciano` precisa saber a `quantidadeDeMarcianos`. Poderíamos dotar a classe `Marciano` com `quantidadeDeMarcianos` como dado de instância. Se fizermos isso, então cada `Marciano` terá uma cópia separada dos dados de instância e toda vez que criarmos um novo `Marciano` teremos de atualizar a variável de instância `quantidadeDeMarcianos` em cada `Marciano`. Isso desperdiça espaço, com as cópias redundantes, e tempo, para atualizar as cópias separadas. Em vez disso, declararemos que `quantidadeDeMarcianos` será `static`. Isso torna `quantidadeDeMarcianos` um dado com escopo de classe. Cada `Marciano` pode ver a `quantidadeDeMarcianos` como se fosse dado de instância de `Marciano`, mas apenas uma cópia da `quantidadeDeMarcianos` estática é mantida por Java. Isso economiza espaço. Economizamos tempo fazendo o construtor `Marciano` incrementar a `quantidadeDeMarcianos` para cada objeto `Marciano`.



Dica de desempenho 8.6

Utilize variáveis de classe `static` para economizar memória quando for suficiente uma única cópia dos dados.

Embora as variáveis de classe `static` possam se parecer com variáveis globais, as variáveis de classe `static` têm escopo de classe. Os membros de classe `public static` de uma classe podem ser acessados através de uma referência a qualquer objeto dessa classe, ou eles podem ser acessados pelo nome da classe, utilizando o operador ponto (por exemplo, `Math.random()`). O membro de classe `private static` de uma classe pode ser acessado somente através de métodos da classe. Na verdade, os membros de classe `static` existem mesmo quando nenhum objeto dessa classe existe – eles estão disponíveis logo que a classe é carregada na memória durante a execução. Para acessar um membro de classe `public static` quando não existe nenhum objeto da classe, simplesmente use o nome da classe e o operador ponto como prefixos para o nome do membro da classe. Para acessar um membro de classe `private static` quando não existe nenhum objeto da classe, deve ser fornecido um método `public static` e o método deve ser chamado, prefixando seu nome com o nome da classe e o operador ponto.

Nosso próximo programa define duas classes – `Employee` (Fig. 8.20) e `EmployeeTest` (Fig. 8.21). A classe `Employee` define uma variável de classe `private static` e um método `public static`. A variável de classe `count` (Fig. 8.20, linha 6) é inicializada com zero por *default*. A variável de classe `count` mantém uma contagem do número de objetos da classe `Employee` que foram instanciados e residem atualmente na memória. Isso inclui objetos que já foram marcados para coleta de lixo, mas ainda não foram reivindicados.

```

1 // Fig. 8.20: Employee.java
2 // Definição da classe Employee
3 public class Employee extends Object {
4     private String firstName;
5     private String lastName;
6     private static int count; // número de objetos na memória
7
8     // inicializa funcionário, soma 1 a static count e envia
9     // String para a saída indicando que o construtor foi chamado
10    public Employee( String first, String last )
11    {
12        firstName = first;
13        lastName = last;
14
15        ++count; // incrementa static count de funcionários
16        System.out.println( "Employee object constructor: " +
17                           firstName + " " + lastName );
18    }
19
20    // subtrai 1 de static count quando o coletor de lixo
21    // chama finalize para limpar o objeto e envia String
22    // para a saída, indicando que finalize foi chamado
23    protected void finalize()
24    {
25        --count; // decrementa static count de funcionários
26        System.out.println( "Employee object finalizer: " +
27                           firstName + " " + lastName + "; count = " + count );
28    }
29
30    // obtém primeiro nome
31    public String getFirstName()
32    {
33        return firstName;
34    }
35
36    // obtém sobrenome
37    public String getLastname()
38    {
39        return lastName;
40    }
41
42    // método static para obter o valor de static count
43    public static int getCount()
44    {
45        return count;
46    }
47
48 } // fim da classe Employee

```

Fig. 8.20 Classe `Employee` que usa uma variável de classe `static` para manter uma contagem do número de objetos `Employee` presentes na memória.

Quando existem objetos da classe `Employee`, pode-se utilizar o membro `count` em qualquer método de um objeto `Employee` – nesse exemplo, `count` é incrementado (linha 15) pelo construtor e decrementado (linha 25) pelo finalizador. Quando não existe nenhum objeto da classe `Employee`, o membro `count` ainda pode ser mencionado, mas somente através de uma chamada ao método `public static getCount` (linhas 43 a 46), como em:

`Employee.getCount()`

que determina o número de objetos `Employee` atualmente na memória. Observe que, quando não há nenhum objeto instanciado no programa, utiliza-se a chamada de método `Employee.getCount()`. Entretanto, quando há

objetos instanciados, o método `getCount` também poder ser chamado por uma referência a um dos objetos, como em

```
e1.getCount()
```



Boa prática de programação 8.11

Sempre invoque os métodos `static` utilizando o nome da classe e o operador ponto `(.)`. Isso enfatiza a outros programadores que leem seu código que o método que está sendo chamado é um método `static`.

Observe que a classe `Employee` tem um método `finalize` (linhas 23 a 28). Esse método é incluído para mostrar quando ele é chamado pelo coletor de lixo em um programa. O método `finalize` normalmente é declarado como `protected`, então não faz parte dos serviços `public` de uma classe. Discutiremos o modificador de acesso `protected` em detalhes no Capítulo 9.

O método `main` do aplicativo `EmployeeTest` (Fig. 8.21) instancia dois objetos `Employee` (linhas 16 e 17). Quando o construtor de cada do objeto `Employee` é invocado, as linhas 12 e 13 da Fig. 8.20 armazenam referências para os objetos `String` que contêm o primeiro nome e o sobrenome desse `Employee`. Observe que essas duas instruções *não* fazem cópias dos argumentos `String` originais. Na verdade, os objetos `String` em Java são *imutáveis* – eles não podem ser modificados depois de criados (a classe `String` não fornece nenhum método `set`). Uma vez que uma referência não pode ser utilizada para modificar um `String`, é seguro ter muitas referências para um objeto `String` em um programa Java. Normalmente, esse não é o caso para a maioria das outras classes em Java. Se os objetos `String` de Java são imutáveis, por que somos capazes de usar os operadores `+ e +=` para concatenar `Strings`? Como discutimos no Capítulo 10, a operação de concatenação de `Strings` na verdade resulta na criação de novos objetos `String` que contêm os valores concatenados. Os objetos `String` originais, na verdade, não são modificados.

Quando `main` terminou sua tarefa com os dois objetos `Employee`, as referências `e1` e `e2` são configuradas como `null` nas linhas 37 e 38. Nesse ponto, as referências `e1` e `e2` não fazem mais referência aos objetos que foram instanciados nas linhas 16 e 17. Isso *marca os objetos para a coleta de lixo*, uma vez que não há mais referências para os objetos no programa.

```

1 // Fig. 8.21: EmployeeTest.java
2 // Testa a classe Employee com uma variável de classe static,
3 // um método de classe static e memória dinâmica
4 import javax.swing.*;
5
6 public class EmployeeTest {
7
8     // testa a classe Employee
9     public static void main( String args[] ) {
10
11         // prova que a contagem é 0 antes de criar Employees
12         String output = "Employees before instantiation: " +
13             Employee.getCount();
14
15         // cria dois Employees; a contagem deve ser 2
16         Employee e1 = new Employee( "Susan", "Baker" );
17         Employee e2 = new Employee( "Bob", "Jones" );
18
19         // Prova que a contagem é 2 após criar dois Employees.
20         // Nota: métodos static devem ser chamados somente através
21         // do nome da classe para a classe na qual eles estão definidos.
22         output += "\n\nEmployees after instantiation: " +
23             "\nvia e1.getCount(): " + e1.getCount() +
24             "\nvia e2.getCount(): " + e2.getCount() +
25             "\nvia Employee.getCount(): " + Employee.getCount();

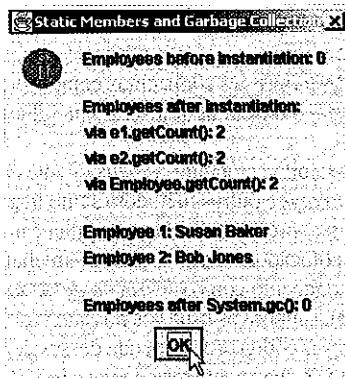
```

Fig. 8.21 Usando uma variável de classe `static` para manter a contagem do número de objetos de uma classe (parte 1 de 2).

```

26
27     // obtém os nomes dos funcionários
28     output += "\n\nEmployee 1: " + e1.getFirstName() +
29         " " + e1.getLastName() + "\nEmployee 2: " +
30         e2.getFirstName() + " " + e2.getLastName();
31
32     // Se existe apenas uma referência para cada Employee
33     // (como neste exemplo), a instrução a seguir marca
34     // aqueles objetos para coleta de lixo. Caso contrário,
35     // esta instrução simplesmente decrementa a contagem de
36     // referências para cada objeto.
37     e1 = null;
38     e2 = null;
39
40     System.gc(); // sugere uma chamada para o coletor de lixo
41
42     // Mostra a contagem de Employee após chamar o coletor de lixo.
43     // A contagem exibida pode ser 0, 1 ou 2, dependendo de se o
44     // coletor de lixo foi executado imediatamente e da
45     // quantidade de objetos Employee que ele coleta.
46     output += "\n\nEmployees after System.gc(): " +
47         Employee.getCount();
48
49     JOptionPane.showMessageDialog( null, output,
50         "Static Members and Garbage Collection",
51         JOptionPane.INFORMATION_MESSAGE );
52
53     System.exit( 0 );
54 }
55
56 } // fim da classe EmployeeTest

```



```

Employee object constructor: Susan Baker
Employee object constructor: Bob Jones
Employee object finalizer: Susan Baker; count = 1
Employee object finalizer: Bob Jones; count = 0

```

Fig. 8.21 Usando uma variável de classe `static` para manter a contagem do número de objetos de uma classe (parte 2 de 2).

Em algum momento, o coletor de lixo reivindica a memória para esses objetos (ou a memória é reivindicada pelo sistema operacional quando o programa termina). Uma vez que não é garantido quando o coletor de lixo será executado, fazemos uma chamada explícita para o coletor de lixo com a linha 40, que utiliza o método `public`

`static gc` da classe `System` (pacote `java.lang`) para indicar que o coletor de lixo deve dar o melhor de si em uma tentativa de coletar objetos que tenham sido marcados para coleta de lixo. Entretanto, isso é apenas uma tentativa – é possível que nenhum objeto ou nenhum subconjunto dos objetos do lixo sejam coletados. Em nosso exemplo, o coletor de lixo foi executado antes de as linhas 49 a 51 exibirem os resultados do programa. A última linha da saída indica que o número dos objetos `Employee` na memória é 0 depois da chamada para `System.gc()`. Além disso, as últimas duas linhas da saída da janela de comando mostram que o objeto `Employee` para `Susan Baker` foi finalizado antes do objeto `Employee` para `Bob Jones`. Lembre-se de que não há garantia de que o coletor de lixo vá ser executado quando `System.gc()` é invocado e nem há garantia de que ele colete objetos em uma ordem específica, de modo que é possível que a saída desse programa em seu sistema possa ser diferente.

[Nota: o método declarado `static` não pode acessar membros de classe não-`static`. Diferentemente dos métodos não-`static`, o método `static` não tem a referência `this` porque as variáveis de classe `static` e os métodos de classe `static` existem independentemente de qualquer objeto de uma classe e antes de qualquer objeto da classe ter sido instanciado.]



Erro comum de programação 8.12

Usar uma referência `this` em um método `static` é um erro de sintaxe.



Erro comum de programação 8.13

É um erro de sintaxe um método `static` chamar um método de instância ou acessar uma variável de instância.



Observação de engenharia de software 8.21

Qualquer variável de classe `static` e qualquer método de classe `static` existem e podem ser utilizados mesmo se nenhum objeto dessa classe tiver sido instanciado.

8.16 Abstração de dados e encapsulamento

As classes normalmente ocultam os detalhes de sua implementação dos clientes das classes. Isso se chama *encapsulamento* ou *ocultamento de informações*. Como exemplo de encapsulamento, vamos considerar uma estrutura de dados chamada *pilha*.

Pense em uma pilha como uma pilha de pratos. Quando um prato é colocado na pilha, ele é sempre colocado no topo (*inserir* – *push* – o prato sobre a pilha) e, quando um prato é removido da pilha, ele é sempre removido do topo (*retirar* – *pop* – o prato da pilha). As pilhas são conhecidas como *estruturas de dados do tipo último a entrar, primeiro a sair (LIFO – last-in, first-out)* – o último item inserido na pilha é o primeiro item que será removido da pilha.

O programador pode criar uma classe pilha e ocultar de seus clientes a implementação da pilha. As pilhas podem ser implementadas facilmente com *arrays* e outros métodos (como listas encadeadas; veja o Capítulo 19 e o Capítulo 20). O cliente de uma classe pilha não precisa saber como a pilha é implementada. O cliente simplesmente requer que, quando os itens de dados são colocados na pilha, eles sejam chamados novamente na ordem do último a entrar, primeiro a sair. Esse conceito é conhecido como *abstração de dados* e as classes Java definem tipos abstratos de dados (*ADTs* – *abstract data types*). Embora os usuários possam vir a conhecer os detalhes de como uma classe é implementada, eles não podem escrever código que dependa desses detalhes. Isso significa que uma classe particular (como a que implementa uma pilha e suas operações de *inserir* e *retirar*) pode ser substituída por outra versão sem afetar o resto do sistema, contanto que os serviços públicos daquela classe não se alterem (isto é, cada método ainda tem o mesmo nome, o mesmo tipo de retorno e a mesma lista de parâmetros na nova definição de classe).

O trabalho de uma linguagem de alto nível é criar uma visualização conveniente para os programadores utilizarem. Não há uma visualização-padrão única aceita – essa é uma razão pela qual existem tantas linguagens de programação. A programação orientada a objetos em Java apresenta mais uma perspectiva.

A maioria das linguagens de programação enfatiza as ações. Nessas linguagens, os dados existem como suporte para as ações que os programas precisam executar. Os dados são “menos interessantes” que as ações, de qualquer maneira. Os dados são “crus”. Há somente alguns tipos de dados predefinidos e é difícil para os programadores criar seus próprios tipos de dados novos.

Essa visão muda com Java e com o estilo de programação orientado a objetos. Java aumenta a importância dos dados. A principal atividade em Java é criar novos tipos de dados (isto é, classes) e expressar as interações entre objetos desses tipos de dados.

Para se mover nessa direção, a comunidade de linguagens de programação precisou formalizar algumas noções sobre os dados. A formalização que consideramos é a noção dos *tipos abstratos de dados* (*ADTs*). Os ADTs recebem tanta atenção hoje quanto a programação estruturada recebeu nas duas últimas décadas. Os ADTs não substituem a programação estruturada. Em vez disso, fornecem uma formalização adicional para melhorar ainda mais o processo de desenvolvimento de programas.

O que é um tipo abstrato de dado? Considere o tipo predefinido `int`. O que vem à mente é a noção de um inteiro em matemática, mas os `int`s em um computador não é exatamente o que um inteiro é em matemática. Em particular, os `int`s dos computadores são normalmente bem limitados em tamanho. Por exemplo, o `int` em uma máquina de 32 bits está limitado ao intervalo -2 bilhões a +2 bilhões, aproximadamente. Se o resultado de cálculo cair fora desse intervalo, ocorre um erro e a máquina responde de alguma maneira dependente de máquina, incluindo a possibilidade de produzir “silenciosamente” um resultado incorreto. Os inteiros matemáticos não têm esse problema. Assim, a noção de um `int` em um computador é realmente apenas uma noção aproximada de um inteiro do mundo real. O mesmo se aplica a um `float`.

A questão é que mesmo os tipos de dados predefinidos fornecidos com as linguagens de programação semelhantes a Java são realmente apenas aproximações ou modelos de conceitos e comportamentos do mundo real. Até aqui, assumimos `int` como algo garantido, mas agora você tem uma nova perspectiva a considerar. Os tipos como `int`, `float`, `char` e outros são exemplos de tipos abstratos de dados. Eles são essencialmente maneiras de representar noções do mundo real em algum nível satisfatório de precisão dentro de um sistema de computador.

O tipo abstrato de dados realmente captura duas noções, a saber, uma *representação dos dados* e as *operações* que são permitidas sobre esses dados. Por exemplo, a noção de `int` define as operações de adição, subtração, multiplicação, divisão e módulo em Java, mas a divisão por zero é indefinida. Outro exemplo é a noção de inteiros negativos cujas operações e representação dos dados são claras, mas a operação de extrair a raiz quadrada de um inteiro negativo é indefinida. Em Java, o programador utiliza classes para implementar tipos abstratos de dados.

Java tem um pequeno conjunto de tipos primitivos. Os ADTs estendem a linguagem de programação básica.

Observação de engenharia de software 8.22



O programador é capaz de criar novos tipos pela utilização do mecanismo de classe. Esses novos tipos podem ser projetados para ser utilizados tão convenientemente quanto os tipos predefinidos. Portanto, Java é uma linguagem extensível. Embora a linguagem seja fácil de estender com esses novos tipos, a linguagem básica em si não é modificável.

Novas classes de Java podem pertencer a um indivíduo, a grupos pequenos, a empresas e assim por diante. Muitas classes são colocadas em *bibliotecas de classe* padrão destinadas a uma ampla distribuição. Isso não produz necessariamente padrões, embora de fato estejam surgindo padrões. O verdadeiro valor de Java será percebido apenas quando as bibliotecas de classe padronizadas e substanciais se tornarem mais amplamente disponíveis do que são hoje. Nos Estados Unidos, essa padronização freqüentemente é promovida pelo *ANSI*, o *American National Standards Institute*. Em nível mundial, a padronização é muitas vezes promovida pela *ISO*, a *International Standards Organization*. Independentemente de como, em última instância, essas bibliotecas aparecem, o leitor que aprender Java e a programação orientada a objetos estará pronto para tirar proveito dos novos tipos de desenvolvimento de software rápido, orientado a componentes, possibilitado pelas bibliotecas de classe.

8.16.1 Exemplo: tipo abstrato de dados fila

Cada um de nós uma vez ou outra precisa entrar em uma fila. A fila de espera também é chamada de *queue*. Esperamos na fila do caixa do supermercado, esperamos na fila para comprar gasolina, esperamos na fila para embarcar em um ônibus, esperamos na fila para pagar o pedágio em uma estrada e todos os estudantes sabem muito bem sobre esperar na fila para se matricular nas disciplinas que querem. Os sistemas de computação utilizam muitas filas de espera internamente, assim escrevemos programas que simulam o que são e o que fazem as filas.

A fila é um bom exemplo de um tipo abstrato de dados. Ela oferece um comportamento bem compreendido para seus clientes. Os clientes colocam coisas na fila uma de cada vez – utilizando a operação *enfileirar* – e os clientes obtêm essas coisas de volta uma por vez, sob demanda – utilizando a operação *desenfileirar*. Conceitualmente, a fila pode tornar-se infinitamente longa. A fila real é, naturalmente, finita. Os itens são retirados da fila na ordem *primeiro a entrar, primeiro a sair (FIFO – first-in, first-out)* – o primeiro item inserido na fila é o primeiro item retirado da fila.

A fila oculta uma representação interna de dados que monitora os itens que estão esperado na fila e oferece um conjunto de operações para seus clientes, a saber, *enfileirar* e *desenfileirar*. Os clientes não estão preocupados com a implementação da fila. Os clientes meramente querem que a fila opere “como anunciado”. Quando um cliente enfileira um novo item, a fila deve aceitar esse item e colocá-lo internamente em alguma estrutura de dados do tipo primeiro a entrar, primeiro a sair. Quando o cliente quer o próximo item do começo da fila, a fila deve retirar o item de sua representação interna e entregar o item para mundo exterior na ordem FIFO (isto é, o item que esteve na fila por mais tempo deve ser o próximo a ser retirado pela próxima operação *desenfileirar*).

O ADT fila garante a integridade de sua estrutura interna de dados. Os clientes não podem manipular diretamente essa estrutura de dados. Somente o ADT fila tem acesso a seus dados internos (isto é, a fila encapsula seus dados). Os clientes podem fazer com que só as operações permitidas sejam realizadas sobre a representação de dados; as operações não-fornecidas na interface pública do ADT são rejeitadas pelo ADT de alguma maneira apropriada. Isso poderia significar emissão de uma mensagem de erro, término da execução ou simplesmente desconhecimento da solicitação de operação.

8.17 (Estudo de caso opcional) Pensando em objetos: começando a programar as classes para a simulação do elevador

Nas Seções “Pensando em objetos” nos Capítulos 1 a 7, apresentamos os fundamentos da orientação a objetos e desenvolvemos um projeto orientado a objetos para nossa simulação de elevador. No Capítulo 8, apresentamos os detalhes da programação com classes de Java. Agora começamos a implementar nosso projeto orientado a objetos em Java. No fim dessa seção, mostramos como gerar código em Java, trabalhando a partir de diagramas de classes. Esse processo se chama *engenharia progressiva (forward engineering)*¹.

Visibilidade

Antes de começarmos a implementar nosso projeto em Java, aplicamos modificadores de acesso a membros (ver Seção 8.2) aos membros de nossas classes. No Capítulo 8, apresentamos os especificadores de acesso **public** e **private** – eles determinam as visibilidades dos atributos e métodos de um objeto para outros objetos. Antes de criarmos arquivos de classes, consideraremos quais atributos e métodos de nossas classes devem ser **public** e quais devem ser **private**.



Observação de engenharia de software 8.23

Cada elemento de uma classe deve ter visibilidade **private**, a menos que possa ser provado que o elemento precisa de visibilidade **public**.

No Capítulo 8, discutimos como os atributos em geral devem ser **private**, mas o que dizer sobre as operações de uma classe – seus métodos? Estas operações são invocadas por clientes daquela classe; portanto, os métodos normalmente devem ser **public**. Na UML, a visibilidade **public** é indicada colocando-se um sinal de mais (+) antes de um elemento particular (isto é, um método ou um atributo); o sinal de menos (-) indica visibilidade **private**. A Fig. 8.22 mostra nosso diagrama de classes atualizado com as notações de visibilidade incluídas.

Implementação: engenharia progressiva

A *engenharia progressiva (forward engineering)* é o processo de transformar um projeto, como aquele em um diagrama de classes, em código em uma linguagem de programação específica, como Java. Agora que já discutimos a programação de classes em Java, fizemos a engenharia progressiva do diagrama de classes da Fig. 8.22 para o cód-

¹ G. Booch, *The Unified Modeling Language User Guide*. Massachussets: Addison Wesley Longman, Inc., 1999:16. [Uma vez que exista o código, o processo de retroceder a partir do código para reproduzir documentos de projeto se chama engenharia reversa (*reverse engineering*)].

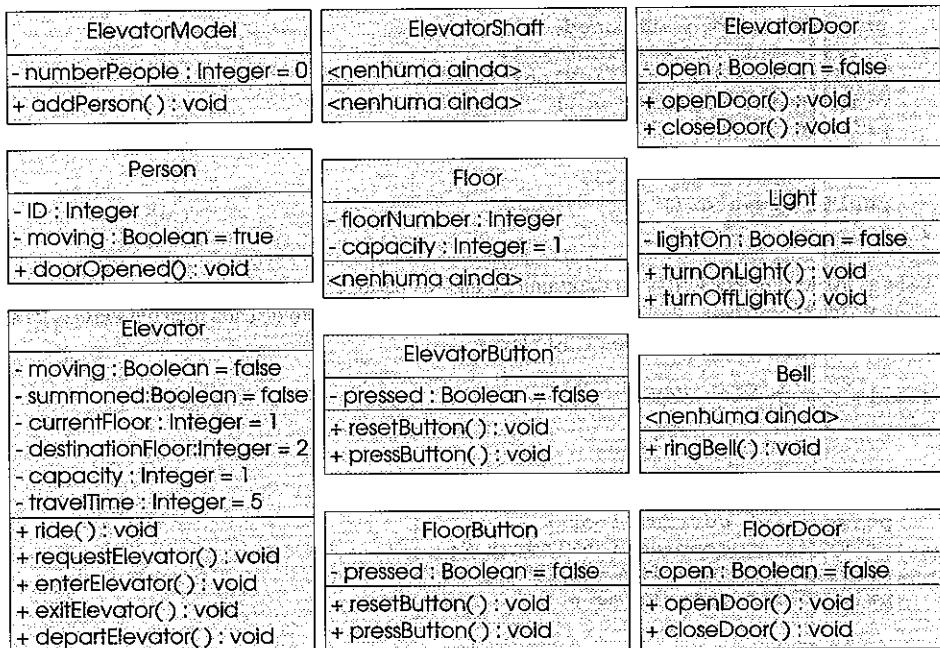


Fig. 8.22 Diagrama de classes completo com notações de visibilidade.

go em Java para nosso simulador de elevador. O código gerado representará somente o “esqueleto”, ou a estrutura, do modelo². Nos Capítulos 9 e 10, modificamos o código para incorporar herança e interfaces, respectivamente. Nos Apêndices G, H e I, apresentamos o código Java completo, que funciona, para nosso modelo.

Como exemplo, fazemos a engenharia progressiva da classe **Elevator** da Fig. 8.22. Usamos esta figura para determinar os atributos e as operações daquela classe. Usamos o diagrama de classes da Fig. 3.23 para determinar associações (e agregações) entre classes. Adotamos as seguintes diretrizes:

1. Usar o nome localizado no primeiro compartimento para declarar a classe como uma classe **public** com um construtor vazio. Por exemplo, a classe **Elevator** dá origem a

```
public class Elevator {
    public Elevator() {}
}
```

2. Usar os atributos localizados no segundo compartimento para declarar as variáveis-membro. Por exemplo, os atributos **private moving**, **summoned**, **currentFloor**, **destinationFloor**, **capacity** e **travelTime** da classe **Elevator** dão origem a

```
public class Elevator {
```

² Até agora, apresentamos aproximadamente a metade do material do estudo de caso – ainda não discutimos herança, tratamento de eventos, *multithreading* e animação. O processo de desenvolvimento padrão recomenda que se termine o processo de projeto antes de começar o processo de codificação. Tecnicamente, não teremos terminado de projetar nosso sistema enquanto não tivermos discutido estes tópicos adicionais, de modo que nossa implementação do código atual poderia parecer prematura. Apresentamos somente uma implementação parcial ilustrando os tópicos abordados no Capítulo 8.

```

    // atributos da classe
    private boolean moving;
    private boolean summoned;
    private int currentFloor = 1;
    private int destinationFloor = 2;
    private int capacity = 1;
    private int travelTime = 5;

    // construtor da classe
    public Elevator() {}
}

```

3. Usar as associações descritas no diagrama de classes para gerar as referências para outros objetos. Por exemplo, de acordo com a Fig. 3.23, **Elevator** contém um objeto de cada uma das classes **ElevatorDoor**, **ElevatorButton** e **Bell**. Isto leva a

```

public class Elevator {

    // atributos da classe
    private boolean moving;
    private boolean summoned;
    private int currentFloor = 1;
    private int destinationFloor = 2;
    private int capacity = 1;
    private int travelTime = 5;

    // objetos da classe
    private ElevatorDoor elevatorDoor;
    private ElevatorButton elevatorButton;
    private Bell bell;

    // construtor da classe
    public Elevator() {}
}

```

4. Usar as operações localizadas no terceiro compartimento da Fig. 8.22 para declarar os métodos. Por exemplo, as operações **public ride**, **requestElevator**, **enterElevator**, **exitElevator** e **departElevator** em **Elevator** dão origem a

```

public class Elevator {

    // atributos da classe
    private boolean moving;
    private boolean summoned;
    private int currentFloor = 1;
    private int destinationFloor = 2;
    private int capacity = 1;
    private int travelTime = 5;

    // objetos da classe
    private ElevatorDoor elevatorDoor;
    private ElevatorButton elevatorButton;
    private Bell bell;

    // construtor da classe
    public Elevator() {}
}

```

```

    // métodos da classe
    public void ride() {}
    public void requestElevator() {}
    public void enterElevator() {}
    public void exitElevator() {}
    public void departElevator() {}
}

```

Isto conclui os fundamentos da engenharia progressiva. Voltaremos a este exemplo nos finais das Seções 9.23 e 10.22 de “Pensando em objetos”, para incorporar herança, interfaces e tratamento de eventos.

Resumo

- A OOP encapsula dados (atributos) e métodos (comportamentos) em objetos; os dados e os métodos de um objeto estão intimamente associados entre si.
- Os objetos têm a propriedade de ocultar informações. Os objetos podem saber como se comunicar entre si através de interfaces bem-definidas, mas normalmente não são autorizados a saber como outros objetos são implementados.
- Os programadores de Java se concentram em criar seus próprios tipos definidos pelo usuário, chamados de classes.
- Os componentes de dados não-**static** de uma classe são chamados de variáveis de instância. Os componentes de dados **static** são chamados de variáveis de classe.
- Java utiliza herança para criar novas classes a partir de definições de classe existentes.
- Cada classe em Java é uma subclasse **Object**. Portanto, cada nova definição de classe tem os atributos (dados) e os comportamentos (métodos) da classe **Object**.
- As palavras-chave **public** e **private** são modificadores de acesso a membros.
- As variáveis de instância e os métodos declarados com o modificador de acesso **public** são acessíveis onde quer que o programa tenha uma referência a um objeto da classe em que eles são definidos.
- As variáveis de instância e os métodos declarados com modificador de acesso **private** são acessíveis somente para métodos da classe em que eles são definidos.
- As variáveis de instância normalmente são declaradas como **private** e os métodos normalmente são declarados como **public**.
- Os métodos **public** (ou serviços **public**) de uma classe são utilizados por clientes da classe para manipular os dados armazenados em objetos da classe.
- O construtor é um método com o mesmo nome da classe, que inicializa as variáveis de instância de um objeto da classe quando o objeto é instanciado. Os métodos construtores podem ser sobrecarregados para uma classe. Os construtores podem aceitar argumentos, mas não podem aceitar um tipo de valor de retorno.
- Os construtores e outros métodos que alteram valores de variável de instância sempre devem manter objetos em um estado consistente.
- O método **toString** não recebe nenhum argumento e retorna um **String**. O método **toString** original da classe **Object** é um marcador de lugar que normalmente é redefinido por uma subclasse.
- Quando um objeto é instanciado, o operador **new** aloca a memória para o objeto, depois **new** chama o construtor da classe para inicializar as variáveis de instância do objeto.
- Se os arquivos **.class** para as classes utilizadas em um programa estão no mesmo diretório que a classe que os utiliza, as instruções **import** não são necessárias.
- Concatenar um **String** e qualquer objeto resulta em uma chamada implícita ao método **toString** do objeto para converter o objeto em um **String**, e então os **Strings** são concatenados.
- Dentro do escopo de uma classe, os membros de classe estão acessíveis para todos os métodos daquela classe e podem ser mencionados simplesmente pelo nome. Fora do escopo da classe, os membros da classe só podem ser acessados por um “handle” (isto é, uma referência a um objeto da classe).
- Se um método define uma variável com o mesmo nome que uma variável com escopo de classe, a variável com escopo da classe é oculta pela variável com escopo de método dentro do método. Uma variável de instância oculta pode ser acessada no método precedendo seu nome com a palavra-chave **this** e o operador ponto.
- Cada classe e interface na Java API pertence a um pacote específico que contém um grupo de classes e interfaces relacionadas.
- Os pacotes são, na verdade, estruturas de diretórios utilizadas para organizar classes e interfaces. Os pacotes fornecem um mecanismo para a reutilização de software e uma convenção para os nomes de classe únicos.

- Criar uma classe reutilizável exige: definir uma classe **public**, adicionar uma instrução **package** ao arquivo de definição da classe, compilar a classe na estrutura de diretórios do pacote apropriada e importar a classe para um programa.
- Ao compilar uma classe em um pacote, a opção **-d** deve ser passada ao compilador para especificar onde criar (ou localizar) todos os diretórios na instrução **package**.
- Os nomes de diretório em **package** tornam-se parte do nome da classe quando a classe é compilada. Utilize esse nome completamente qualificado em programas ou **importe** a classe e utilize seu nome abreviado (o nome da classe sozinho) no programa.
- Se nenhum construtor é definido para uma classe, o compilador cria um construtor **default**.
- Quando um objeto de uma classe tem uma referência para outro objeto da mesma classe, o primeiro objeto pode acessar todos os dados e métodos do segundo objeto.
- As classes freqüentemente fornecem métodos **public** para permitir aos clientes da classe configurar (*set* – isto é, atribuir valores a) ou obter (*get* – isto é, obter os valores de) as variáveis de instância **private**. Os métodos *get* também são comumente chamados de métodos de acesso ou métodos de consulta. Os métodos *set* também são comumente chamados de métodos modificadores (porque em geral alteram um valor).
- Cada evento tem uma origem – o componente GUI com que o usuário interagiu a fim de sinalizar para que o programa faça uma tarefa.
- Utilize a palavra-chave **final** para especificar que uma variável não é modificável e que qualquer tentativa de modificar a variável é um erro. A variável **final** não pode ser modificada por atribuição depois que ela é inicializada. Essa variável deve ser inicializada em sua declaração ou em cada construtor da classe.
- Com composição, a classe tem como membros referências a objetos de outras classes.
- Quando não é especificado nenhum modificador de acesso de membro para um método ou variável no momento em que ele é definido em uma classe, o método ou a variável é considerado como se tivesse acesso de pacote.
- Se um programa utiliza múltiplas classes do mesmo pacote, essas classes podem acessar diretamente os métodos e dados com acesso de pacote umas das outras, através de uma referência para um objeto.
- Cada objeto tem acesso a uma referência a si próprio, chamada de referência **this**, que pode ser utilizada dentro dos métodos da classe para fazer referência aos dados e outros métodos do objeto explicitamente.
- Sempre que você tiver uma referência em um programa (mesmo que seja o resultado de uma chamada a um método), a referência pode ser seguida pelo operador ponto e uma chamada a um dos métodos para o tipo da referência.
- Java realiza coleta de lixo da memória automaticamente. Quando um objeto não é mais utilizado no programa (isto é, não há nenhuma referência ao objeto), o objeto é marcado para coleta de lixo.
- Cada classe em Java pode ter um método finalizador que devolve recursos para o sistema. O método finalizador de uma classe sempre tem o nome **finalize**, não recebe nenhum parâmetro e não retorna nenhum valor. O método **finalize** é definido originalmente na classe **Object** como um marcador de lugar que não faz nada. Isso garante que cada classe tenha um método **finalize** para o coletor de lixo chamar.
- Um variável de classe **static** representa informações que abrangem toda a classe – todos os objetos da classe compartilham a mesma porção de dados. Os membros **public static** de uma classe podem ser acessados através de uma referência a qualquer objeto dessa classe ou podem ser acessados através do nome da classe utilizando o operador ponto.
- O método **public static gc** da classe **System** sugere que o coletor de lixo faça imediatamente uma tentativa de coletar objetos para o lixo. Não há garantia de que coletor de lixo colete objetos em uma ordem específica.
- O método declarado **static** não pode acessar membros de classe **não-static**. Diferentemente dos métodos **não-static**, o método **static** não tem a referência **this** porque as variáveis de classe **static** e os métodos de classe **static** existem independentemente de qualquer objeto de uma classe.
- Os membros de classe **static** existem mesmo quando nenhum objeto dessa classe existe – eles estão disponíveis logo que as classes são carregadas na memória durante a execução.

Terminologia

acesso de pacote

biblioteca de classe

agregação

capacidade de reutilização de software

atributo

chamadas de método

*chamadas de métodos em cadeia
chamadas de métodos em cascata
classe
classe contêiner
cliente de uma classe
código reutilizável
comportamento
composição
construtor
construtor default
construtor sem argumento
controle de acesso a membro
definição de classe
desenvolvimento rápido de aplicações (RAD)
encapsulamento
escopo de classe
estado consistente para uma variável de instância
finalizador
implementação de uma classe
inicializar um objeto de classe
instância de uma classe
instanciar um objeto de uma classe
instrução package
interface para uma classe
interface pública de uma classe
mensagem
método
método auxiliar
método de acesso
método de classe (static)*

*método de consulta
método de instância
método estático
método get
método modificador
método predicado
método set
método utilitário
modificadores de acesso a membro
objeto
ocultamento de informações
opção de compilador -d
operador de acesso a membro (.)
operador new
operador ponto (.)
princípio do menor privilégio
private
programação baseada em objetos (OBP)
programação orientada a objetos (OOP)
public
referência this
serviços de uma classe
tipo abstrato de dados (ADT)
tipo de dados
tipo definido pelo programador
tipo definido pelo usuário
variável de classe
variável de instância
variável estática de classe*

Exercícios de auto-revisão

8.1

Preencha as lacunas em cada uma das frases seguintes:

- Os membros de classe são acessados através do operador _____, junto com uma referência a um objeto da classe.
- Os membros de uma classe especificados como _____ são acessíveis somente para métodos da classe.
- O _____ é um método especial utilizado para inicializar as variáveis de instância de uma classe.
- O método _____ é utilizado para atribuir valores a variáveis de instância **private** de uma classe.
- Os métodos de uma classe normalmente são declarados _____ e as variáveis de instância de uma classe normalmente são declaradas _____.
- O método _____ é utilizado para recuperar valores de dados **private** de uma classe.
- A palavra-chave _____ introduz uma definição de classe.
- Os membros de uma classe especificados como _____ são acessíveis em qualquer lugar em que um objeto da classe esteja em escopo.
- O operador _____ aloca memória dinamicamente para um objeto de um tipo especificado e retorna uma _____ para aquele tipo.
- Uma variável de instância representa informações que abrangem toda a classe.
- A palavra-chave _____ especifica que um objeto ou uma variável não é modificável depois de inicializado.
- O método declarado **static** não pode acessar membros _____ da classe.

Respostas aos exercícios de auto-revisão

- a) ponto (.), b) **private**. c) construtor. d) **set**. e) **public**, **private**. f) **get**. g) **class**. h) **public**. i) **new**, referência. j) **static**. k) **final**. l) não-**static**.

Exercícios

8.2 Crie uma classe chamada **Complex** para realizar aritmética com números complexos. Escreva um programa para testar sua classe.

Os números complexos têm a forma

```
realPart + imaginaryPart * i
```

onde i é

$$\sqrt{-1}$$

Utilize variáveis de ponto flutuante para representar os dados **private** da classe. Forneça um método construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. Forneça um construtor sem argumento com valores *default* caso nenhum inicializador seja fornecido. Forneça métodos **public** para cada um dos itens seguintes:

- Somar dois números **Complex**: as partes reais são somadas de um lado e as partes imaginárias são somadas de outro.
- Subtrair dois números **Complex**: a parte real do operando direito é subtraída da parte real do operando esquerdo e a parte imaginária do operando direito é subtraída da parte imaginária do operando esquerdo.
- Imprimir os números **Complex** na forma (a, b) , onde a é a parte real e b é a parte imaginária.

8.3 Crie uma classe chamada **Rational** para realizar aritmética com frações. Escreva um programa para testar sua classe.

Utilize variáveis do tipo inteiro para representar as variáveis de instância **private** da classe – o **numerator** e o **denominator**. Forneça um método construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. O construtor deve armazenar a fração na forma reduzida (isto é, a fração

$2/4$

seria armazenada no objeto como 1 no **numerator** e 2 no **denominator**). Forneça um construtor sem argumentos com valores *default* caso nenhum inicializador seja fornecido. Forneça métodos **public** para cada um dos itens seguintes:

- Adição de dois números **Rational**. O resultado da adição deve ser armazenado na forma reduzida.
- Subtração de dois números **Rational**. O resultado da subtração deve ser armazenado na forma reduzida.
- Multiplicação de dois números **Rational**. O resultado da multiplicação deve ser armazenado na forma reduzida.
- Divisão de dois números **Rational**. O resultado da divisão deve ser armazenado na forma reduzida.
- Impressão de números **Rational** na forma a/b , onde a é o **numerator** e b é o **denominator**.
- Impressão de números **Rational** em formato de ponto flutuante. (Considere a possibilidade de fornecer facilidades de formatação que permitam que o usuário da classe especifique o número de dígitos de precisão à direita do ponto de fração decimal.)

8.4 Modifique a classe **Time3** da Fig. 8.8 para incluir o método **tick** que incrementa a hora armazenada em um objeto **Time3** em um segundo. Além disso, forneça um método **incrementMinute** para incrementar o minuto e o método **incrementHour** para incrementar a hora. O objeto **Time3** sempre deve permanecer em um estado consistente. Escreva um programa que testa o método **tick**, o método **incrementMinute** e o método **incrementHour**, para assegurar que eles funcionam corretamente. Certifique-se de testar os seguintes casos:

- Incrementar para o próximo minuto.
- Incrementar para a próxima hora.
- Incrementar para o próximo dia (isto é, 11:59:59 PM para 12:00:00 AM).

8.5 Modifique a classe **Date** da Fig. 8.13 para realizar a verificação de erros nos valores inicializadores para as variáveis de instância **month**, **day** e **year** (atualmente, ela valida somente o mês e o dia). Além disso, forneça um método **nextDay** para incrementar o dia por um. O objeto **Date** sempre deve permanecer em um estado consistente. Escreva um programa que testa o método **nextDay** em um laço que imprime a data durante cada iteração do laço para ilustrar que o método **nextDay** funciona corretamente. Certifique-se de testar os seguintes casos:

- Incrementar para o próximo mês.
- Incrementar para o próximo ano.

8.6 Combine a classe **Time3** modificada do Exercício 8.4 e a classe **Date** modificada do Exercício 8.5 em uma classe chamada **DateAndTime**. Modifique o método **tick** para chamar o método **nextDay** se a data for incrementada para o próximo dia. Modifique os métodos **toString** e **toUniversalString()** para dar saída à data além da hora. Escreva um programa para testar a nova classe **DateAndTime**. Teste especificamente o incremento de tempo para o próximo dia.

8.7 Modifique os métodos *set* na classe **Time3** da Fig. 8.8 para retornar valores de erro apropriados se for feita uma tentativa de configurar uma das variáveis de instância **hour**, **minute** ou **second** de um objeto da classe **Time** com um valor inválido. (*Dica:* utilize tipos de retorno **boolean** em cada método.)

8.8 Crie uma classe **Rectangle**. A classe tem atributos **length** e **width**; cada um deles é configurado com o valor *default* 1. Ela tem métodos que calculam o perímetro (**perimeter**) e a área (**area**) do retângulo. Tem métodos *set* e *get* para o comprimento (**length**) e a largura (**width**). Os métodos *set* devem verificar se **length** e **width** são números de ponto flutuante maiores que 0,0 e menores que 20,0. Escreva um programa para testar a classe **Rectangle**.

8.9 Crie uma classe **Rectangle** mais sofisticada que aquela que você criou no Exercício 8.8. Essa classe armazena somente as coordenadas cartesianas dos quatro vértices do retângulo. O construtor chama um método *set* que aceita quatro conjuntos de coordenadas e verifica se cada um deles está no primeiro quadrante e sem coordenadas **x** ou **y** individualmente maiores que 20,0. O método *set* também verifica se as coordenadas fornecidas especificam de fato um retângulo. Forneça métodos para calcular **length**, **width**, **perimeter** e **area**. O comprimento é a maior das duas dimensões. Inclua um método de predicado **isSquare** que determina se o retângulo é um quadrado. Escreva um programa para testar a classe **Rectangle**.

8.10 Modifique a classe **Rectangle** do Exercício 8.9 para incluir um método **draw** que exibe o retângulo dentro de uma caixa de 25 por 25 que inclui a parte do primeiro quadrante em que o retângulo está. Utilize os métodos da classe **Graphics** para ajudar a exibir o **Rectangle**. Se você se sentir motivado, talvez você queira incluir métodos para redimensionar o tamanho do retângulo, girá-lo e movê-lo dentro da parte designada do primeiro quadrante.

8.11 Crie uma classe **HugeInteger** que utiliza um *array* de elementos de 40 dígitos para armazenar inteiros com até 40 dígitos cada um. Forneça os métodos **inputHugeInteger**, **outputHugeInteger**, **addHugeIntegers** e **subtractHugeIntegers**. Para comparar objetos **HugeInteger**, forneça os métodos **isEqual**, **isNotEqual**, **isGreater**, **isLess**, **isGreaterOrEqual** e **isLessOrEqual** – cada um desses é um método de “predicado” que simplesmente retorna **true** se a relação entre os dois **HugeIntegers** for verdadeira e retorna **false** se a relação não for verdadeira. Forneça um método de predicado **isZero**. Se você se sentir motivado, forneça também o método **multiplyHugeIntegers**, o método **divideHugeIntegers** e o método **modulusHugeIntegers**.

8.12 Crie uma classe **TicTacToe** que permita escrever um programa completo para jogar o jogo da velha. A classe contém como dados privados um *array* bidimensional de inteiros 3 por 3. O construtor deve inicializar o tabuleiro vazio com zeros em todas as posições. Permita dois jogadores humanos. Para onde quer que o primeiro jogador se move, coloque um 1 no quadrado especificado; coloque um 2 onde quer que o segundo jogador se move. Todo movimento deve ocorrer para um quadrado vazio. Depois de cada movimento, determine se alguém ganhou o jogo ou se houve empate. Se você se sentir motivado, modifique o programa de modo que o computador faça automaticamente o movimento para um dos jogadores. Além disso, permita que o jogador especifique se quer ser o primeiro ou o segundo a jogar. Se você se sentir excepcionalmente motivado, desenvolva um programa que jogue o jogo da velha tridimensional em um tabuleiro 4 por 4 por 4 [Nota: isso é um projeto desafiador que pode consumir muitas semanas de esforço!]

8.13 Explique a noção de acesso de pacote em Java. Explique os aspectos negativos do acesso de pacote como descrito no texto.

8.14 O que acontece quando um tipo de retorno, mesmo **void**, é especificado para um construtor?

8.15 Crie uma classe **Date** com as seguintes capacidades:

- Gerar como saída a data em múltiplos formatos como

```
MM/DD/YYYY
June 14, 1992
DDD YYYY
```

- Utilizar construtores sobrecarregados para criar objetos **Date** inicializados com datas dos formatos da parte (a).

8.16 Crie a classe **SavingsAccount**. Utilize a variável de classe **static** para armazenar o **annualInterestRate** para todos os possuidores de conta. Cada objeto da classe contém uma variável de instância **private savingsBalance** para indicar a quantidade que o poupador atualmente tem em depósito. Forneça um método **calculateMonthlyInterest** para calcular os juros mensais multiplicando o **savingsBalance** por **annualInterestRate** dividido por 12; esses juros devem ser adicionados ao **savingsBalance**. Forneça um método **static modifyInterestRate** que configure o **annualInterestRate** com um novo valor. Escreva um programa para testar a classe **SavingsAccount**. Instancie dois objetos **savingsAccount**, **saver1** e **saver2**, com saldos de R\$ 2.000,00 e R\$ 3.000,00, respectivamente. Configure **annualInterestRate** como 4%, depois calcule o juro mensal e imprima

os novos saldos para cada um dos poupadões. Depois, configure o `annualInterestRate` como 5% e calcule os juros do próximo mês e imprima os novos saldos para cada um dos poupadões.

8.17 Crie a classe `IntegerSet`. Cada objeto da classe pode armazenar inteiros no intervalo 0 a 100. Um conjunto é representado internamente como um array de `booleans`. O elemento do array `a[i]` é `true` se o inteiro i estiver no conjunto. O elemento do array `a[j]` é `false` se o inteiro j não estiver no conjunto. O construtor sem argumentos inicializa um conjunto para o chamado “conjunto vazio” (isto é, um conjunto cuja representação de array contém todos os valores `false`).

Forneça os seguintes métodos: o método `unionOfIntegerSets` cria um terceiro conjunto que é a união teórica de dois conjuntos existentes (isto é, um elemento do terceiro array do conjunto está configurado como `true` se esse elemento for `true` em qualquer um dos conjuntos existentes ou em ambos; caso contrário, o elemento do terceiro conjunto é configurado como `false`). O método `intersectionOfIntegerSets` cria um terceiro conjunto que é a interseção teórica de dois conjuntos existentes (isto é, um elemento do array do terceiro conjunto é configurado como `false` se esse elemento for `false` em qualquer um ou em ambos os conjuntos existentes; caso contrário, o elemento do terceiro conjunto é configurado como `true`). O método `insertElement` insere um novo inteiro k em um conjunto (configurando `a[k]` como `true`). O método `deleteElement` exclui o inteiro m (configurando `a[m]` como `false`). O método `setPrint` imprime um conjunto como uma lista de números separados por espaços. Imprima somente os elementos que estão presentes no conjunto. Imprima ---- para um conjunto vazio. O método `isEqualTo` determina se dois conjuntos são iguais. Escreva um programa para testar sua classe `IntegerSet`. Instancie vários objetos `IntegerSet`. Teste se todos os seus métodos funcionam adequadamente.

8.18 Seria perfeitamente razoável para a classe `Time1` da Fig. 8.1 representar a hora internamente como o número de segundos desde a meia-noite, em vez dos três valores inteiros `hour`, `minute` e `second`. Os clientes poderiam utilizar os mesmos métodos `public` e obter os mesmos resultados. Modifique a classe `Time1` da Fig. 8.1 para implementar `Time1` como o número de segundos desde a meia-noite e mostre que não há alteração visível para os clientes da classe.

8.19 (*Programa de desenho*) Crie um *applet* de desenho que desenha linhas, retângulos e elipses aleatoriamente. Para esse propósito, crie um conjunto de classes de formas “inteligentes” em que os objetos dessas classes sabem desenhar a si mesmos se providos de um objeto `Graphics` que lhes informa onde desenhar (isto é, o objeto `Graphics` do *applet* permite desenhar uma forma no fundo do *applet*). Os nomes de classe devem ser `MyLine`, `MyRect` e `MyOval`.

Os dados para a classe `MyLine` devem incluir as coordenadas `x1`, `y1`, `x2` e `y2`. O método `drawLine` da classe `Graphics` conectará os dois pontos fornecidos com uma linha. Os dados para as classes `MyRect` e `MyOval` devem incluir um valor da coordenada superior esquerda `x`, um valor da coordenada superior esquerda `y`, uma *largura* (deve ser não-negativa) e uma *altura* (deve ser não-negativa). Todos os dados em cada classe devem ser `private`.

Além dos dados, cada classe deve definir pelo menos os seguintes métodos `public`:

- Um construtor sem argumentos que configura as coordenadas como 0.
- Um construtor com argumentos que configura as coordenadas com os valores fornecidos.
- Métodos `set` para cada parte individual dos dados que permitam que o programador configure independentemente qualquer parte dos dados em uma forma (por exemplo, se você tiver uma variável de instância `x1`, você deve ter um método `setX1`).
- Métodos `get` para cada parte individual dos dados que permitam que o programador recupere independentemente qualquer parte dos dados em uma forma (por exemplo, se você tiver uma variável de instância `x1`, você deve ter um método `getX1`).
- Um método `draw` com a primeira linha

```
public void draw( Graphics g )
```

será chamado a partir do método `paint` do *applet* para desenhar uma forma sobre a tela.

Os métodos precedentes são obrigatórios. Se você quiser fornecer mais métodos para maior flexibilidade, faça isto.

Comece definindo a classe `MyLine` e um *applet* para testar suas classes. O *applet* deve ter uma variável de instância `MyLine line` que possa fazer referência ao objeto `MyLine` (criado no método `init` do *applet* com coordenadas aleatórias). O método `paint` do *applet* deve desenhar a forma com uma instrução como

```
line.draw( g );
```

onde `line` é a referência `MyLine` e `g` é o objeto `Graphics` que a forma utilizará para desenhar a si próprio no *applet*.

Em seguida, altere a única referência `MyLine` para transformá-la em um array das referências `MyLine` e incorpore vários objetos `MyLine` no programa para desenhar. O método `paint` do *applet* deve percorrer o array de objetos `MyLine` e desenhar todos eles.

Depois que a parte precedente estiver funcionando, você deve definir as classes `MyOval` e `MyRect` e adicionar objetos dessas classes aos arrays `MyRect` e `MyOval1`. O método `paint` do *applet* deve percorrer cada array e desenhar cada forma. Crie cinco formas de cada tipo.

Quando o *applet* estiver sendo executado, selecione **Reload** a partir do menu **Applet** do `appletviewer` para re-carregar o *applet*. Isso fará com que o *applet* escolha novos números aleatórios para as formas e desenhe as formas novamente.

No Capítulo 9, modificaremos esse exercício para tirar proveito das semelhanças entre as classes e não reinventarmos a roda.

9

Programação orientada a objetos

Objetivos

- Entender herança e reutilização de *software*.
- Entender superclasses e subclasses.
- Mostrar como é possível ampliar e manter o polimorfismo.
- Entender a distinção entre classes abstratas e classes concretas.
- Aprender a criar classes e interfaces abstratas.

Nunca diga que você conhece inteiramente uma pessoa até dividir uma herança com ela.

Johann Kaspar Lavater

Esse método é para definir como número de uma classe a classe de todas as classes similares à classe fornecida.

Bertrand Russell

Melhor que herdar uma biblioteca é colecionar uma.

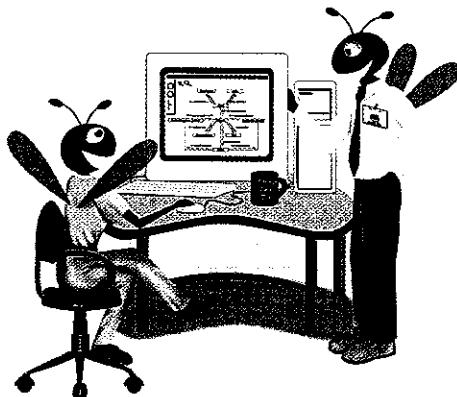
Augustine Birrell

Propostas genéricas não decidem casos concretos.

Oliver Wendell Holmes

*Um filósofo de estatura imponente não pensa em um vazio.
Mesmo suas idéias mais abstratas são, em alguma medida,
condicionadas pelo que é ou não conhecido na época em
que ele vive.*

Alfred North Whitehead



Sumário do capítulo

- 9.1 Introdução
- 9.2 Superclasses e subclasses
- 9.3 Membros `protected`
- 9.4 Relacionamento entre objetos de superclasse e objetos de subclasse
- 9.5 Construtores e finalizadores em subclasses
- 9.6 Conversão implícita de objeto de subclasse para objeto de superclasse
- 9.7 Engenharia de software com herança
- 9.8 Composição versus herança
- 9.9 Estudo de caso: ponto, círculo, cilindro
- 9.10 Introdução ao polimorfismo
- 9.11 Campos de tipo e instruções `switch`
- 9.12 Vinculação dinâmica de método
- 9.13 Métodos e classes final
- 9.14 Superclasses abstratas e classes concretas
- 9.15 Exemplos de polimorfismo
- 9.16 Estudo de caso: um sistema de folha de pagamento utilizando polimorfismo
- 9.17 Novas classes e vinculação dinâmica
- 9.18 Estudo de caso: herança de interface e implementação
- 9.19 Estudo de caso: criando e utilizando interfaces
- 9.20 Definições de classe interna
- 9.21 Notas sobre definições de classe interna
- 9.22 Classes envolvidas de tipo para tipos primitivos
- 9.23 (Estudo de caso opcional) Pensando em objetos: incorporando herança à simulação do elevador
- 9.24 (Opcional) Descobrindo padrões de projeto: apresentando os padrões de criação, estruturais e comportamentais de projeto

*Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão
Exercícios*

9.1 Introdução

Neste capítulo discutimos a *programação orientada a objetos* (*OOP – object-oriented programming*) e suas principais tecnologias – *herança* e *polimorfismo*. A herança é uma forma de reutilização de *software* em que novas classes são criadas a partir das classes existentes, absorvendo seus atributos e comportamentos e adicionando novos recursos que as novas classes exigem. A herança tira proveito dos relacionamentos entre classes, nos quais os objetos de uma certa classe – como uma classe de veículos – têm as mesmas características. As classes de objetos recém-criadas derivam-se ao absorver as características das classes existentes e ao adicionar suas próprias características. Um objeto da classe “conversível” certamente tem as características da classe mais genérica “automóvel”, mas o teatro de um conversível sobe e baixa.

A reutilização de *software* economiza tempo no desenvolvimento dos programas. Ela incentiva a reutilização de *software* de alta qualidade testados e depurados, reduzindo os problemas depois que um sistema se torna operacional. Essas são possibilidades animadoras. O polimorfismo permite-nos escrever programas de uma forma geral para tratar uma ampla variedade de classes relacionadas existentes e ainda a serem especificadas. O polimorfismo

torna fácil adicionar novos recursos a um sistema. A herança e o polimorfismo são técnicas eficazes para lidar com a complexidade dos softwares.

Ao criar uma nova classe, em vez de escrever completamente novas variáveis de instância e métodos de instância, o programador pode determinar que a nova classe deve *herdar* as variáveis e os métodos de instância de uma *superclasse* definida previamente. A nova classe é conhecida como *subclasse*. Cada subclasse se torna uma candidata a ser uma superclasse para alguma subclasse futura.

A *superclasse direta* de uma subclasse é a superclasse da qual a subclasse herda explicitamente (através da palavra-chave `extends`). Uma superclasse indireta é herdada de dois ou mais níveis acima na hierarquia da classe. Por exemplo, a classe `JApplet` (pacote `javax.swing`) estende a classe `Applet` (pacote `java.applet`). Assim, cada classe *applet* que definimos é uma subclasse direta de `JApplet` e uma subclasse indireta de `Applet`.

Com a *herança simples*, uma classe se deriva de uma superclasse. Java não suporta *herança múltipla* (como C++ suporta) mas suporta a noção de *interfaces*. As interfaces ajudam Java a alcançar muitas das vantagens da herança múltipla sem os problemas associados. Discutiremos os detalhes das interfaces neste capítulo. Analisamos tanto os princípios gerais como um exemplo específico detalhado da criação e utilização de interfaces.

A subclasse normalmente adiciona suas próprias variáveis de instância e seus próprios métodos de instância; portanto, em geral, uma subclasse é maior que sua superclasse. A subclasse é mais específica que sua superclasse e representa um grupo menor e mais especializado de objetos. Com herança simples, no início, a subclasse é essencialmente idêntica à superclasse. A força real da herança surge da capacidade de definir na subclasse adições, ou substituições, aos recursos herdados da superclasse.

Cada objeto de uma subclasse também é um objeto da superclasse daquela classe. Por exemplo, todos os *applets* que definimos são considerados objetos da classe `JApplet`. Além disso, como `JApplet` estende `Applet`, todos os *applets* que definimos são considerados um `Applet`. Esta informação é fundamental quando se desenvolve *applets*, porque um contêiner de *applets* pode executar um programa somente se ele for um `Applet`. Embora um objeto de subclasse sempre possa ser tratado como um dos tipos de sua superclasse, os objetos da superclasse não são considerados objetos dos tipos de suas subclasses. Tiraremos proveito desse relacionamento de “um objeto da subclasse é um objeto da superclasse” para realizar algumas manipulações poderosas. Por exemplo, um aplicativo de desenho pode manter uma lista de formas a serem exibidas. Se todas as formas estendem a mesma superclasse, direta ou indiretamente, o programa de desenho pode armazenar todas as formas em um *array* (ou outra estrutura de dados) de objetos da superclasse. Como veremos neste capítulo, essa capacidade de processar um conjunto de objetos como um único tipo é um dos princípios fundamentais da programação orientada a objetos.

Adicionamos uma nova forma de controle de acesso de membros neste capítulo, a saber: o acesso `protected`. Os métodos de subclasse e os métodos de outras classes no mesmo pacote que a superclasse podem acessar os membros `protected` da superclasse.

A experiência na construção de sistemas de *software* indica que partes significativas do código lidam com casos especiais intimamente relacionados. Torna-se difícil nesses sistemas ter uma “visão geral”, porque o projetista e o programador ficam preocupados com os casos especiais. A programação orientada a objetos fornece várias maneiras de “ver a floresta através das árvores”. O programador e projetista se concentra na visão geral – o que há de comum entre os objetos no sistema – e não nos casos especiais. Este processo se chama *abstração*.

Se um programa procedural tem muitos casos especiais intimamente relacionados, então é comum ver estruturas `switch` ou estruturas `if/else` aninhadas que distinguem entre os casos especiais e fornecem a lógica do processamento para lidar com cada caso individualmente. Mostraremos como utilizar herança e polimorfismo para substituir essa lógica `switch` por uma lógica muito mais simples.

Distinguimos entre o *relacionamento “é um”* e o *relacionamento “tem um”*. “É um” é a herança. Em um relacionamento do tipo “é um”, o objeto de um tipo de subclasse também pode ser tratado como objeto de seu tipo de superclasse. “Tem um” é a composição (como discutimos no Capítulo 8). Em um relacionamento “tem um”, o objeto de classe tem um ou mais objetos de outras classes como membros. Por exemplo, o carro *tem uma* direção.

Os métodos de uma subclasse podem precisar acessar certas variáveis de instância e métodos de sua superclasse. Um aspecto crucial da engenharia de *software* em Java é que uma subclasse não pode acessar diretamente os membros `private` de sua superclasse. Se uma subclasse pudesse acessar os membros `private` da superclasse, isso violaria o ocultamento de informações na superclasse.

Observação de engenharia de software 9.1

*Uma subclasse não pode acessar diretamente os membros **private** de sua superclasse.*

Dica de teste e depuração 9.1

*Ocultar os membros **private** é uma ajuda enorme no teste, na depuração e na modificação correta de sistemas. Se uma subclasse pudesse acessar os membros **private** de sua superclasse, seria possível para as classes derivadas dessa subclasse acessar esses dados também, e assim por diante. Isso propagaria o acesso ao que se supõe serem dados **private**, e os benefícios do ocultamento de informações seriam perdidos por toda a hierarquia de classe.*

A subclasse pode, entretanto, acessar os membros **public** e **protected** de sua superclasse. A subclasse também pode usar os membros com acesso de pacote de sua superclasse se a subclasse e a superclasse estiverem no mesmo pacote. Os membros de superclasse que não devem ser acessíveis para uma subclasse através de herança são declarados **private** na superclasse. A subclasse pode executar alterações de estado em membros **private** de superclasse somente através de métodos **public**, **protected** e com acesso de pacote fornecidos na superclasse e herdados pela subclasse. [Nota: usamos variáveis de instância **protected** neste capítulo para demonstrar como elas funcionam. Diversos exercícios neste capítulo pedem que você use somente variáveis de instância privadas, para manter o encapsulamento.]

Observação de engenharia de software 9.2

*Para preservar o encapsulamento, todas as variáveis de instância devem ser declaradas **private** e devem ser acessíveis somente através de métodos set e get da classe.*

Um problema da herança é que uma subclasse pode herdar métodos de que ela não necessita ou que não deveria ter. É responsabilidade do projetista da classe assegurar que os recursos fornecidos por uma classe são apropriados para subclasses futuras. Mesmo quando os métodos da superclasse são apropriados para as subclasses, é comum que uma subclasse exija que o método execute uma tarefa de uma maneira que é específica para a subclasse. Em tais casos, o método da superclasse pode ser *sobrescrito* (redefinido) na subclasse por uma implementação apropriada.

Talvez a mais animadora seja a noção de que as novas classes podem herdar *bibliotecas de classes* abundantes, como aquelas oferecidas com a Java API. As organizações desenvolvem suas próprias bibliotecas de classes e podem tirar proveito de outras bibliotecas disponíveis no mundo. Algum dia, a maior parte do *software* poderá ser construída de *componentes reutilizáveis padronizados*, exatamente como se constrói o *hardware*. Isso ajudará a superar os desafios de se desenvolver os *softwares* cada vez mais poderosos que precisaremos no futuro.

9.2 Superclasses e subclasses

Freqüentemente o objeto de uma classe também “é um” objeto de outra classe. O retângulo certamente é *um* quadrilátero (assim como o são os quadrados, os paralelogramos e os trapézios). Portanto, pode-se dizer que a classe **Retangulo** herda da classe **Quadrilatero**. Nesse contexto, a classe **Quadrilatero** é uma superclasse e a classe **Retangulo** é uma subclasse. O retângulo é *um* tipo específico de quadrilátero, mas é incorreto afirmar que o quadrilátero é *um* retângulo (o quadrilátero pode ser um paralelogramo). A Fig. 9.1 mostra vários exemplos simples de herança de superclasses e subclasses em potencial.

A herança normalmente produz subclasses com *mais* recursos que suas superclasses, de modo que os termos *superclasse* e *subclasse* podem gerar confusão. Entretanto, há uma outra maneira de encarar esses termos que oferece um sentido perfeitamente claro. Como cada objeto de subclasse “é um” objeto de sua superclasse e, como uma superclasse pode ter muitas subclasses, o conjunto de objetos representado por uma superclasse é normalmente maior que o conjunto de objetos representados por qualquer uma das subclasses dessa superclasse. Por exemplo, a superclasse **Veiculo** representa de forma genérica todos os veículos, como carros, caminhões, barcos, bicicletas, e assim por diante. Entretanto, a subclasse **Carro** representa apenas um pequeno subconjunto de todos os **Veiculos** no mundo.

Os relacionamentos de herança formam estruturas hierárquicas do tipo árvore. A superclasse existe em um relacionamento hierárquico com suas subclasses. A classe certamente pode existir sozinha, mas é quando uma classe é utilizada com o mecanismo de herança que ela se torna uma superclasse que fornece atributos e comportamentos para outras classes ou uma classe que herda esses atributos e comportamentos. Freqüentemente, a classe é tanto uma subclass como uma superclasse.

Superclasse	Subclasses
Aluno	AlunoDeGraduação AlunoDePósGraduação
Forma	Círculo Triângulo Retângulo
Financiamento	FinanciamentoDoCarro FinanciamentoDaReformaDaCasa FinanciamentoDaCasa
Empregado	CorpoDocente Funcionário
Conta	ContaCorrente ContaDePoupança

Fig. 9.1 Alguns exemplos simples de herança nos quais a subclass “é uma” superclasse.

Vamos desenvolver uma hierarquia de herança simples. Uma comunidade universitária típica tem milhares de pessoas que são membros da comunidade. Essas pessoas são empregados, alunos e graduados. Entre os empregados incluem-se o corpo docente e os funcionários. Os docentes são tanto administradores (como diretores e chefes de departamento) como professores. Isso produz a hierarquia de herança mostrada na Fig. 9.2. Observe que a hierarquia de herança pode conter muitas outras classes. Por exemplo, os alunos podem ser alunos de pós-graduação ou alunos de graduação. Os alunos de graduação podem ser primeiranistas, segundanistas, terceiranistas e quartanistas. E assim por diante. As setas na hierarquia representam o relacionamento “é um”. Por exemplo, baseado nessa hierarquia de classe podemos afirmar, “o **Empregado** é um **MembroDaComunidade**” ou “o **Professor** é um membro do **CorpoDocente**”. **MembroDaComunidade** é a *superclasse direta* de **Empregado**, **Aluno** e **Graduado**. **MembroDaComunidade** é uma *superclasse indireta* de todas as outras classes no diagrama da hierarquia. Observe que a classe **Empregado** é tanto uma subclass de **MembroDaComunidade** quanto uma superclasse de **Professor** e **Funcionario**.

Além disso, iniciando da parte inferior do diagrama, você pode seguir as setas e aplicar o relacionamento *é um* progressivamente até a superclasse de mais alto nível da hierarquia. Por exemplo, o **Administrador** é um membro do **CorpoDocente**, é um **Empregado** e é um **MembroDaComunidade**. E, em Java, o **Administrador** também é um **Object**, porque todas as classes em Java têm **Object** como uma de suas superclasses diretas ou indiretas. Portanto, todas as classes em Java são relacionadas em um relacionamento hierárquico em que compartilham os 11 métodos definidos pela classe **Object**, que inclui os métodos **toString** e **finalize** discutidos anteriormente. Outros métodos da classe **Object** serão discutidos quando forem necessários no texto.

Outra hierarquia de herança substancial é a hierarquia **Forma** da Fig. 9.3. Há exemplos abundantes de hierarquias no mundo real, mas os alunos não estão acostumados a categorizar o mundo real dessa maneira; assim, é necessária alguma adaptação do pensamento. De fato, os alunos de biologia têm mais prática com hierarquias. Tudo que estudamos em biologia é agrupado em uma hierarquia encabeçada por coisas vivas e elas podem ser plantas ou animais e assim por diante.

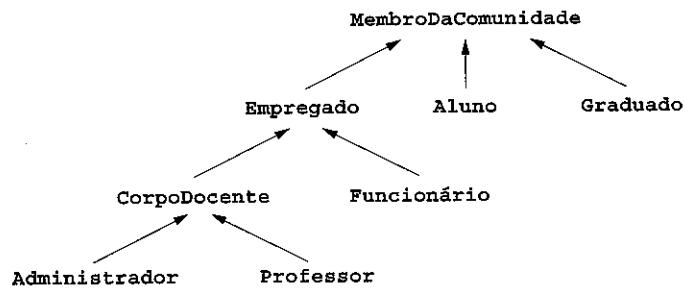


Fig. 9.2 Hierarquia de herança para **MembroDaComunidade** em uma universidade.

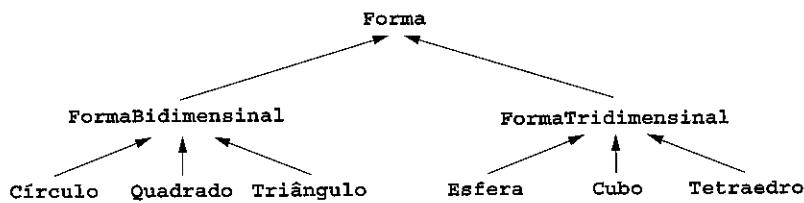


Fig. 9.3 Parte de uma hierarquia de classes **Forma**.

Para especificar que a classe **FormaBidimensional** deriva-se (ou herda) da classe **Forma**, a classe **FormaBidimensional** pode ser definida em Java como segue:

```

public class FormaBidimensional extends Forma{
    ...
}
  
```

Com a herança, os membros **private** de uma superclasse não são diretamente acessíveis a partir das subclasses dessa classe. Os membros com acesso de pacote da superclasse são acessíveis em uma subclass apenas se a superclasse e sua subclass estiverem no mesmo pacote. Todos os outros membros da superclasse tornam-se membros da subclass, utilizando seu acesso original de membro (isto é, os membros **public** da superclasse tornam-se os membros **public** da subclass e os membros **protected** da superclasse tornam-se os membros **protected** da subclass).



Observação de engenharia de software 9.3

Os construtores nunca são herdados – eles são específicos para a classe em que são definidos.

É possível tratar objetos de superclasse e objetos de subclass de maneira semelhante; essa propriedade em comum é expressa nos atributos e comportamentos da superclasse. Os objetos de todas as classes derivadas de uma superclasse comum podem ser tratados como objetos dessa superclasse.

Consideraremos muitos exemplos em que podemos tirar proveito desse relacionamento de herança com uma facilidade de programação não-disponível em linguagens não-orientadas a objetos, como C.

9.3 Membros `protected`

Os membros `public` de uma superclasse são acessados de qualquer lugar em que o programa tenha uma referência para o tipo dessa superclasse ou para um dos tipos de suas subclasses. Os membros `private` de uma superclasse são acessados apenas em métodos dessa superclasse.

Os membros de acesso `protected` de uma superclasse servem como nível intermediário de proteção entre o acesso `public` e `private`. Os membros `protected` de uma superclasse podem ser acessados apenas por métodos da superclasse, por métodos de subclasses e por métodos de outras classes no mesmo pacote (os membros `protected` têm acesso de pacote).

Os métodos de subclasses normalmente podem fazer referência aos membros `public` e `protected` da superclasse simplesmente com os nomes dos membros. Quando um método de subclass sobrescreve um método de superclasse, o método de superclasse pode ser acessado a partir da subclass que precede o nome do método de superclasse com a palavra-chave `super` seguida pelo operador ponto (`.`). Essa técnica é ilustrada várias vezes ao longo do capítulo.

9.4 Relacionamento entre objetos de superclasse e objetos de subclass

O objeto de uma subclass pode ser tratado como um objeto de sua superclasse. Isso possibilita algumas manipulações interessantes. Por exemplo, apesar de os objetos de uma variedade de classes derivadas de uma superclasse em particular poderem ser bem diferentes uns dos outros, podemos criar um *array* de referências para eles – contanto que os tratemos como objetos de superclasse. Mas o contrário não é verdadeiro: o objeto de superclasse não pode sempre ser tratado como um objeto de subclass. Por exemplo, uma `Forma` não é sempre um `Círculo`.

Entretanto, pode-se utilizar uma coerção explícita para converter uma referência de superclasse em uma referência de subclass. Isso pode ser feito apenas quando a referência de superclasse estiver fazendo referência a um objeto de subclass; caso contrário, Java indicará uma `ClassCastException` – uma indicação de que a operação de coerção não é permitida. As exceções são discutidas em detalhes no Capítulo 14.



Erro comum de programação 9.1

Atribuir um objeto de uma superclasse a uma referência de subclass (sem coerção) é um erro de sintaxe.



Observação de engenharia de software 9.4

Se um objeto foi atribuído a uma referência de uma de suas superclasses, é aceitável fazer coerção desse objeto de volta para seu próprio tipo. Na verdade, isso deve ser feito a fim de enviar para esse objeto as mensagens que não apareçam naquela superclasse.

Nosso primeiro exemplo consiste em duas classes. A Fig. 9.4 mostra uma definição de classe `Point`. A Fig. 9.5 mostra uma definição de classe `Circle`. Veremos que a classe `Circle` herda da classe `Point`. A Fig. 9.6 mostra a classe de aplicativo `InheritanceTest`, que demonstra a atribuição de referências para subclass e referências para superclasse, e coerção de referências para superclasses a referências para subclass.

Cada *applet* que definimos anteriormente usou alguma das técnicas apresentadas aqui. Estamos agora formalizando o conceito de herança. No Capítulo 3, afirmamos que cada definição de classe em Java deve estender outra classe. Entretanto, observe na Fig. 9.4 que a classe `Point` (linha 4) não utiliza explicitamente a palavra-chave `extends`. Se uma nova definição de classe não estende explicitamente uma definição de classe existente, Java utiliza implicitamente a classe `Object` (pacote `java.lang`) como a superclasse para a nova definição de classe. A classe `Object` fornece um conjunto de métodos que pode ser utilizado com qualquer objeto de qualquer classe.



Observação de engenharia de software 9.5

Cada classe em Java estende `Object` implicitamente, a menos que seja especificado o contrário na primeira linha da definição de classe, caso em que a classe estende `Object` indiretamente. Portanto, a classe `Object` é a superclasse de toda a hierarquia de classes de Java.

Examinemos primeiro a definição de classe `Point` (Fig. 9.4). Os serviços `public` da classe `Point` incluem os métodos `setPoint`, `getX`, `getY`, `toString` e dois construtores `Point`. As variáveis de instância `x` e `y` de `Point` são especificadas como `protected`. Isso impede que os clientes de objetos `Point` acessem diretamente

os dados (a menos que sejam classes nos mesmos pacotes), mas permite que as classes derivadas de **Point** acessem as variáveis de instância herdadas diretamente. Se os dados foram especificados como **private**, os métodos não-**private** de **Point** teriam de ser utilizados para acessar os dados, mesmo por subclasses. Observe que o método **toString** da classe **Point** sobrescreve o método **toString** original herdado da classe **Object**.

```

1 // Fig. 9.4: Point.java
2 // Definição da classe Point
3
4 public class Point {
5     protected int x, y;    // coordenadas de Point
6
7     // Construtor sem argumentos
8     public Point()
9     {
10         // chamada implícita do construtor da superclasse ocorre aqui
11         setPoint( 0, 0 );
12     }
13
14     // construtor
15     public Point( int xCoordinate, int yCoordinate )
16     {
17         // chamada implícita do construtor da superclasse ocorre aqui
18         setPoint( xCoordinate, yCoordinate );
19     }
20
21     // configura as coordenadas x e y de Point
22     public void setPoint( int xCoordinate, int yCoordinate )
23     {
24         x = xCoordinate;
25         y = yCoordinate;
26     }
27
28     // obtém a coordenada x
29     public int getX()
30     {
31         return x;
32     }
33
34     // obtém a coordenada y
35     public int getY()
36     {
37         return y;
38     }
39
40     // converte para uma representação de String
41     public String toString()
42     {
43         return "[" + x + ", " + y + "]";
44     }
45
46 } // fim da classe Point

```

Fig. 9.4 Definição da classe **Point**.

Os construtores da classe **Point** (linhas 8 a 12 e 15 a 19) devem chamar o construtor da classe **Object**. Na verdade, cada construtor de subclass deve chamar o construtor da sua superclasse direta como sua primeira tarefa, implícita ou explicitamente (a sintaxe para essa chamada é discutida com a classe **Circle** em seguida). Se não houver chamada explícita ao construtor da superclasse, Java automaticamente tenta chamar o construtor *default* da superclasse. Observe que as linhas 10 e 17 são comentários que indicam onde ocorre a chamada ao construtor *default* da superclasse **Object**.



Observação de engenharia de software 9.6

Todos os construtores de subclasses devem chamar um dos construtores da superclasse direta explícita ou implicitamente. Podem-se fazer chamadas implícitas apenas para o construtor sem argumentos da superclasse. Se a superclasse não fornece um construtor sem argumentos, todas as subclasses diretas daquela classe devem chamar um dos construtores da superclasse explicitamente.

A classe **Circle** (Fig. 9.5) herda da classe **Point** como especificado com a palavra-chave **extends** na linha 4. A palavra-chave **extends** na definição de classe indica herança. Todos os membros (não-**private**) da classe **Point** (exceto os construtores) são herdados na classe **Circle**. Portanto, a interface **public** para **Circle** inclui os métodos **public** da classe **Point**, bem como os dois construtores sobrecarregados **Circle** e os métodos **setRadius**, **getRadius**, **area** e **toString** de **Circle**. Repare que o método **area** (linhas 38 a 41) utiliza a constante predefinida **Math.PI** da classe **Math** (pacote **java.lang**) para calcular a área de um círculo.

```

1 // Fig. 9.5: Circle.java
2 // Definição da classe Circle
3
4 public class Circle extends Point { // herda de Point
5     protected double radius;
6
7     // construtor sem argumentos
8     public Circle()
9     {
10         // chamada implícita para o construtor da superclasse ocorre aqui
11         setRadius( 0 );
12     }
13
14     // construtor
15     public Circle( double circleRadius, int xCoordinate,
16                     int yCoordinate )
17     {
18         // chama o construtor da superclasse para configurar coordenadas
19         super( xCoordinate, yCoordinate );
20
21         // configura o raio
22         setRadius( circleRadius );
23     }
24
25     // configura o raio do Circle
26     public void setRadius( double circleRadius )
27     {
28         radius = ( circleRadius >= 0.0 ? circleRadius : 0.0 );
29     }
30
31     // obtém o raio do Circle
32     public double getRadius()
33     {
34         return radius;
35     }
36
37     // calcula a área do Circle
38     public double area()
39     {
40         return Math.PI * radius * radius;
41     }
42
43     // converte o Circle para um String

```

Fig. 9.5 Definição da classe **Circle** (parte 1 de 2).

```

44     public String toString()
45     {
46         return "Center = " + "[" + x + ", " + y + "] " +
47             "; Radius = " + radius;
48     }
49
50 } // fim da classe Circle

```

Fig. 9.5 Definição da classe **Circle** (parte 2 de 2).

Os construtores **Circle** (linhas 8 a 12 e 15 a 23) devem invocar um construtor **Point** para inicializar a parte da superclasse de um objeto **Circle** (isto é, as variáveis **x** e **y** herdadas de **Point**). O construtor *default* nas linhas 8 a 12 não chama explicitamente um construtor **Point**, então Java automaticamente chama o construtor *default* da classe **Point** (definido na linha 8 da Fig. 9.4), que inicializa os membros **x** e **y** da superclasse com zeros. Se a classe **Point** contivesse apenas o construtor com argumentos (isto é, não houvesse fornecido um construtor *default*), ocorreria um erro de compilação.

A linha 19 no corpo do segundo construtor **Circle** invoca explicitamente o construtor **Point** (definido na linha 15 da Fig. 9.4) utilizando a *sintaxe de chamada de construtor de superclasse* – a palavra-chave **super** seguida por um conjunto de parênteses que contém os argumentos para o construtor da superclasse. Nesse caso, os argumentos são os valores **xCoordinate** e **yCoordinate** que são usados pelo construtor **Point** para inicializar os membros da superclasse **x** e **y**. A chamada para o construtor da superclasse deve ser a primeira linha no corpo do construtor da subclasse. Para chamar explicitamente o construtor *default* da superclasse utilize a instrução

super(); // chamada explícita ao construtor default da superclasse



Erro comum de programação 9.2

Ocorrerá um erro de sintaxe se uma chamada **super** por uma subclasse para seu construtor de superclasse não for a primeira instrução no construtor de subclasse.



Erro comum de programação 9.3

Ocorrerá um erro de sintaxe se os argumentos para uma chamada **super** por uma subclasse para seu construtor de superclasse não corresponderem aos parâmetros especificados em uma das definições do construtor de superclasse.

A subclasse pode redefinir um método de superclasse com a mesma assinatura; esse processo se chama *sobreescriver (override)* um método de superclasse. Quando esse método é mencionado por nome na subclasse, a versão de subclasse é chamada automaticamente. Na verdade, em todos os *applets* deste livro estivemos sobreescrivendo métodos. Quando estendemos **JApplet** para criar uma nova classe de *applet*, a nova classe herda versões de **init** e **paint** (e muitos outros métodos). Todas as vezes que definimos **init** ou **paint**, estávamos sobreescrivendo a versão original que foi herdada. Além disso, quando fornecemos o método **toString** para muitas das classes no Capítulo 8, estávamos sobreescrivendo a versão original de **toString** fornecida pela classe **Object**. Como vemos na Fig. 9.8, pode-se usar a referência **super**, seguida pelo operador ponto, para acessar a versão original desse método de superclasse a partir da subclasse.

Observe que o método **toString** da classe **Circle** (linhas 44 a 48) sobreescrava o método **toString** da classe **Point** (linhas 41 a 44 da Fig. 9.4). O método **toString** da classe **Point** sobreescrava o método original **toString** fornecido pela classe **Object**. Na verdade, todas as classes herdam um método **toString**, porque a classe **Object** fornece o método original **toString**. Esse método converte um objeto de qualquer classe em uma representação de **String** e, às vezes, é chamado implicitamente pelo programa (por exemplo, quando um objeto é concatenado com um **String**). O método **toString** de **Circle** acessa diretamente as variáveis de instância **protected x** e **y** que foram herdadas da classe **Point**. O método **toString** usa os valores de **x** e **y** como parte da representação do **Circle** como **String**. Na verdade, se você estudar o método **toString** da classe **Point** e o método **toString** da classe **Circle**, perceberá que o **toString** de **Circle** utiliza a mesma formatação que o **toString** de **Point** para as partes **Point** de **Circle**. Além disso, lembre-se de nossa *Observação de engenharia de software 8.14*, indicando que, se existe um método que realiza parte da tarefa de outro método, é necessário chamar o método. O **toString** de **Point** realiza parte da tarefa do **toString** de **Circle**. Para chamar o **toString** de **Point** a partir da classe **Circle**, utilize a expressão

```
super.toString()
```

Observação de engenharia de software 9.7



A redefinição de um método de superclasse em uma subclasse não precisa ter a mesma assinatura que o método da superclasse. Essa redefinição não é uma sobrescrita de método, mas, simplesmente, um exemplo de sobrecarga de método.

Observação de engenharia de software 9.8



Qualquer objeto pode ser convertido em um **String** com uma chamada explícita ou implícita para o método **toString** do objeto.

Observação de engenharia de software 9.9



Cada classe deve sobreescrivêr o método **toString** para retornar informações úteis sobre os objetos dessa classe.

Erro comum de programação 9.4



Ocorrerá um erro de sintaxe se o método em uma superclasse e um método em sua subclasse tiver a mesma assinatura mas um tipo diferente de retorno.

O aplicativo **InheritanceTest** (Fig. 9.6) instancia o objeto **Point point1** e o objeto **Circle circle1** nas linhas 18 e 19 em **main**. As representações como **String** de cada objeto são atribuídas ao **String output** para mostrar que eles foram inicializadas corretamente (linhas 21 e 22). Veja as primeiras duas linhas da saída na captura de tela para confirmar isso.

```

1 // Fig. 9.6: InheritanceTest.java
2 // Demonstrando o relacionamento "é um"
3
4 // Pacotes do núcleo de Java
5 import java.text.DecimalFormat;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JOptionPane;
9
10 public class InheritanceTest {
11
12     // testa as classes Point e Circle
13     public static void main( String args[] )
14     {
15         Point point1, point2;
16         Circle circle1, circle2;
17
18         point1 = new Point( 30, 50 );
19         circle1 = new Circle( 2.7, 120, 89 );
20
21         String output = "Point point1: " + point1.toString() +
22             "\nCircle circle1: " + circle1.toString();
23
24         // usa relacionamento "é um" para fazer referência a
25         // um Circle com uma referência a Point
26         point2 = circle1;    // atribui Circle a uma referência para Point
27
28         output += "\n\nCircle circle1 (via point2 reference): " +
29             point2.toString();
30
31         // usa "downcasting" (coerção de uma referência para superclasse para
32         // um tipo de dado de subclasse) para atribuir point2 a circle2
33         circle2 = ( Circle ) point2;
```

Fig. 9.6 Atribuindo referências para subclasse a referências para superclasse (parte 1 de 2).

```

34     output += "\n\nCircle circle1 (via circle2): " +
35         circle2.toString();
36
37
38     DecimalFormat precision2 = new DecimalFormat( "0.00" );
39     output += "\nArea of c (via circle2): " +
40         precision2.format( circle2.area() );
41
42     // tentando fazer referência a um objeto Point com uma referência a Circle
43     if ( point1 instanceof Circle ) {
44         circle2 = ( Circle ) point1;
45         output += "\nncast successful";
46     }
47     else
48         output += "\npoint1 does not refer to a Circle";
49
50     JOptionPane.showMessageDialog( null, output,
51         "Demonstrating the \"is a\" relationship",
52         JOptionPane.INFORMATION_MESSAGE );
53
54     System.exit( 0 );
55 }
56
57 } // fim da classe InheritanceTest

```

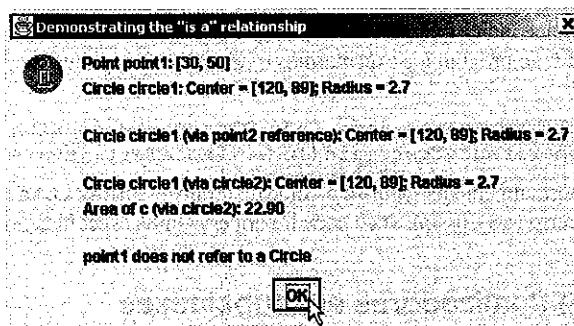


Fig. 9.6 Atribuindo referências para subclasse a referências para superclasse (parte 2 de 2).

A linha 26 atribui **circle1** (uma referência para um objeto da subclasse **Circle**) a **point2** (uma referência para a superclasse **Point**). É sempre aceitável, em Java, atribuir uma referência para subclasse a uma referência para superclasse, por causa do relacionamento “é um” da herança. O **Circle** é um **Point**, porque a classe **Circle** estende a classe **Point**. Atribuir uma referência para superclasse a uma referência para subclasse é perigoso, como veremos.

As linhas 28 e 29 acrescentam o resultado de **point2.toString()** a **output**. O interessante é que, quando **point2** é enviado à mensagem **toString**, Java sabe que o objeto é, na verdade, um **Circle**, então escolhe o método **toString** da classe **Circle** em vez de utilizar o método **toString** da classe **Point**, como você poderia estar esperando. Isso é um exemplo de *polimorfismo* e *vinculação dinâmica* – conceitos que tratamos em profundidade mais adiante neste capítulo. O compilador examina a expressão precedente e faz a pergunta: “O tipo de dados da referência **point2** (isto é, **Point**) tem um método **toString** sem argumentos?” A resposta a essa pergunta é sim (veja a definição de **toString** de **Point** na linha 41 da Fig. 9.4). O compilador simplesmente verifica a sintaxe da expressão e assegura que o método existe. Durante a execução, o interpretador faz a pergunta: “De que tipo é o objeto ao qual **point2** faz referência?” Cada objeto em Java conhece seu próprio tipo de dados, assim a resposta à pergunta é que **point2** faz referência a um objeto **Circle**. Com base nessa resposta, o interpretador chama o método **toString** do tipo de dados do objeto real – o método **toString** da classe **Circle**. Veja a ter-

ceira linha da captura de tela para confirmar isso. As duas principais técnicas de programação que utilizamos para alcançar esse efeito de polimorfismo são

1. estender a classe `Point` para criar a classe `Circle`; e
2. sobrescrever o método `toString` com exatamente a mesma assinatura na classe `Point` e na classe `Circle`.

A linha 33 faz a coerção de `point2`, que comprovadamente está fazendo referência a um `Circle` nesse momento na execução do programa, para um `Circle` e atribui o resultado a `circle2` (essa coerção seria perigosa se `point2` realmente estivesse fazendo referência a um `Point`, como discutiremos a seguir). Então utilizamos `circle2` para acrescentar a `output` os vários fatos sobre `circle2`. As linhas 35 e 36 invocam o método `toString` para acrescentar a representação como `String` do `Circle`. As linhas 39 e 40 acrescentam a `area` do `Circle` a `output`.

Em seguida, a estrutura `if/else` nas linhas 43 a 48 tenta uma coerção perigosa na linha 44. Fazemos coerção de `point1` para um `Circle`. Se o programa tentasse executar esta instrução, Java determinaria que `point1` na verdade faz referência a um `Point`, reconheceria que a coerção para `Circle` é perigosa e indicaria uma coerção imprópria com a mensagem `ClassCastException`. Entretanto, evitamos a execução dessa instrução com a condição `if`

```
if ( point1 instanceof Circle ) {
```

que utiliza o operador `instanceof` para determinar se o objeto ao qual `point1` faz referência é um `Circle`. Essa condição é avaliada como `true` somente se o objeto ao qual `point1` faz referência é um `Circle`; caso contrário, a condição é avaliada como `false`. A referência `point1` não faz referência a um `Circle`, de modo que a condição falha e um `String` que indica que `point1` não faz referência a um `Circle` é acrescentado a `output`.

Se removermos o teste `if` do programa e executarmos o programa, a seguinte mensagem será gerada durante a execução:

```
Exception in thread "main" java.lang.ClassCastException: Point
at InheritanceTest.main(InheritanceTest.java:43)
```

Entre as mensagens de erro normalmente incluem-se o nome do arquivo (`InheritanceTest.java`) e o número da linha em que ocorreu o erro (43), assim, você pode ir para essa linha específica no programa para depurar.

9.5 Construtores e finalizadores em subclasses

Quando um objeto de uma classe subclasse é instanciado, o construtor da superclasse deve ser chamado para fazer qualquer inicialização necessária das variáveis de instância da superclasse do objeto de subclasse. Uma chamada explícita ao construtor de superclasse (através da referência `super`) pode ser fornecida como a primeira instrução no construtor de subclasse. Caso contrário, o construtor de subclasse chamará o construtor `default` da superclasse (ou construtor sem argumentos) implicitamente.

Os construtores de superclasse não são herdados por subclasses. Os construtores de subclasse, entretanto, podem chamar os construtores de superclasse através da referência `super`.



Observação de engenharia de software 9.10

Quando se cria um objeto de uma subclasse, primeiro o construtor da subclasse chama o construtor da superclasse (explicitamente através de `super` ou implicitamente), o construtor da superclasse é executado, então o resto do corpo do construtor da subclasse é executado.

Se as classes em sua hierarquia de classes definem métodos `finalize`, o método `finalize` da subclasse deveria invocar o método `finalize` da superclasse para assegurar que todas as partes de um objeto são finalizadas adequadamente se o coletor de lixo reivindicar a memória para o objeto.

O aplicativo das Figs. 9.7 a 9.9 mostra a ordem em que os construtores e os finalizadores da superclasse e da subclasse são chamados. Para o propósito desse exemplo, a classe `Point` e a classe `Circle` são simplificadas.

A classe `Point` (Fig. 9.7) contém dois construtores, um finalizador, um método `toString` e as variáveis de instância `protected x` e `y`. O construtor e o finalizador imprimem que estão sendo executados e depois exibem o `Point` para o qual são invocados. Observe o uso de `this` nas chamadas a `System.out.println` para provo-

car uma chamada implícita ao método `toString`. Repare na primeira linha do método `finalize` (linha 23). O método `finalize` sempre deve ser definido como `protected` para que as subclasses tenham acesso ao método, mas não as classes que simplesmente utilizam os objetos `Point`.

A classe `Circle` (Fig. 9.8) deriva-se de `Point` e contém dois construtores, um finalizador, um método `toString` e a variável de instância `radius` protegida. O construtor e o finalizador imprimem que estão sendo executados, depois exibem o `Circle` para o qual são invocados. Observe que o método `toString` de `Circle` invoca o `toString` de `Point` através de `super` (linha 19).



Erro comum de programação 9.11

Quando um método de superclasse é sobreescrito em uma subclass, é comum fazer a versão de subclass chamar a versão de superclasse e fazer algum trabalho adicional. Neste cenário, o método da superclasse executa as tarefas comuns a todas as subclasses daquela classe e o método da subclass executa tarefas adicionais específicas para uma determinada subclass.

```

1 // Fig. 9.7: Point.java
2 // Definição da classe Point
3 public class Point extends Object {
4     protected int x, y;    // coordenadas do Point
5
6     // construtor sem argumentos
7     public Point()
8     {
9         x = 0;
10        y = 0;
11        System.out.println( "Point constructor: " + this );
12    }
13
14    // construtor
15    public Point( int xCoordinate, int yCoordinate )
16    {
17        x = xCoordinate;
18        y = yCoordinate;
19        System.out.println( "Point constructor: " + this );
20    }
21
22    // finalizador
23    protected void finalize()
24    {
25        System.out.println( "Point finalizer: " + this );
26    }
27
28    // converte Point para uma representação como String
29    public String toString()
30    {
31        return "[" + x + ", " + y + "]";
32    }
33
34 } // fim da classe Point

```

Fig. 9.7 Definição da classe `Point` para demonstrar quando os construtores e finalizadores são chamados.

```

1 // Fig. 9.8: Circle.java
2 // Definição da classe Circle
3 public class Circle extends Point {    // herda de Point
4     protected double radius;

```

Fig. 9.8 Definição da classe `Circle` para demonstrar quando os construtores e finalizadores são chamados (parte 1 de 2).

```

5
6     // construtor sem argumentos
7     public Circle()
8     {
9         // chamada implícita para o construtor da superclasse aqui
10        radius = 0;
11        System.out.println( "Circle constructor: " + this );
12    }
13
14    // Construtor
15    public Circle( double circleRadius, int xCoordinate,
16                  int yCoordinate )
17    {
18        // chama o construtor da superclasse
19        super( xCoordinate, yCoordinate );
20
21        radius = circleRadius;
22        System.out.println( "Circle constructor: " + this );
23    }
24
25    // finalizador
26    protected void finalize()
27    {
28        System.out.println( "Circle finalizer: " + this );
29        super.finalize(); // chama o método finalize da superclasse
30    }
31
32    // converte o Circle para um String
33    public String toString()
34    {
35        return "Center = " + super.toString() +
36               "; Radius = " + radius;
37    }
38
39 } // fim da classe Circle

```

Fig. 9.8 Definição da classe `Circle` para demonstrar quando os construtores e finalizadores são chamados (parte 2 de 2).



Erro comum de programação 9.5

Quando um método sobreescrito chama a versão de superclasse do mesmo método, não usar a palavra-chave `super` para fazer referência ao método da superclasse provoca recursão infinita, porque o método da subclasse na verdade chama a si mesmo.



Erro comum de programação 9.6

Colocar em cascata referências `super` para fazer referência a um membro (método ou variável) vários níveis acima na hierarquia (como em `super.super.x`) é um erro de sintaxe.

A classe de aplicativo `Test` (Fig. 9.9) usa essa hierarquia de herança `Point/Circle`. O aplicativo inicia no método `main` instanciando o objeto `Circle circle1` (linha 11). Isso invoca o construtor `Circle` na linha 15 da Fig. 9.8, que imediatamente invoca o construtor `Point` na linha 15 da Fig. 9.7. O construtor `Point` envia para a saída os valores recebidos do construtor `Circle`, chamando implicitamente o método `toString`, e devolve o controle do programa para o construtor `Circle`. Assim, o construtor `Circle` gera como saída o `Circle` completo chamando o método `toString`. Observe que as duas primeiras linhas da saída desse programa mostram valores para `x`, `y` e `radius`. O polimorfismo mais uma vez está fazendo com que o método `toString` de `Circle` seja executado, porque é um objeto `Circle` que está sendo criado. Quando `toString` é invocado a partir do construtor `Point`, `0.0` é exibido para `radius` porque `radius` ainda não foi inicializado no construtor `Circle`.

O objeto `Circle circle2` é instanciado a seguir. Novamente, ambos os construtores `Point` e `Circle` são executados. Repare, na janela de saída de linha de comando, que o corpo do construtor `Point` é executado antes do corpo do construtor `Circle`, mostrando que os objetos são construídos de “dentro para fora”.

```

1 // Fig. 9.9: Test.java
2 // Demonstra quando os construtores e finalizadores
3 // de superclasse e subclasse são chamados.
4 public class Test {
5
6     // testa quando os construtores e finalizadores são chamados
7     public static void main( String args[] )
8     {
9         Circle circle1, circle2;
10
11         circle1 = new Circle( 4.5, 72, 29 );
12         circle2 = new Circle( 10, 5, 5 );
13
14         circle1 = null; // marca para coleta de lixo
15         circle2 = null; // marca para coleta de lixo
16
17         System.gc(); // chama o coletor de lixo
18     }
19
20 } // fim da classe Test

```

```

Point constructor: Center = [72, 29]; Radius = 0.0
Circle constructor: Center = [72, 29]; Radius = 4.5
Point constructor: Center = [5, 5]; Radius = 0.0
Circle constructor: Center = [5, 5]; Radius = 10.0
Circle finalizer: Center = [72, 29]; Radius = 4.5
Point finalizer: Center = [72, 29]; Radius = 4.5
Circle finalizer: Center = [5, 5]; Radius = 10.0
Point finalizer: Center = [5, 5]; Radius = 10.0

```

Fig. 9.9 Ordem na qual os construtores e finalizadores são chamados.

As linhas 14 e 15 configuraram `circle1` como `null`, depois configuraram `circle2` como `null`. Cada um desses objetos não é mais necessário no programa, de modo que Java marca a memória ocupada por `circle1` e `circle2` para *coleta de lixo*. Java garante que, antes que o coletor de lixo seja executado para reivindicar o espaço de cada um desses objetos, os métodos `finalize` para cada objeto serão chamados. O coletor de lixo é uma *thread* de baixa prioridade que é executada automaticamente sempre que o tempo do processador está disponível. Escolhemos aqui pedir para o coletor de lixo seja executado com a chamada para o método `static gc` da classe `System` na linha 17. Java não garante a ordem em que os objetos serão coletados pelo coletor de lixo; portanto, Java não pode garantir qual finalizador de objeto será executado primeiro. Repare, na janela de saída de linha de comando, que os métodos `finalize` são chamados tanto para `Circle` quanto para `Point` quando cada objeto `Circle` é coletado pelo coletor de lixo.

9.6 Conversão implícita de objeto de subclasse para objeto de superclasse

Apesar do fato de que um objeto de subclasse também “é um” objeto de superclasse, o tipo da subclasse e o tipo da superclasse são diferentes. Os objetos de subclasse podem ser tratados como objetos de superclasse. Isso faz sentido porque a subclasse tem membros correspondentes a cada um dos membros da superclasse – lembre-se de que a subclasse normalmente tem mais membros que a superclasse. A atribuição na outra direção não é permitida porque

atribuir um objeto de superclasse a uma referência para subclasse deixaria os membros adicionais da subclasse indefinidos.

Uma referência para um objeto de subclasse pode ser convertida implicitamente em uma referência para um objeto de superclasse porque um objeto de subclasse é *um* objeto de superclasse por herança.

Há quatro maneiras possíveis de misturar e corresponder referências para superclasse e referências para subclasse com objetos de superclasse e objetos de subclasse:

1. Fazer referência a um objeto de superclasse com uma referência para superclasse é simples e direto.
2. Fazer referência a um objeto de subclasse com uma referência para subclasse é simples e direto.
3. Fazer referência a um objeto de subclasse com uma referência para superclasse é seguro, porque o objeto de subclasse também é *um* objeto de sua superclasse. Tal código pode fazer referência apenas aos membros de superclasse. Se esse código fizer referência aos membros que existem apenas na subclasse através da referência para superclasse, o compilador informará um erro de sintaxe.
4. Fazer referência a um objeto de superclasse com uma referência para subclasse é um erro de sintaxe.

Por mais conveniente que seja poder tratar objetos de subclasse como objetos de superclasse e fazer isso manipulando todos esses objetos com referências de superclasse, parece haver um problema. Em um sistema de folha de pagamento, por exemplo, gostaríamos de poder percorrer um *array* de empregados e calcular o pagamento semanal de cada pessoa. Mas a intuição sugere que utilizar referências de superclasse permitiria ao programa chamar apenas a rotina de cálculo de folha de pagamento da superclasse (se, de fato, houver essa rotina na superclasse). Precisamos de uma maneira de invocar a rotina de cálculo de folha de pagamento adequada para cada objeto, quer seja um objeto de superclasse, quer seja um objeto de subclasse, e fazer isso simplesmente utilizando a referência para superclasse. De fato, essa é precisamente a maneira como Java se comporta e isso é discutido neste capítulo ao considerarmos o polimorfismo e a vinculação dinâmica.

9.7 Engenharia de software com herança

Podemos utilizar herança para personalizar *softwares* existentes. Quando utilizamos herança para criar uma nova classe a partir de uma classe existente, a nova classe herda os atributos e comportamentos dessa classe existente; assim, podemos adicionar atributos e comportamentos ou sobreescriver comportamentos da superclasse para personalizar a classe a fim de atender às nossas necessidades.

Pode ser difícil para os alunos avaliar os problemas com que se defrontam os projetistas e implementadores em projetos de *software* de larga escala nas empresas. Pessoas experientes nesses projetos invariavelmente declararão que a chave para aprimorar o processo de desenvolvimento de *software* é estimular a reutilização de *software*. A programação orientada a objetos em geral e Java em particular certamente fazem isso.

É a disponibilidade de bibliotecas de classe substanciais e úteis que oferece os benefícios máximos da reutilização de *software* por herança. À medida que o interesse por Java crescer, o interesse pelas bibliotecas de classe Java aumentará. Assim como o *software* empacotado produzido por fornecedores independentes tornou-se um setor de crescimento explosivo com a chegada do computador pessoal, assim também se tornará a criação e a venda de bibliotecas de classe Java. Os projetistas de aplicativos construirão seus aplicativos com essas bibliotecas e os projetistas de biblioteca serão recompensados por ter suas bibliotecas empacotadas com os aplicativos. O que vemos se aproximando é um compromisso mundial em massa com o desenvolvimento de bibliotecas de classe Java para uma enorme variedade de áreas de aplicação.

Observação de engenharia de software 9.12



Criar uma subclasse não afeta o código-fonte da sua superclasse ou os bytecodes Java da superclasse; a integridade de uma superclasse é preservada por herança.

A superclasse especifica aspectos comuns. Todas as classes derivadas de uma superclasse herdam os recursos dessa superclasse. No processo de projeto orientado a objetos, o projetista procura o que há de comum entre um conjunto de classes e fatora isso para formar superclasses desejáveis. As subclasses são então personalizadas além das propriedades herdadas da superclasse.



Observação de engenharia de software 9.13

Assim como o projetista de sistemas não-orientados a objetos deve evitar a proliferação desnecessária de funções, o projetista de sistemas orientados a objetos deve evitar a proliferação desnecessária de classes. A proliferação de classes cria problemas de gerenciamento e pode impedir a reutilização de softwares, simplesmente porque é mais difícil para um usuário em potencial de uma classe encontrar essa classe em uma enorme coleção. A alternativa é criar menos classes, cada uma fornecendo funcionalidade substancial adicional, mas tais classes talvez sejam muito ricas para certos usuários.



Dica de desempenho 9.1

Quando criar uma nova classe, herde da classe “mais próxima” da que você precisa – isto é, aquela que oferece o conjunto mínimo de recursos necessários para uma nova classe executar suas tarefas. As subclasses podem herdar dados e funcionalidade que elas não irão usar; caso em que memória e recursos de processamento podem ser desperdiçados.

Observe que a leitura de um conjunto de declarações de subclasse pode ser confusa porque os membros herdados não são mostrados, mas os membros herdados, contudo, estão presentes nas subclasses. Um problema semelhante pode existir na documentação de subclasses.



Observação de engenharia de software 9.14

Em um sistema orientado a objetos, as classes são, com freqüência, proximamente relacionadas. “Fatore” atributos e comportamentos comuns e coloque-os em uma superclasse. Depois, utilize herança para formar subclasses sem ter que repetir atributos e comportamentos comuns.



Observação de engenharia de software 9.15

As modificações para uma superclasse não exigem que as subclasses mudem, contanto que a interface pública da superclasse permaneça inalterada.

9.8 Composição versus herança

Discutimos os relacionamentos é *um* que são implementados por herança. Também discutimos os relacionamentos tem *um* (e vimos exemplos nos capítulos anteriores), nos quais uma classe pode ter objetos de outras classes como membros – tais relacionamentos criam novas classes por *composição* de classes existentes. Por exemplo, dadas as classes **Empregado**, **DataDeNascimento** e **NumeroDeTelefone**, é impróprio dizer que o **Empregado** é *uma DataDeNascimento* ou que o **Empregado** é *um NumeroDeTelefone*. Mas é certamente apropriado dizer que o **Empregado** tem *uma DataDeNascimento* e que o **Empregado** tem *um NumeroDeTelefone*.

9.9 Estudo de caso: ponto, círculo, cilindro

Agora vejamos um exemplo substancial de herança. Considere uma hierarquia de ponto, círculo, cilindro. Primeiro, desenvolvemos e utilizamos a classe **Point** (Fig. 9.10 e Fig. 9.11). Depois apresentamos um exemplo em que derivamos a classe **Circle** da classe **Point** (Fig. 9.12 e Fig. 9.13). Por fim, apresentamos um exemplo em que derivamos a classe **Cylinder** da classe **Circle** (Fig. 9.14 e Fig. 9.15).

A Fig. 9.10 é a definição da classe **Point**. A classe **Point** é definida como parte do pacote **com.deitel.jhttp4.ch09** (linha 3). Observe que as variáveis de instância de **Point** são **protected**. Portanto, quando a classe **Circle** é derivada da classe **Point**, os métodos de classe **Circle** serão capazes de fazer referência direta às coordenadas **x** e **y**, em vez de utilizar métodos de acesso. Isso pode resultar em melhor desempenho.

```

1 // Fig. 9.10: Point.java
2 // Definição da classe Point
3 package com.deitel.jhttp4.ch09;
4

```

Fig. 9.10 Definição da classe **Point** (parte 1 de 2).

```

5  public class Point {
6      protected int x, y;      // coordenadas de Point
7
8      // construtor sem argumentos
9      public Point()
10     {
11         // a chamada implícita para o construtor da superclasse ocorre aqui
12         setPoint( 0, 0 );
13     }
14
15     // construtor
16     public Point( int xCoordinate, int yCoordinate )
17     {
18         // a chamada implícita para o construtor da superclasse ocorre aqui
19         setPoint( xCoordinate, yCoordinate );
20     }
21
22     // configura as coordenadas x e y de Point
23     public void setPoint( int xCoordinate, int yCoordinate )
24     {
25         x = xCoordinate;
26         y = yCoordinate;
27     }
28
29     // obtém a coordenada x
30     public int getX()
31     {
32         return x;
33     }
34
35     // obtém a coordenada y
36     public int getY()
37     {
38         return y;
39     }
40
41     // converte para uma representação de String
42     public String toString()
43     {
44         return "[" + x + ", " + y + "]";
45     }
46
47 } // fim da classe Point

```

Fig. 9.10 Definição da classe Point (parte 2 de 2).

A Fig. 9.11 mostra um aplicativo **Test** para testar a classe **Point**. O método **main** deve utilizar **getX** e **getY** para ler os valores das variáveis de instância **protected x** e **y**. Lembre-se de que as variáveis de instância **protected** são acessíveis apenas para os métodos de sua classe, suas subclasses e outras classes no mesmo pacote. Além disso, observe a chamada implícita para **toString** quando **point** é adicionado a um **String** na linha 25.

```

1 // Fig. 9.11: Test.java
2 // Applet para testar a classe Point
3
4 // Pacotes de extensão de Java
5 import javax.swing.JOptionPane;
6
7 // Pacotes Deitel
8 import com.deitel.jhtp4.ch09.Point;

```

Fig. 9.11 Testando a classe Point (parte 1 de 2).

```

9
10 public class Test {
11
12     // testa a classe Point
13     public static void main( String args[] )
14     {
15         Point point = new Point( 72, 115 );
16
17         // obtém coordenadas
18         String output = "X coordinate is " + point.getX() +
19                     "\nY coordinate is " + point.getY();
20
21         // configura coordenadas
22         point.setPoint( 10, 10 );
23
24         // usa chamada implícita para point.toString()
25         output += "\n\nThe new location of point is " + point;
26
27         JOptionPane.showMessageDialog( null, output,
28             "Demonstrating Class Point",
29             JOptionPane.INFORMATION_MESSAGE );
30
31         System.exit( 0 );
32     }
33
34 } // fim da classe Test

```

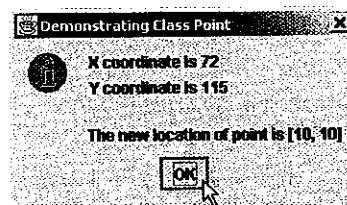


Fig. 9.11 Testando a classe `Point` (parte 2 de 2).

Nosso próximo exemplo importa a definição da classe `Point` da Fig. 9.10, de modo que não mostramos novamente a definição da classe. A Fig. 9.12 mostra a definição da classe `Circle` com as definições de métodos de `Circle`. Observe que a classe `Circle` estende (`extends`) a classe `Point`. Significa que a interface `public` para `Circle` inclui os métodos `Point`, os métodos `setRadius`, `getRadius`, `area` e `toString` de `Circle` e os construtores de `Circle`.

```

1 // Fig. 9.12: Circle.java
2 // Definição da classe Circle
3 package com.deitel.jhtp4.ch09;
4
5 public class Circle extends Point { // herda de Point
6     protected double radius;
7
8     // construtor sem argumentos
9     public Circle()
10    {
11        // a chamada implícita para o construtor da superclasse acontece aqui
12        setRadius( 0 );
13    }

```

Fig. 9.12 Definição da classe `Circle` (parte 1 de 2).

```

14
15 // construtor
16 public Circle( double circleRadius, int xCoordinate,
17   int yCoordinate )
18 {
19   // chama o construtor da superclasse para configurar as coordenadas
20   super( xCoordinate, yCoordinate );
21
22   // configura o raio
23   setRadius( circleRadius );
24 }
25
26 // configura o raio do Circle
27 public void setRadius( double circleRadius )
28 {
29   radius = ( circleRadius >= 0.0 ? circleRadius : 0.0 );
30 }
31
32 // obtém o raio do Circle
33 public double getRadius()
34 {
35   return radius;
36 }
37
38 // calcula a área do Circle
39 public double area()
40 {
41   return Math.PI * radius * radius;
42 }
43
44 // converte o Circle para um String
45 public String toString()
46 {
47   return "Center = " + "[" + x + ", " + y + "] " +
48     "; Radius = " + radius;
49 }
50
51 } // fim da classe Circle

```

Fig. 9.12 Definição da classe **Circle** (parte 2 de 2).

O aplicativo **Test** (Fig. 9.13) instancia um objeto da classe **Circle** (linha 19), depois utiliza os métodos *get* para obter as informações sobre o objeto **Circle**. O método **main** faz referência indiretamente aos dados **protected** da classe **Circle** através de chamadas de métodos. O método **main**, então, utiliza os métodos *set setRadius* e *setPoint* para redefinir o raio e as coordenadas do centro do círculo. Por fim, **main** exibe o objeto **Circle circle** e calcula e exibe sua área.

```

1 // Fig. 9.13: Test.java
2 // Applet para testar a classe Circle
3
4 // Pacotes do núcleo de Java
5 import java.text.DecimalFormat;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JOptionPane;
9
10 // Pacotes Deitel
11 import com.deitel.jhtp4.ch09.Circle;

```

Fig. 9.13 Testando a classe **Circle** (parte 1 de 2).

```

12
13 public class Test {
14
15     // testa a classe Circle
16     public static void main( String args[] )
17     {
18         // cria um Circle
19         Circle circle = new Circle( 2.5, 37, 43 );
20         DecimalFormat precision2 = new DecimalFormat( "0.00" );
21
22         // obtém as coordenadas e o raio
23         String output = "X coordinate is " + circle.getX() +
24             "\nY coordinate is " + circle.getY() +
25             "\nRadius is " + circle.getRadius();
26
27         // configura as coordenadas e o raio
28         circle.setRadius( 4.25 );
29         circle.setPoint( 2, 2 );
30
31         // obtém representação de Circle como String e calcula a área
32         output +=
33             "\n\nThe new location and radius of c are\n" + circle +
34             "\nArea is " + precision2.format( circle.area() );
35
36         JOptionPane.showMessageDialog( null, output,
37             "Demonstrating Class Circle",
38             JOptionPane.INFORMATION_MESSAGE );
39
40         System.exit( 0 );
41     }
42
43 } // fim da classe Test

```

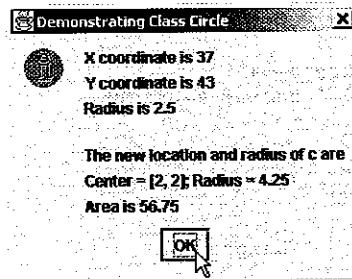


Fig. 9.13 Testando a classe `Circle` (parte 2 de 2).

Nosso último exemplo é mostrado na Fig. 9.14 e na Fig. 9.15. A Fig. 9.14 mostra a definição da classe `Cylinder` com as definições de métodos de `Cylinder`. Observe que a classe `Cylinder` estende (`extends`) a classe `Circle`. Significa que a interface `public` para `Cylinder` inclui os métodos de `Circle` e os métodos de `Point`, bem como o construtor de `Cylinder` e os métodos `setHeight`, `getHeight`, `area` (que sobrescreve o método `Circle area`), `volume` e `toString` de `Cylinder`.

```

1 // Fig. 9.14: Cylinder.java
2 // Definição da classe Cylinder
3 package com.deitel.jhtp4.ch09;
4
5 public class Cylinder extends Circle {

```

Fig. 9.14 Definição da classe `Cylinder` (parte 1 de 2).

```

6   protected double height;    // altura do Cylinder
7
8   // construtor sem argumentos
9   public Cylinder()
10  {
11     // chamada implícita ao construtor da superclasse aqui
12     setHeight( 0 );
13 }
14
15 // construtor
16 public Cylinder( double cylinderHeight, double cylinderRadius,
17   int xCoordinate, int yCoordinate )
18 {
19   // chama o construtor da superclasse para configurar coordenadas/raio
20   super( cylinderRadius, xCoordinate, yCoordinate );
21
22   // configura a altura do cilindro
23   setHeight( cylinderHeight );
24 }
25
26 // configura a altura de Cylinder
27 public void setHeight( double cylinderHeight )
28 {
29   height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
30 }
31
32 // obtém a altura de Cylinder
33 public double getHeight()
34 {
35   return height;
36 }
37
38 // calcula a área de Cylinder (i.e., a área da superfície)
39 public double area()
40 {
41   return 2 * super.area() +
42         2 * Math.PI * radius * height;
43 }
44
45 // calcula o volume de Cylinder
46 public double volume()
47 {
48   return super.area() * height;
49 }
50
51 // converte o Cylinder para um String
52 public String toString()
53 {
54   return super.toString() + "; Height = " + height;
55 }
56
57 } // fim da classe Cylinder

```

Fig. 9.14 Definição da classe **Cylinder** (parte 2 de 2).

O método **main** do aplicativo **Test** (Fig. 9.15) instancia um objeto de classe **Cylinder** (linha 19), depois utiliza os métodos **set** (linhas 23 a 26) para obter as informações sobre o objeto **Cylinder**. Novamente, o método **main** do aplicativo **Test** não pode fazer referência diretamente aos dados **protected** da classe **Cylinder**. O método **main** utiliza os métodos **set setHeight, setRadius e setPoint** (linhas 29 a 31) para redefinir **height, radius** e as coordenadas do **Cylinder**. Depois, **main** utiliza **toString, area e volume** para im-

```
1 // Fig. 9.15: Test.java
2 // Aplicativo que testa a classe Cylinder
3
4 // Pacotes do núcleo de Java
5 import java.text.DecimalFormat;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JOptionPane;
9
10 // Pacotes Deitel
11 import com.deitel.jhtp4.ch09.Cylinder;
12
13 public class Test {
14
15     // testa a classe Cylinder
16     public static void main( String args[] )
17     {
18         // cria o Cylinder
19         Cylinder cylinder = new Cylinder( 5.7, 2.5, 12, 23 );
20         DecimalFormat precision2 = new DecimalFormat( "0.00" );
21
22         // obtém coordenadas, raio e altura
23         String output = "X coordinate is " + cylinder.getX() +
24             "\nY coordinate is " + cylinder.getY() +
25             "\nRadius is " + cylinder.getRadius() +
26             "\nHeight is " + cylinder.getHeight();
27
28         // configura coordenadas, raio e altura
29         cylinder.setHeight( 10 );
30         cylinder.setRadius( 4.25 );
31         cylinder.setPoint( 2, 2 );
32
33         // obtém a representação de Cylinder como um String
34         // e calcula área e volume
35         output += "\n\nThe new location, radius " +
36             "and height of cylinder are\n" + cylinder +
37             "\nArea is " + precision2.format( cylinder.area() ) +
38             "\nVolume is " + precision2.format( cylinder.volume() );
39
40         JOptionPane.showMessageDialog( null, output,
41             "Demonstrating Class Cylinder",
42             JOptionPane.INFORMATION_MESSAGE );
43
44         System.exit( 0 );
45     }
46
47 } // fim da classe Test
```

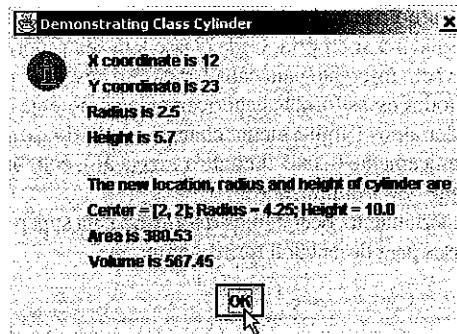


Fig. 9.15 Testando a classe Cylinder.

primir os atributos e alguns fatos sobre **Cylinder**. A Fig. 9.15 é um aplicativo **Test** que testa os recursos da classe **Cylinder**.

A série de exemplos nessa seção demonstra bem a herança, a definição e a referência para variáveis de instância **protected**. O leitor deve agora estar íntimo com os princípios básicos de herança. Nas próximas seções, mostraremos como programar com hierarquias de herança de maneira geral, utilizando polimorfismo. A abstração de dados, a herança e o polimorfismo são os pontos cruciais da programação orientada a objetos.

9.10 Introdução ao polimorfismo

Com o **polimorfismo**, é possível projetar e implementar sistemas que são mais facilmente *extensíveis*. Os programas podem ser escritos para processar genericamente – como objetos de superclasse – objetos de todas as classes existentes em uma hierarquia. As classes que não existem durante o desenvolvimento do programa podem ser adicionadas com pouca ou nenhuma modificação da parte genérica do programa – contanto que essas classes façam parte da hierarquia que está sendo processada genericamente. As únicas partes de um programa que requerem modificações são aquelas partes que exigem conhecimento direto da classe particular que é adicionada à hierarquia. Estudaremos duas hierarquias de classe substanciais e mostraremos como os objetos em todas essas hierarquias são manipulados com o polimorfismo.

9.11 Campos de tipo e instruções switch

Um meio de lidar com objetos de muitos tipos diferentes é utilizar uma instrução **switch** para executar uma ação apropriada sobre cada objeto de acordo com o tipo daquele objeto. Por exemplo, em uma hierarquia de formas em que cada forma tem uma variável de instância **shapeType**, a estrutura **switch** pode determinar qual método **print** chamar de acordo com o **shapeType** do objeto.

Há muitos problemas em se utilizar lógica de **switch**. O programador pode esquecer de fazer um teste de tipo como este quando um é garantido. Pode se esquecer de testar todos os casos possíveis em um **switch**. Se um sistema baseado em **switch** for modificado adicionando-se novos tipos, o programador talvez se esqueça de inserir os novos casos nas instruções **switch** existentes. Cada adição ou exclusão de uma classe exige que cada instrução **switch** no sistema seja modificada; monitorar isso pode consumir tempo e provocar erros.

Como veremos, a programação polimórfica pode eliminar a necessidade da lógica de **switch**. O programador pode utilizar o mecanismo de polimorfismo de Java para executar a lógica equivalente automaticamente, evitando, assim, os tipos de erros geralmente associados com a lógica **switch**.



Dica de teste e depuração 9.2

Uma consequência interessante de se utilizar polimorfismo é que os programas assumem uma aparência simplificada. Eles contêm menos lógica de desvio a favor de código seqüencial mais simples. Essa simplificação facilita o teste, a depuração e a manutenção do programa.

9.12 Vinculação dinâmica de método

Suponha que um conjunto de classes de formas como **Circulo**, **Triangulo**, **Retangulo**, **Quadrado**, etc. seja inteiramente derivado da superclasse **Forma**. Em programação orientada a objetos, cada uma dessas classes pode ser dotada da capacidade de desenhar a si própria. Cada classe tem seu próprio método **desenhar**, e a implementação do método **desenhar** é bem diferente para cada forma. Ao desenhar uma forma, qualquer que seja essa forma, seria ótimo se poder tratar todas essas formas genericamente como objetos da superclasse **Forma**. Assim, para desenhar qualquer forma, poderíamos simplesmente chamar o método **desenhar** da superclasse **Forma** e deixar o programa determinar dinamicamente (isto é, durante a execução), com base no tipo de objeto real, qual é o método **desenhar** de subclasse que deve ser utilizado.

Para permitir esse tipo de comportamento, declaramos **desenhar** na superclasse e então sobrescrevemos **desenhar** em cada uma das subclasses para desenhar a forma apropriada.



Observação de engenharia de software 9.16

Quando uma subclasse opta por não redefinir um método, ela simplesmente herda a definição de método de sua superclasse imediata.

Se utilizarmos uma referência para a superclasse para fazer referência a um objeto da subclasse e invocarmos o método **desenhar**, o programa escolherá o método **desenhar** correto da subclasse dinamicamente (isto é, durante a execução). Esse processo se chama *vinculação dinâmica de método*. A vinculação dinâmica de método é um importante mecanismo para a implementação do processamento polimórfico de objetos e será ilustrada nos estudos de caso mais adiante neste capítulo.

9.13 Métodos e classes final

Vimos no Capítulo 7 que as variáveis podem ser declaradas **final** para indicar que não podem ser modificadas depois que são declaradas e que devem ser inicializadas quando declaradas. Também é possível definir métodos e classes com o modificador **final**.

O método que é declarado **final** não pode ser sobreescrito em uma subclasse. Os métodos que são declarados **static** e os métodos que são declarados **private** são implicitamente **final**. Como a definição de um método **final** nunca pode ser alterada, o compilador pode otimizar o programa removendo as chamadas aos métodos **final** e substituindo-as pelo código expandido de suas definições em cada ponto de chamada do método – uma técnica conhecida como *inclusão de código inline*.

A classe que é declarada **final** não pode ser uma superclasse (isto é, classe não pode herdar de a classe **final**). Todos os métodos em uma classe **final** são implicitamente **final**.

Dica de desempenho 9.2



O compilador pode decidir substituir por uma inclusão inline uma chamada de método **final** e fará isso para os métodos **final** pequenos e simples. A inclusão inline não viola o encapsulamento nem o ocultamento de informações (mas aprimora o desempenho porque elimina o custo de se fazer uma chamada de método).

Observação de engenharia de software 9.17



A classe declarada **final** não pode ser estendida e cada método é implicitamente **final**.

Observação de engenharia de software 9.18



Na Java API, a grande maioria dos milhares de classes não é declarada **final**. Isto permite o processamento de herança e polimorfismo – os recursos fundamentais da programação orientada a objetos. No entanto, em alguns casos é importante declarar classes como **final** – geralmente por razões de segurança¹ ou desempenho.

9.14 Superclasses abstratas e classes concretas

Quando pensamos em uma classe como um tipo, supomos que os objetos desse tipo serão instanciados. Entretanto, há casos em que é útil definir classes para as quais o programador nunca pretende instanciar nenhum objeto. Essas classes são chamadas de *classes abstratas*. Como essas classes são utilizadas como superclasses em situações de herança, normalmente nos referimos a elas como *superclasses abstratas*. Nenhum objeto de superclasses abstratas pode ser instanciado.

O único propósito de uma classe abstrata é fornecer uma superclasse apropriada da qual outras classes podem herdar interface e/ou implementação (veremos exemplos de cada uma brevemente). As classes da qual os objetos podem ser instanciados chamam-se *classes concretas*.

¹ A classe **String** é um exemplo de uma classe **final**. Esta classe não pode ser estendida, de modo que os programas que usam **Strings** possam confiar na funcionalidade de objetos **String** como especificada na Java API. Tornar a classe **final** também evita que os programadores criem subclasses que possam contornar restrições de segurança. Por exemplo, quando um programa Java tenta abrir um arquivo em seu computador, o programa fornece um **String** que representa o nome do arquivo. Em muitos casos, abrir um arquivo está sujeito a restrições de segurança. Se fosse possível criar uma subclass de **String**, aquela subclass poderia ser implementada de uma maneira que lhe permitiria especificar um **String** que passasse em um teste de segurança ou permissão, e depois especificar um nome diferente quando o programa realmente abrisse o arquivo. Para obter mais informações sobre classes e métodos **final**, visite o endereço: java.sun.com/docs/books/tutorial/java/javaOO/final.html [Nota: os **Strings** são abordados em detalhes no Capítulo 10 e o processamento de arquivos é abordado em detalhe no Capítulo 16].

Poderíamos ter uma superclasse abstrata **ObjetoBidimensional** e derivar as classes concretas como **Quadrado**, **Círculo** e **Triângulo**. Também poderíamos ter uma superclasse abstrata **ObjetoTridimensional** e derivar as classes concretas como **Cubo**, **Esfera** e **Cilindro**. As superclasses abstratas são muito genéricas para definir os objetos reais; precisamos ser mais específicos antes de podermos pensar em instanciar objetos. Por exemplo, se alguém lhe dissesse para “desenhar a forma”, que forma você desenharia? As classes concretas fornecem os aspectos específicos que tornam razoável o ato de instanciar objetos.

Uma classe torna-se abstrata declarando-a com a palavra-chave **abstract**. A hierarquia não precisa conter nenhuma classe **abstract** mas, como veremos, muitos sistemas bons orientados a objetos têm hierarquias de classe baseadas em superclasses **abstract**. Em alguns casos, as classes **abstract** constituem os poucos níveis superiores da hierarquia. Um bom exemplo disso é a hierarquia de forma na Fig. 9.3. A hierarquia inicia com a superclasse **abstract Forma**. No nível abaixo a seguir, temos mais duas superclasses **abstract**, a saber, **FormaBidimensional** e **FormaTridimensional**. O nível abaixo seguinte começaria a definir as classes concretas para formas bidimensionais como **Círculo** e **Quadrado** e as classes concretas para formas tridimensionais como **Esfera** e **Cubo**.

9.15 Exemplos de polimorfismo

Eis um exemplo de polimorfismo. Se a classe **Retângulo** for derivada da classe **Quadrilátero**, então um objeto **Retângulo** é uma versão mais específica de um objeto **Quadrilátero**. Uma operação (como calcular o perímetro ou a área) que pode ser realizada sobre um objeto da classe **Quadrilátero** também pode ser realizada sobre um objeto da classe **Retângulo**. Essas operações também podem ser realizadas sobre outros “tipos de” **Quadrilateros**, como **Quadrados**, **Paralelogramos** e **Trapezios**. Quando se faz uma solicitação através de uma referência para superclasse para utilizar um método, Java escolhe polimorficamente o método sobrescrito correto na subclasse apropriada associada com o objeto.

Eis outro exemplo de polimorfismo. Suponha que temos um vídeo game que manipula objetos diversos, incluindo objetos das classes **Marciano**, **Venusiano**, **Plutôniano**, **NaveEspacial**, **RaioLaser** e outros mais. Cada uma dessas classes estende uma superclasse comum como **PecaDoJogo** que contém um método chamado **desenheASiProprio**. Esse método é definido por cada subclasse. Um programa Java de gerenciamento de tela simplesmente manteria algum tipo de contêiner (como um *array PecaDoJogo*) de referências a objetos dessas várias classes. Para atualizar a tela periodicamente, o gerenciador de tela simplesmente enviaria a mesma mensagem para cada objeto, a saber, **desenheASiProprio**. Cada objeto responderia de sua própria maneira exclusiva. O objeto **Marciano** desenharia a si mesmo com o número apropriado de antenas. O objeto **NaveEspacial** desenharia a si mesmo como brilhante e prateado. O objeto **CanhaoDeLaser** desenharia a si mesmo como um feixe vermelho brilhante cruzando a tela. Portanto, a mesma mensagem enviada para uma variedade de objetos assumiria “muitas formas” de resultados – daí o termo *polimorfismo*.

Tal gerenciador de tela polimórfico torna especialmente fácil adicionar novos tipos de objetos a um sistema, com impacto mínimo. Suponha que queremos adicionar **Mercurianos** ao nosso vídeo game. Certamente temos de construir uma nova classe **Mercuriano** que estende **PecaDoJogo** e fornece sua própria definição do método **desenheASiProprio**. Então, quando os objetos da classe **Mercuriano** aparecem no contêiner, o gerenciador de tela não precisa ser modificado. Ele simplesmente envia a mensagem **desenheASiProprio** para cada objeto no contêiner independentemente do tipo de objeto, de modo que os novos objetos **Mercurianos** apenas “se ajustam”. Portanto, com o polimorfismo, novos tipos de objetos nem mesmo vislumbrados quando se criou um sistema podem ser adicionados sem modificações no sistema (exceto a própria nova classe, naturalmente).

Pelo uso de polimorfismo, uma chamada de método pode fazer com que ocorram ações diferentes, dependendo do tipo do objeto que recebe a chamada. Isso fornece tremenda capacidade de expressão para o programador. Veremos exemplos do poder do polimorfismo nas próximas seções.



Observação de engenharia de software 9.19

Com o polimorfismo, o programador pode tratar de aspectos gerais e deixar o ambiente de tempo de execução preocupar-se com os aspectos específicos. Ele pode instruir uma ampla variedade de objetos a se comportar de maneiras apropriadas para esses objetos sem nem mesmo conhecer os tipos desses objetos.



Observação de engenharia de software 9.20

O polimorfismo promove a capacidade de extensão: o software escrito para invocar comportamento polimórfico é escrito independentemente dos tipos de objeto para os quais as mensagens (isto é, as chamadas de método) são en-

viadas. Portanto, novos tipos de objetos que podem responder às mensagens existentes podem ser adicionados nesse sistema sem modificar o sistema básico.

Observação de engenharia de software 9.21



Se um método é declarado **final**, ele não pode ser sobreescrito em subclasses, de modo que as chamadas de método não podem ser enviadas polimorficamente para os objetos dessas subclasses. A chamada de método ainda pode ser enviada para as subclasses, mas todas elas responderão de maneira idêntica, e não polimorficamente.

Observação de engenharia de software 9.22



Uma classe **abstract** define uma interface comum para os vários membros de uma hierarquia de classe. A classe **abstract** contém métodos que serão definidos nas subclasses. Todas as classes na hierarquia podem utilizar essa mesma interface por meio de polimorfismo.

Apesar de não podermos instanciar objetos de superclasses **abstract**, podemos declarar referências para superclasses **abstract**. Podem-se usar essas referências para permitir manipulações polimórficas de objetos de subclasse quando esses objetos são instanciados a partir de classes concretas.

Vamos analisar mais aplicações do polimorfismo. O gerenciador de tela precisa exibir diversos objetos, incluindo novos tipos de objetos que serão adicionados ao sistema mesmo depois que o gerenciador de tela for escrito. O sistema pode precisar exibir várias formas (isto é, a superclasse é **Forma**) como **Círculo**, **Triângulo** e **Retângulo**. Cada classe de forma é derivada da superclasse **Forma**. O gerenciador de tela utiliza referências para a superclasse **Forma** para gerenciar os objetos a serem exibidos. Para desenhar qualquer objeto (independentemente do nível em que esse objeto aparece na hierarquia de herança), o gerenciador de tela utiliza uma referência de superclasse para o objeto e simplesmente envia uma mensagem de **desenhar** para o objeto. O método **desenhar** foi declarado **abstract** na superclasse **Forma** e foi sobreescrito em cada uma das subclasses. Cada objeto **Forma** sabe desenhar a si mesmo. O gerenciador de tela não tem de se preocupar com o tipo de cada objeto ou com o fato de o gerenciador de tela ter ou não visto objetos desse tipo antes – o gerenciador de tela simplesmente instrui cada objeto a se **desenhar**.

O polimorfismo é particularmente eficaz para implementar sistemas de *software* em camadas. Em sistemas operacionais, por exemplo, cada tipo de dispositivo físico pode operar de maneira bastante diferente dos outros. Mesmo assim, os comandos para *ler* (*read*) os dados dos dispositivos ou *escrevê-los neles* (*write*) podem ter uma certa uniformidade. A mensagem de *escrever* enviada para um objeto controlador de dispositivo precisa ser interpretada especificamente no contexto desse controlador de dispositivo e como esse controlador de dispositivo manipula dispositivos de um tipo específico. Entretanto, a chamada de *escrever* propriamente dita não tem realmente nenhuma diferença de *escrever* em qualquer outro dispositivo no sistema – simplesmente transfira um número de *bytes* da memória para esse dispositivo. Um sistema operacional orientado a objetos poderia usar uma superclasse **abstract** para fornecer uma interface apropriada para todos os controladores de dispositivo. Então, por herança dessa superclasse **abstract**, são formadas subclasses que operam de maneira semelhante. Os recursos (isto é, a interface **public**) oferecidos pelos controladores de dispositivos são fornecidos como métodos **abstract** na superclasse **abstract**. As implementações desses métodos **abstract** são fornecidas nas subclasses que correspondem aos tipos de controladores de dispositivo específicos.

É comum em programação orientada a objetos definir uma classe **iterador** que pode percorrer todos os objetos em um contêiner (como um *array*). Caso se deseje imprimir uma lista de objetos em uma lista encadeada, por exemplo, pode-se instanciar um objeto iterador que devolverá o próximo elemento da lista encadeada toda vez que o iterador é chamado. Os iteradores são comumente utilizados em programação polimórfica para percorrer um *array* ou uma lista encadeada de objetos de vários níveis de uma hierarquia. Todas as referências nessa lista seriam referências de superclasse (veja mais sobre listas encadeadas no Capítulo 19). Uma lista de objetos da superclasse **Forma-Bidimensional** pode conter objetos das classes **Quadrado**, **Círculo**, **Triângulo**, etc. Enviar uma mensagem **desenhar** para cada objeto na lista, utilizando polimorfismo, desenharia a figura correta na tela.

9.16 Estudo de caso: um sistema de folha de pagamento utilizando polimorfismo

Vamos utilizar classes **abstract**, métodos **abstract** e polimorfismo para realizar cálculos de folha de pagamento baseados no tipo de empregado (Fig. 9.16). Utilizamos uma superclasse **abstract Employee**. As subclasses de **Employee** são **Boss** (Fig. 9.17) – pago com um salário fixo semanal independentemente do número de ho-

ras trabalhadas – **CommissionWorker** (Fig. 9.18) – pago com um salário fixo mais uma porcentagem sobre as vendas – **PieceWorker** (Fig. 9.19) – pago pelo número de itens produzidos – e **HourlyWorker** (Fig. 9.20) – que é pago por hora e que recebe um adicional por hora extra trabalhada. Cada subclasse de **Employee** foi declarada **final** porque não pretendemos herdar delas novamente.

A chamada de método **earnings** certamente se aplica genericamente a todos os empregados. Mas a maneira como os rendimentos de cada pessoa são calculados depende da classe do empregado, e essas classes derivam-se da superclasse **Employee**. Então **earnings** é declarado **abstract** na superclasse **Employee** e as implementações apropriadas de **earnings** são preparadas para cada uma das subclasses. Então, para calcular os rendimentos de qualquer empregado, o programa simplesmente utiliza uma referência de superclasse para o objeto desse empregado e invoca o método **earnings**. Em um sistema de folha de pagamento real, os vários objetos **Employee** poderiam ser mencionados por elementos individuais em um *array* de referências para **Employee**s. O programa simplesmente percorreria o *array*, um elemento por vez, utilizando as referências **Employee** para invocar o método **earnings** de cada objeto.



Observação de engenharia de software 9.23

*Se uma subclasse é derivada de uma superclasse com um método **abstract** e se nenhuma definição é fornecida na subclasse para esse método **abstract** (isto é, se ele não é sobreescrito na subclasse), ele permanece **abstract** na subclasse. Consequentemente, a subclasse também é uma classe **abstract** e deve ser explicitamente declarada como uma classe **abstract**.*



Observação de engenharia de software 9.24

*A capacidade de declarar um método **abstract** confere um considerável poder ao projetista de classe com relação à maneira como as subclasses são implementadas em uma hierarquia de classes. Qualquer nova classe que queira herdar dessa classe é forçada a sobreescrivê-lo o método **abstract** (tanto diretamente como ao herdar de uma classe que sobrecreveu o método). Caso contrário, essa nova classe conterá um método **abstract** e, portanto, será uma classe **abstract** incapaz de instanciar objetos.*



Observação de engenharia de software 9.25

*O **abstract** ainda pode ter dados de instância e métodos não-**abstract** sujeitos às regras normais de herança por subclasses. A classe **abstract** também pode ter construtores.*



Erro comum de programação 9.7

*Tentar instanciar um objeto de uma classe **abstract** (isto é, uma classe que contém um ou mais métodos **abstract**) é um erro de sintaxe.*



Erro comum de programação 9.8

*Ocorrerá um erro de sintaxe se uma classe com um ou mais métodos **abstract** não for explicitamente declarada **abstract**.*

Vamos considerar a classe **Employee** (Fig. 9.16). Os métodos **public** incluem um construtor que recebe o nome e o sobrenome como argumentos; um método **getFirstName** que retorna o nome; um método **getLastName** que retorna o sobrenome; um método **toString** que retorna o nome e o sobrenome separados por um espaço; e um método **abstract** – **earnings**. Por que esse método é **abstract**? A resposta é que não faz sentido fornecer uma implementação desse método na classe **Employee**. Não podemos calcular os rendimentos de um empregado genérico – devemos primeiro saber *de que tipo* de empregado se trata. Tornando esse método **abstract**, estamos indicando que forneceremos uma implementação em cada subclasse concreta, mas não na própria superclasse.

```

1 // Fig. 9.16: Employee.java
2 // Classe base abstrata Employee.
3
4 public abstract class Employee {
5     private String firstName;

```

Fig. 9.16 Superclasse abstract **Employee** (parte 1 de 2).

```

6     private String lastName;
7
8     // construtor
9     public Employee( String first, String last )
10    {
11        firstName = first;
12        lastName = last;
13    }
14
15    // obtém nome
16    public String getFirstName()
17    {
18        return firstName;
19    }
20
21    // obtém sobrenome
22    public String getLastname()
23    {
24        return lastName;
25    }
26
27    public String toString()
28    {
29        return firstName + ' ' + lastName;
30    }
31
32    // Método abstract que deve ser implementado para
33    // cada classe derivada de Employee a partir da
34    // qual os objetos são instanciados.
35    public abstract double earnings();
36
37 } // fim da classe Employee

```

Fig. 9.16 Superclasse **abstract Employee** (parte 2 de 2).

A classe **Boss** (Fig. 9.17) deriva-se da **Employee**. Os métodos **public** incluem um construtor que recebe um nome, um sobrenome e um salário semanal como argumentos e passa o nome e o sobrenome para o construtor **Employee** para inicializar os membros **firstName** e **lastName** da parte de superclasse do objeto da subclasse. Outros métodos **public** incluem um método **setWeeklySalary**, para atribuir um novo valor à variável de instância **private weeklySalary**; um método **earnings**, para definir como calcular os rendimentos de um **Boss**; e um método **toString**, que forma um **String** que contém o tipo do empregado (isto é, "Boss: ") seguido pelo nome do chefe (*boss*).

```

1 // Fig. 9.17: Boss.java
2 // Classe Boss derivada de Employee.
3
4 public final class Boss extends Employee {
5     private double weeklySalary;
6
7     // construtor para a classe Boss
8     public Boss( String first, String last, double salary )
9     {
10        super( first, last ); // chama o construtor da superclasse
11        setWeeklySalary( salary );
12    }
13
14    // configura o salário de Boss
15    public void setWeeklySalary( double salary )

```

Fig. 9.17 **Boss** estende a classe **abstract Employee** (parte 1 de 2).

```

16    {
17        weeklySalary = ( salary > 0 ? salary : 0 );
18    }
19
20    // obtém o rendimento de Boss
21    public double earnings()
22    {
23        return weeklySalary;
24    }
25
26    // obtém a representação do nome de Boss em forma de String
27    public String toString()
28    {
29        return "Boss: " + super.toString();
30    }
31
32 } // fim da classe Boss

```

Fig. 9.17 Boss estende a classe abstract Employee (parte 2 de 2).

A classe **CommissionWorker** (Fig. 9.18) deriva-se de **Employee**. Os métodos **public** incluem um construtor que recebe como argumentos um nome, um sobrenome, um salário, uma comissão e uma quantidade de itens vendidos e passa o nome e o sobrenome para o construtor **Employee**; os métodos **set**, para atribuir os novos valores às variáveis de instância **salary**, **commission** e **quantity**; um método **earnings**, para calcular os rendimentos de um **CommissionWorker**; e um método **toString**, que forma um **String** que contém o tipo de empregado (isto é, "Commission worker: ") seguido pelo nome do trabalhador comissionado (*commission worker*).

```

1 // Fig. 9.18: CommissionWorker.java
2 // Classe CommissionWorker derivada de Employee
3
4 public final class CommissionWorker extends Employee {
5     private double salary;      // salário básico semanal
6     private double commission; // valor por item vendido
7     private int quantity;      // total de itens vendidos na semana
8
9     // construtor para a classe CommissionWorker
10    public CommissionWorker( String first, String last,
11        double salary, double commission, int quantity )
12    {
13        super( first, last );    // chama o construtor da superclasse
14        setSalary( salary );
15        setCommission( commission );
16        setQuantity( quantity );
17    }
18
19    // configura o salário básico semanal do CommissionWorker
20    public void setSalary( double weeklySalary )
21    {
22        salary = ( weeklySalary > 0 ? weeklySalary : 0 );
23    }
24
25    // configura a comissão do CommissionWorker
26    public void setCommission( double itemCommission )
27    {
28        commission = ( itemCommission > 0 ? itemCommission : 0 );
29    }
30

```

Fig. 9.18 CommissionWorker estende a classe abstract Employee (parte 1 de 2).

```

31 // configura a quantidade vendida pelo CommissionWorker
32 public void setQuantity( int totalSold )
33 {
34     quantity = ( totalSold > 0 ? totalSold : 0 );
35 }
36
37 // determina os rendimentos do CommissionWorker
38 public double earnings()
39 {
40     return salary + commission * quantity;
41 }
42
43 // obtém representação do nome do CommissionWorker no formato de String
44 public String toString()
45 {
46     return "Commission worker: " + super.toString();
47 }
48
49 } // fim da classe CommissionWorker

```

Fig. 9.18 CommissionWorker estende a classe abstract Employee (parte 2 de 2).

A classe **PieceWorker** (Fig. 9.19) deriva-se de **Employee**. Os métodos **public** incluem um construtor que recebe um nome, um sobrenome, um salário por item produzido e uma quantidade de itens produzidos como argumentos e passa o nome e o sobrenome para o construtor **Employee**; os métodos *set*, para atribuir novos valores às variáveis de instância **wagePerPiece** e **quantity**; um método **earnings**, definindo como calcular os rendimentos de um **PieceWorker**; e um método **toString**, que forma um **String** que contém o tipo do empregado (isto é, "Piece worker: ") seguido pelo nome do trabalhador por produção (*piece worker*).

```

1 // Fig. 9.19: PieceWorker.java
2 // Classe PieceWorker derivada de Employee
3
4 public final class PieceWorker extends Employee {
5     private double wagePerPiece; // remuneração por peça produzida
6     private int quantity; // produção da semana
7
8     // construtor para a classe PieceWorker
9     public PieceWorker( String first, String last,
10         double wage, int numberOfItems )
11     {
12         super( first, last ); // chama o construtor da superclasse
13         setWage( wage );
14         setQuantity( numberOfItems );
15     }
16
17     // configura a remuneração do PieceWorker
18     public void setWage( double wage )
19     {
20         wagePerPiece = ( wage > 0 ? wage : 0 );
21     }
22
23     // configura a quantidade de itens produzidos
24     public void setQuantity( int numberOfItems )
25     {
26         quantity = ( numberOfItems > 0 ? numberOfItems : 0 );
27     }
28
29     // determina os rendimentos do PieceWorker

```

Fig. 9.19 PieceWorker estende a classe abstract Employee (parte 1 de 2).

```

30     public double earnings()
31     {
32         return quantity * wagePerPiece;
33     }
34
35     public String toString()
36     {
37         return "Piece worker: " + super.toString();
38     }
39
40 } // fim da classe PieceWorker

```

Fig. 9.19 PieceWorker estende a classe abstract Employee (parte 2 de 2).

A classe **HourlyWorker** (Fig. 9.20) deriva-se de **Employee**. Os métodos **public** incluem um construtor que recebe um nome, um sobrenome, um salário e o número de horas trabalhadas como argumentos e passa o nome e o sobrenome para o construtor **Employee**; os métodos **set**, para atribuir os novos valores para as variáveis de instância **wage** e **hours**; um método **earnings**, definindo como calcular os rendimentos de um **HourlyWorker**; e um método **toString**, que forma um **String** que contém o tipo do empregado (isto é, "Hourly worker:") seguido pelo nome do trabalhador horista (*hourly worker*).

```

1 // Fig. 9.20: HourlyWorker.java
2 // Definição da classe HourlyWorker
3
4 public final class HourlyWorker extends Employee {
5     private double wage; // remuneração por hora
6     private double hours; // horas trabalhadas por semana
7
8     // construtor para a classe HourlyWorker.
9     public HourlyWorker( String first, String last,
10         double wagePerHour, double hoursWorked )
11     {
12         super( first, last ); // chama o construtor da superclasse
13         setWage( wagePerHour );
14         setHours( hoursWorked );
15     }
16
17     // configura a remuneração
18     public void setWage( double wagePerHour )
19     {
20         wage = ( wagePerHour > 0 ? wagePerHour : 0 );
21     }
22
23     // configura as horas trabalhadas
24     public void setHours( double hoursWorked )
25     {
26         hours = ( hoursWorked >= 0 && hoursWorked < 168 ?
27             hoursWorked : 0 );
28     }
29
30     // obtém os rendimentos do HourlyWorker
31     public double earnings() { return wage * hours; }
32
33     public String toString()
34     {
35         return "Hourly worker: " + super.toString();
36     }
37
38 } // fim da classe HourlyWorker

```

Fig. 9.20 HourlyWorker estende a classe abstract Employee.

O método `main` do aplicativo `Test` (Fig. 9.21) começa declarando a referência `Employee ref`. Cada um dos tipos de `Employees` é tratado de maneira semelhante em `main`; assim, discutiremos somente o caso em que `main` lida com um objeto `Boss`.

```

1 // Fig. 9.21: Test.java
2 // Programa de teste para a hierarquia Employee
3
4 // Pacotes do núcleo de Java
5 import java.text.DecimalFormat;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JOptionPane;
9
10 public class Test {
11
12     // testa a hierarquia Employee
13     public static void main( String args[] )
14     {
15         Employee employee; // referência para a superclasse
16         String output = "";
17
18         Boss boss = new Boss( "John", "Smith", 800.0 );
19
20         CommissionWorker commisionWorker =
21             new CommissionWorker(
22                 "Sue", "Jones", 400.0, 3.0, 150 );
23
24         PieceWorker pieceWorker =
25             new PieceWorker( "Bob", "Lewis", 2.5, 200 );
26
27         HourlyWorker hourlyWorker =
28             new HourlyWorker( "Karen", "Price", 13.75, 40 );
29
30         DecimalFormat precision2 = new DecimalFormat( "0.00" );
31
32         // Referência Employee para um Boss
33         employee = boss;
34
35         output += employee.toString() + " earned $" +
36             precision2.format( employee.earnings() ) + "\n" +
37             boss.toString() + " earned $" +
38             precision2.format( boss.earnings() ) + "\n";
39
40         // Referência Employee para um CommissionWorker
41         employee = commisionWorker;
42
43         output += employee.toString() + " earned $" +
44             precision2.format( employee.earnings() ) + "\n" +
45             commissionWorker.toString() + " earned $" +
46             precision2.format(
47                 commissionWorker.earnings() ) + "\n";
48
49         // Referência Employee para um PieceWorker
50         employee = pieceWorker;
51
52         output += employee.toString() + " earned $" +
53             precision2.format( employee.earnings() ) + "\n" +
54             pieceWorker.toString() + " earned $" +
55             precision2.format( pieceWorker.earnings() ) + "\n";

```

Fig. 9.21 Testando a hierarquia de classes `Employee` usando uma superclasse `abstract` (parte 1 de 2).

```

56
57     // Referência Employee para um HourlyWorker
58     employee = hourlyWorker;
59
60     output += employee.toString() + " earned $" +
61         precision2.format( employee.earnings() ) + "\n" +
62         hourlyWorker.toString() + " earned $" +
63         precision2.format( hourlyWorker.earnings() ) + "\n";
64
65     JOptionPane.showMessageDialog( null, output,
66         "Demonstrating Polymorphism",
67         JOptionPane.INFORMATION_MESSAGE );
68
69     System.exit( 0 );
70 }
71
72 } // fim da classe Test

```

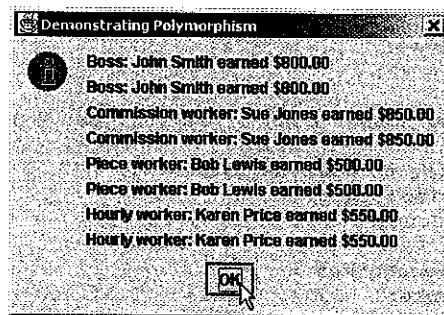


Fig. 9.21 Testando a hierarquia de classes `Employee` usando uma superclasse `abstract` (parte 2 de 2).

A linha 18 instancia a classe de objeto de subclasse `Boss` e fornece vários argumentos de construtor, incluindo um nome, um sobrenome e um salário fixo semanal. O novo objeto é atribuído à referência `Boss` para `boss`.

A linha 33 atribui à referência `employee` para a superclasse `Employee` uma referência para o objeto da subclasse `Boss` à qual `boss` faz referência. Isso é precisamente o que devemos fazer para efetivar o comportamento polimórfico.

A chamada de método na linha 35 invoca o método `toString` do objeto mencionado por `employee`. O sistema determina que o objeto mencionado é um `Boss` e invoca o método `toString` da subclasse `Boss` – novamente, comportamento polimórfico. Essa chamada de método é um exemplo de vinculação dinâmica de método – a decisão sobre qual método invocar é adiada até a hora da execução.

A chamada de método na linha 36 invoca o método `earnings` do objeto ao qual `employee` se refere. O sistema determina que o objeto é um `Boss` e invoca o método `earnings` da subclasse `Boss`, em vez do método `earnings` da superclasse. Isso também é um exemplo de vinculação dinâmica de método.

A chamada de método na linha 37 invoca explicitamente a versão de `Boss` do método `toString` colocando o operador ponto após a referência `boss`. Incluímos a linha 37 para fins de comparação, para assegurar que o método vinculado dinamicamente invocado com `employee.toString()` era, de fato, o método adequado.

A chamada de método na linha 38 invoca explicitamente a versão de `Boss` de método `earnings` utilizando o operador ponto com a referência específica `Boss` para `boss`. Essa chamada também é incluída para fins de comparação, para assegurar que o método vinculado dinamicamente invocado com `employee.earnings()` era, de fato, o método adequado.

Para provar que a referência de superclasse `employee` pode ser utilizada para invocar `toString` e `earnings` para os outros tipos de empregados, as linhas 41, 50 e 58 atribuem um tipo diferente de objeto `Employee` (`CommissionWorker`, `PieceWorker` e `HourlyWorker`, respectivamente) à referência de superclasse `employee`; a seguir, os dois métodos são chamados após cada atribuição, para mostrar que Java sempre pode determinar o tipo do objeto mencionado antes de invocar um método.

9.17 Novas classes e vinculação dinâmica

O polimorfismo certamente funciona bem quando todas as classes possíveis são conhecidas antecipadamente. Mas ele também funciona quando se adicionam novos tipos de classes aos sistemas.

As novas classes são acomodadas por vinculação dinâmica de método (também chamada de *vinculação tardia*). O tipo de um objeto não precisa ser conhecido durante a compilação para que uma chamada polimórfica seja compilada. Durante a execução, a chamada é associada ao método do objeto chamado.

Um programa gerenciador de tela agora pode tratar (sem recompilação) novos tipos de objetos a exibir à medida que são adicionados ao sistema. A chamada de método **desenhar** permanece a mesma. Cada um dos novos objetos contém, ele próprio, um método **desenhar** que implementa a capacidade real de desenhar. Isso torna fácil adicionar novas capacidades a sistemas com impacto mínimo. Isso também promove a reutilização de *software*.

Dica de desempenho 9.3



*Os tipos de manipulações polimórficas possibilitados pela vinculação dinâmica também podem ser conseguidos utilizando-se a lógica **switch** codificada manualmente com base em campos de tipo em objetos. O código polimórfico gerado pelo compilador Java é executado com desempenho comparável a uma lógica **switch** codificada eficientemente.*

Observação de engenharia de software 9.26



*Java oferece mecanismos para carregar dinamicamente as classes para um programa, para aumentar a funcionalidade de um programa que está sendo executado. Em particular, pode-se mudar o método estático **forName** da classe **Class** (pacote **java.lang**) para carregar uma definição de classe e depois criar novos objetos dessa classe para uso em um programa. Este conceito ultrapassa o escopo deste livro. Para obter mais informações, veja **Class** na documentação on-line da API.*

9.18 Estudo de caso: herança de interface e implementação

Nosso próximo exemplo (Fig. 9.22 a Fig. 9.26) reexamina a hierarquia **Point**, **Circle**, **Cylinder**, exceto pelo fato de que agora baseamos a hierarquia na superclasse **abstract Shape** (Fig. 9.22). Essa hierarquia demonstra mecanicamente o poder do polimorfismo. Nos exercícios, exploramos uma hierarquia de formas mais realista.

Shape contém o método **abstract getName** de modo que **Shape** deve ser declarada como superclasse **abstract**. **Shape** contém dois outros métodos, **area** e **volume**, e cada um deles tem uma implementação que retorna zero por *default*. **Point** herda essas implementações de **Shape**. Isso faz sentido porque tanto a área quanto o volume de um ponto são zero. **Circle** herda o método **volume** de **Point**, mas **Circle** fornece sua própria implementação para o método **area**. **Cylinder** fornece suas próprias implementações tanto para o método **area** (interpretada como a área da superfície do cilindro) quanto para o método **volume**.

Nesse exemplo, utiliza-se a classe **Shape** para definir um conjunto de métodos que todas as **Shapes** em nossa hierarquia têm em comum. Definir esses métodos na classe **Shape** nos permite chamar genericamente esses métodos através de uma referência **Shape**. Lembre-se, os únicos métodos que podem ser chamados por qualquer referência são aqueles métodos **public** definidos no tipo de classe declarado da referência e quaisquer métodos **public** herdados nessa classe. Portanto, podemos chamar métodos de **Object** e **Shape** através de uma referência **Shape**.

Observe que, embora **Shape** seja uma superclasse **abstract**, ela ainda contém implementações dos métodos **area** e **volume**, e essas implementações podem ser herdadas. A classe **Shape** fornece uma interface (conjunto de serviços) que pode ser herdada, na forma de três métodos que todas as classes da hierarquia conterão. A classe **Shape** também fornece algumas implementações que as subclasses nos primeiros níveis da hierarquia utilizarão.

Esse estudo de caso enfatiza que uma subclasse pode herdar interface e/ou implementação de uma superclasse.

Observação de engenharia de software 9.27



As hierarquias projetadas para implementação de herança tendem a ter sua funcionalidade em uma alta posição na hierarquia – cada nova subclasse herda um ou mais métodos que foram definidos em uma superclasse e utiliza as definições da superclasse.



Observação de engenharia de software 9.28

As hierarquias projetadas para herança de interface tendem a ter sua funcionalidade em uma posição mais baixa na hierarquia – uma superclasse específica um ou mais métodos que devem ser chamados identicamente para cada objeto na hierarquia (isto é, eles têm a mesma assinatura), mas as subclasses individuais fornecem suas próprias implementações do(s) método(s).

A superclasse **Shape** (Fig. 9.22) estende **Object**, consiste em três métodos **public** e não contém nenhum dado (embora pudesse). O método **getName** é **abstract**, por isso é sobreescrito em cada uma das subclasses. Os métodos **area** e **volume** são definidos para retornar **0.0**. Esses métodos são sobreescritos em subclasses quando é apropriado para essas classes ter um cálculo de **area** diferente (classes **Circle** e **Cylinder**) e/ou um cálculo de **volume** diferente (classe **Cylinder**).

A classe **Point** (Fig. 9.23) deriva-se de **Shape**. Um **Point** tem uma área de **0.0** e um volume de **0.0**, assim os métodos **area** e **volume** da superclasse não são sobreescritos aqui – são herdados como foram definidos em **Shape**. Outros métodos incluem **setPoint**, para atribuir as novas coordenadas **x** e **y** a um **Point** e **getX**, e **getY**, para retornar as coordenadas **x** e **y** de um **Point**. O método **getName** é uma implementação do método **abstract** na superclasse. Se esse método não fosse definido, a classe **Point** seria uma classe abstrata.

```

1 // Fig. 9.22: Shape.java
2 // Definição da classe base abstrata Shape
3
4 public abstract class Shape extends Object {
5
6     // devolve a área da forma
7     public double area()
8     {
9         return 0.0;
10    }
11
12     // devolve o volume da forma
13     public double volume()
14     {
15         return 0.0;
16     }
17
18     // o método abstrato deve ser definido por subclasses concretas
19     // para retornar o nome apropriado da forma
20     public abstract String getName();
21
22 } // fim da classe Shape

```

Fig. 9.22 Superclasse abstrata **Shape** para a hierarquia **Point**, **Circle** e **Cylinder**.

```

1 // Fig. 9.23: Point.java
2 // Definição da classe Point
3
4 public class Point extends Shape {
5     protected int x, y;    // coordenadas do Point
6
7     // construtor sem argumentos
8     public Point()
9     {
10        setPoint( 0, 0 );
11    }
12

```

Fig. 9.23 Subclasse **Point** da classe abstrata **Shape** (parte 1 de 2).

```

13  // construtor
14  public Point( int xCoordinate, int yCoordinate )
15  {
16      setPoint( xCoordinate, yCoordinate );
17  }
18
19  // configura as coordenadas x e y do Point
20  public void setPoint( int xCoordinate, int yCoordinate )
21  {
22      x = xCoordinate;
23      y = yCoordinate;
24  }
25
26  // obtém a coordenada x
27  public int getX()
28  {
29      return x;
30  }
31
32  // obtém a coordenada y
33  public int getY()
34  {
35      return y;
36  }
37
38  // converte o ponto para uma representação como String
39  public String toString()
40  {
41      return "[" + x + ", " + y + "]";
42  }
43
44  // retorna o nome da forma
45  public String getName()
46  {
47      return "Point";
48  }
49
50 } // fim da classe Point

```

Fig. 9.23 Subclasse Point da classe abstrata Shape (parte 2 de 2).

A classe **Circle** (Fig. 9.24) deriva-se de **Point**. O **Circle** tem um volume de 0.0; assim, o método de superclasse **volume** não é sobreescrito – é herdado da classe **Point**, que o herdou de **Shape**. O **Circle** tem uma área diferente daquela de **Point**, assim o método **área** é sobreescrito. O método **getName** é uma implementação do método **abstract** da superclasse. Se esse método não fosse redefinido aqui, a versão de **getName** de **Point** seria herdada. Outros métodos incluem **setRadius** para atribuir um novo **radius** a um **Circle**, e **getRadius**, para retornar o **radius** de um **Circle**.

```

1 // Fig. 9.24: Circle.java
2 // Definição da classe Circle
3
4 public class Circle extends Point { // herda de Point
5     protected double radius;
6
7     // construtor sem argumentos
8     public Circle()
9     {

```

Fig. 9.24 Subclasse **Circle** de **Point** – subclasse indireta da classe **abstract Shape** (parte 1 de 2).

```

10     // chamada implícita para o construtor da superclasse aqui
11     setRadius( 0 );
12 }
13
14 // construtor
15 public Circle( double circleRadius, int xCoordinate,
16     int yCoordinate )
17 {
18     // chama o construtor da superclasse
19     super( xCoordinate, yCoordinate );
20
21     setRadius( circleRadius );
22 }
23
24 // configura o raio do Circle
25 public void setRadius( double circleRadius )
26 {
27     radius = ( circleRadius >= 0 ? circleRadius : 0 );
28 }
29
30 // obtém o raio do Circle
31 public double getRadius()
32 {
33     return radius;
34 }
35
36 // calcula a área do Circle
37 public double area()
38 {
39     return Math.PI * radius * radius;
40 }
41
42 // converte Circle para uma representação de String
43 public String toString()
44 {
45     return "Center = " + super.toString() +
46         ", Radius = " + radius;
47 }
48
49 // devolve o nome da forma
50 public String getName()
51 {
52     return "Circle";
53 }
54
55 } // fim da classe Circle

```

Fig. 9.24 Subclasse **Circle** de **Point** – subclasse indireta da classe **abstract Shape** (parte 2 de 2).



Observação de engenharia de software 9.29

Uma subclasse sempre herda a versão mais recentemente definida de cada método **public e protected** de suas superclasses diretas e indiretas.

A classe **Cylinder** (Fig. 9.25) deriva-se de **Circle**. O **Cylinder** tem área e volume diferentes daqueles da classe **Circle**, assim os métodos **area** e **volume** são sobreescritos. O método **getName** é uma implementação do método **abstract** da superclasse. Se esse método não tivesse sido redefinido aqui, a versão de **getName** de **Circle** seria herdada. Outros métodos incluem **setHeight**, para atribuir uma nova **height** para um **Cylinder**, e **getHeight**, para retornar a **height** de um **Cylinder**.

```

1 // Fig. 9.25: Cylinder.java
2 // Definição da classe Cylinder.
3
4 public class Cylinder extends Circle {
5     protected double height;    // altura do Cylinder
6
7     // construtor sem argumentos
8     public Cylinder()
9     {
10         // chamada implícita para o construtor da superclasse aqui
11         setHeight( 0 );
12     }
13
14     // construtor
15     public Cylinder( double cylinderHeight,
16                     double cylinderRadius, int xCoordinate,
17                     int yCoordinate )
18     {
19         // chama o construtor da superclasse
20         super( cylinderRadius, xCoordinate, yCoordinate );
21
22         setHeight( cylinderHeight );
23     }
24
25     // configura a altura do Cylinder
26     public void setHeight( double cylinderHeight )
27     {
28         height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
29     }
30
31     // obtém a altura do Cylinder
32     public double getHeight()
33     {
34         return height;
35     }
36
37     // calcula a área do Cylinder (i.e., área da superfície do cilindro)
38     public double area()
39     {
40         return 2 * super.area() + 2 * Math.PI * radius * height;
41     }
42
43     // calcula o volume do Cylinder
44     public double volume()
45     {
46         return super.area() * height;
47     }
48
49     // converte o Cylinder para uma representação de String
50     public String toString()
51     {
52         return super.toString() + "; Height = " + height;
53     }
54
55     // devolve o nome da forma
56     public String getName()
57     {
58         return "Cylinder";
59     }
60 }
61 } // fim da classe Cylinder

```

Fig. 9.25 Subclasse Cylinder de Circle – subclasse indireta da classe abstrata Shape.

O método **main** da classe **Test** (Fig. 9.26) instancia o objeto **Point point**, o objeto **Circle circle** e o objeto **Cylinder cylinder** (linhas 16 a 18). Em seguida, o array **array arrayOfShapes** é instanciado (linha 21). Esse array de referências de superclasse **Shape** fará referência a cada objeto de subclasse instanciado. Na linha 24, a referência **point** é atribuída ao elemento do array **arrayOfShapes[0]**. Na linha 27, a referência **circle** é atribuída ao elemento do array **arrayOfShapes[1]**. Na linha 30, a referência **cylinder** é atribuída ao elemento do array **arrayOfShapes[2]**. Agora, cada referência de superclasse **Shape** no array faz referência a um objeto de subclasse do tipo **Point**, **Circle** ou **Cylinder**.

```

1 // Fig. 9.26: Test.java
2 // Classe que testa a hierarquia Shape, Point, Circle, Cylinder
3
4 // Pacotes do núcleo de Java
5 import java.text.DecimalFormat;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JOptionPane;
9
10 public class Test {
11
12     // testa a hierarquia de formas
13     public static void main( String args[] ) {
14
15         // cria formas
16         Point point = new Point( 7, 11 );
17         Circle circle = new Circle( 3.5, 22, 8 );
18         Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );
19
20         // cria array de Shapes
21         Shape arrayOfShapes[] = new Shape[ 3 ];
22
23         // aponta arrayOfShapes[ 0 ] para o objeto da subclasse Point
24         arrayOfShapes[ 0 ] = point;
25
26         // aponta arrayOfShapes[ 1 ] para o objeto da subclasse Circle
27         arrayOfShapes[ 1 ] = circle;
28
29         // aponta arrayOfShapes[ 2 ] para o objeto da subclasse Cylinder
30         arrayOfShapes[ 2 ] = cylinder;
31
32         // obtém o nome e a representação como String de cada forma
33         String output =
34             point.getName() + ": " + point.toString() + "\n" +
35             circle.getName() + ": " + circle.toString() + "\n" +
36             cylinder.getName() + ": " + cylinder.toString();
37
38         DecimalFormat precision2 = new DecimalFormat( "0.00" );
39
40         // percorre com um laço arrayOfShapes e obtém nome,
41         // área e volume de cada forma em arrayOfShapes
42         for ( int i = 0; i < arrayOfShapes.length; i++ ) {
43             output += "\n\n" + arrayOfShapes[ i ].getName() +
44                 ": " + arrayOfShapes[ i ].toString() +
45                 "\nArea = " +
46                 precision2.format( arrayOfShapes[ i ].area() ) +
47                 "\nVolume = " +
48                 precision2.format( arrayOfShapes[ i ].volume() );
49
50     }
}

```

Fig. 9.26 Hierarquia **Shape**, **Point**, **Circle** e **Cylinder** (parte 1 de 2).

```

51     JOptionPane.showMessageDialog( null, output,
52             "Demonstrating Polymorphism",
53             JOptionPane.INFORMATION_MESSAGE );
54
55     System.exit( 0 );
56 }
57
58 } // fim da classe Test

```

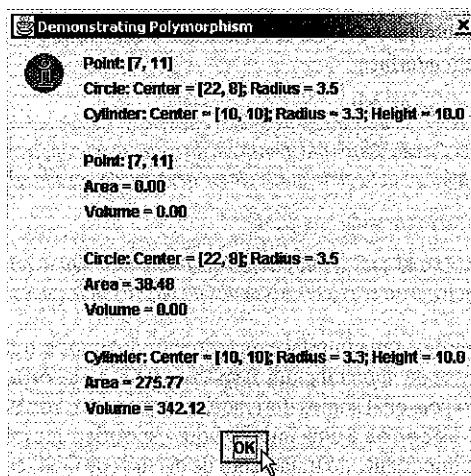


Fig. 9.26 Hierarquia **Shape**, **Point**, **Circle** e **Cylinder** (parte 2 de 2).

As linhas 33 a 36 invocam os métodos **getName** e **toString** para ilustrar que os objetos são inicializados corretamente (veja as primeiras três linhas da captura de tela).

Em seguida, a estrutura **for** nas linhas 42 a 49 percorre **arrayOfShapes** e as seguintes chamadas são feitas durante cada iteração do laço:

```

arrayOfShapes[ i ].getName()
arrayOfShapes[ i ].toString()
arrayOfShapes[ i ].area()
arrayOfShapes[ i ].volume()

```

Cada uma dessas chamadas de método é invocada sobre o objeto ao qual **arrayOfShapes[i]** faz referência atualmente. Quando o compilador analisa cada uma dessas chamadas, ele simplesmente está tentando determinar se pode utilizar uma referência para **Shape** (**arrayOfShapes[i]**) chamar esses métodos. Para os métodos **getName**, **area** e **volume** a resposta é sim, porque cada um desses métodos é definido na classe **Shape**. Para o método **toString**, o compilador primeiro analisa a classe **Shape** para determinar se **toString** não é definida lá; assim, o compilador prossegue para a superclasse de **Shape** (**Object**) para determinar se **Shape** herdou um método **toString** que não recebe argumentos (o que efetivamente ocorreu, porque todos os **Objects** têm um método **toString**).

A captura de tela ilustra que todos os quatro métodos são invocados adequadamente com base no tipo do objeto mencionado. Primeiro, o string "**Point:**" e as coordenadas do objeto **point** (**arrayOfShapes[0]**) são enviados para a saída; a área e o volume são 0. Em seguida, o string "**Circle:**", as coordenadas do objeto **circle** e o raio do objeto **circle** (**arrayOfShapes[1]**) são enviados para a saída; a área de **circle** é calculada e o volume é 0. Por fim, o string "**Cylinder:**", as coordenadas do objeto **cylinder**, o raio do objeto **cylinder** e a altura do objeto **cylinder** (**arrayOfShapes[2]**) são enviados para a saída; calculam-se a área de **cylinder** e o volume de **cylinder**. Todas as chamadas de método para **getName**, **toString**, **area** e **volume** são resolvidas durante a execução, com vinculação dinâmica.

9.19 Estudo de caso: criando e utilizando interfaces

Nosso próximo exemplo (Fig. 9.27 a Fig. 9.31) reexamina a hierarquia de `Point`, `Circle`, `Cylinder` uma última vez, substituindo a superclasse `abstract Shape` pela interface `Shape` (Fig. 9.27). Uma definição de interface inicia com a palavra-chave `interface` e contém um conjunto de métodos `public abstract`. As interfaces também podem conter dados `static final public`. Para utilizar uma interface, a classe deve especificar que ela implementa (`implements`) a interface e a classe deve definir cada método na interface com o número de argumentos e o tipo de retorno especificados na definição da interface. Se a classe deixar qualquer método da interface indefinido, a classe torna-se uma classe `abstract` e deve ser declarada `abstract` na primeira linha de sua definição de classe. Implementar uma interface é como assinar um contrato com o compilador que declara: “Definirei todos os métodos especificados pela interface”.



Erro comum de programação 9.9

Deixar um método de uma interface indefinido em uma classe que implementa (`implements`) a interface resulta em um erro de compilação indicando que a classe deve ser declarada `abstract`.



Observação de engenharia de software 9.30

Declarar uma referência `final` indica que a referência sempre se refere ao mesmo objeto. Entretanto, isso não afeta o objeto ao qual a referência se refere – os dados do objeto ainda podem ser modificados.

Começamos a utilizar o conceito de uma interface quando introduzimos o tratamento de evento GUI no Capítulo 6. Lembre-se de que nossa classe de *applet* incluiu `implements ActionListener` (uma interface no pacote `java.awt.event`). A razão pela qual somos obrigados a definir `actionPerformed` nos *applets* com tratamento de evento é que `ActionListener` é uma interface que especifica que `actionPerformed` deve ser definido. As interfaces são uma parte importante do tratamento de eventos de GUI, como discutiremos na próxima seção.

Geralmente se usa uma interface em lugar de uma classe `abstract` quando não há implementação *default* a herdar – isto é, não há variáveis de instância nem implementações de métodos *default*. Assim como as classes `public abstract`, as interfaces são em geral tipos de dados `public`, portanto elas normalmente são definidas em arquivos separados, com o mesmo nome que a interface e a extensão `.java`.

A definição da interface `Shape` inicia na linha 4 da Fig. 9.27. A interface `Shape` tem os métodos `abstract area`, `volume` e `getName`. Por coincidência, todos os três métodos não recebem argumentos. Entretanto, isso não é um requisito para métodos em uma interface.

Na Fig. 9.28, a linha 4 indica que a classe `Point` estende a classe `Object` e implementa a interface `Shape`. A classe `Point` fornece as definições de todos os três métodos na interface. O método `area` é definido nas linhas 45 a 48. O método `volume` é definido nas linhas 51 a 54. O método `getName` é definido nas linhas 57 a 60. Esses três métodos satisfazem o requisito de implementação dos três métodos definidos na interface. Cumprimos nosso contrato com o compilador.

```

1 // Fig. 9.27: Shape.java
2 // Definição da interface Shape
3
4 public interface Shape {
5
6     // calcula a área
7     public abstract double area();
8
9     // calcula o volume
10    public abstract double volume();
11
12    // devolve o nome da forma
13    public abstract String getName();
14 }
```

Fig. 9.27 Hierarquia ponto, círculo, cilindro com uma interface `Shape`.

```

1 // Fig. 9.28: Point.java
2 // Definição da classe Point
3
4 public class Point extends Object implements Shape {
5     protected int x, y;    // coordenadas do Point
6
7     // construtor sem argumentos
8     public Point()
9     {
10        setPoint( 0, 0 );
11    }
12
13    // construtor
14    public Point( int xCoordinate, int yCoordinate )
15    {
16        setPoint( xCoordinate, yCoordinate );
17    }
18
19    // Configura as coordenadas x e y do Point
20    public void setPoint( int xCoordinate, int yCoordinate )
21    {
22        x = xCoordinate;
23        y = yCoordinate;
24    }
25
26    // obtém a coordenada x
27    public int getX()
28    {
29        return x;
30    }
31
32    // obtém a coordenada y
33    public int getY()
34    {
35        return y;
36    }
37
38    // converte o ponto para representação de String
39    public String toString()
40    {
41        return "[" + x + ", " + y + "]";
42    }
43
44    // calcula a área
45    public double area()
46    {
47        return 0.0;
48    }
49
50    // calcula o volume
51    public double volume()
52    {
53        return 0.0;
54    }
55
56    // devolve o nome da forma
57    public String getName()
58    {
59        return "Point";
60    }

```

Fig. 9.28 Implementação de `Point` com a interface `Shape` (parte 1 de 2).

```

61
62 } // fim da classe Point

```

Fig. 9.28 Implementação de `Point` com a interface `Shape` (parte 2 de 2).

Quando uma classe implementa uma interface, aplica-se o mesmo relacionamento é *um* fornecido pela herança. Por exemplo, a classe `Point` implementa `Shape`. Portanto, o objeto `Point` é *um Shape*. De fato, os objetos de qualquer classe que estende `Point` também são objetos `Shape`. Utilizando esse relacionamento, mantivemos as definições originais da classe `Circle`, da classe `Cylinder` e da classe de aplicativo `Test` da Fig. 9.18 (repetidas nas Figs. 9.29 a 9.31) para ilustrar que se pode utilizar uma interface em vez de uma classe `abstract` para processar `Shapes` de maneira polimórfica. Repare que a saída do programa (Fig. 9.31) é idêntica à da Fig. 9.22. Além disso, repare que o método `toString` de `Object` é chamado por uma referência para a interface `Shape` (linha 44).



Observação de engenharia de software 9.31

Todos os métodos da classe `Object` podem ser chamados utilizando-se uma referência de um tipo de dados de interface – uma referência se refere a um objeto, e todos os objetos têm os métodos definidos pela classe `Object`.

Uma vantagem de se utilizar interfaces é que a classe pode implementar quantas interfaces ela precisar, além de estender uma classe. Para implementar mais de uma interface, forneça simplesmente uma lista de nomes de interface separados por vírgulas depois da palavra-chave `implements` na definição da classe. Isso é particularmente útil no mecanismo de tratamento de eventos de GUI. A classe que implementa mais de uma interface ouvinte de eventos (como `ActionListener` em exemplos anteriores) pode processar tipos diferentes de eventos de GUI, como veremos nos Capítulos 12 e 13.

```

1 // Fig. 9.29: Circle.java
2 // Definição da classe Circle
3
4 public class Circle extends Point { // herda de Point
5     protected double radius;
6
7     // construtor sem argumentos
8     public Circle()
9     {
10         // chamada implícita para o construtor da superclasse aqui
11         setRadius( 0 );
12     }
13
14     // construtor
15     public Circle( double circleRadius, int xCoordinate,
16                     int yCoordinate )
17     {
18         // chama o construtor da superclasse
19         super( xCoordinate, yCoordinate );
20
21         setRadius( circleRadius );
22     }
23
24     // configura o raio do Circle
25     public void setRadius( double circleRadius )
26     {
27         radius = ( circleRadius >= 0 ? circleRadius : 0 );
28     }
29
30     // obtém o raio do Circle
31     public double getRadius()

```

Fig. 9.29 Subclasse `Circle` de `Point` – implementação indireta da interface `Shape` (parte 1 de 2).

```

32     {
33         return radius;
34     }
35
36     // calcula a área do Circle
37     public double area()
38     {
39         return Math.PI * radius * radius;
40     }
41
42     // converte Circle para uma representação de String
43     public String toString()
44     {
45         return "Center = " + super.toString() +
46             "; Radius = " + radius;
47     }
48
49     // devolve o nome da forma
50     public String getName()
51     {
52         return "Circle";
53     }
54
55 } // fim da classe Circle

```

Fig. 9.29 Subclasse Circle de Point – implementação indireta da interface Shape (parte 2 de 2).

```

1 // Fig. 9.30: Cylinder.java
2 // Definição da classe Cylinder.
3
4 public class Cylinder extends Circle {
5     protected double height;    // altura do Cylinder
6
7     // construtor sem argumentos
8     public Cylinder()
9     {
10        // chamada implícita para o construtor da superclasse aqui
11        setHeight( 0 );
12    }
13
14    // construtor
15    public Cylinder( double cylinderHeight,
16                    double cylinderRadius, int xCoordinate,
17                    int yCoordinate )
18    {
19        // chama o construtor da superclasse
20        super( cylinderRadius, xCoordinate, yCoordinate );
21
22        setHeight( cylinderHeight );
23    }
24
25    // configura a altura do Cylinder
26    public void setHeight( double cylinderHeight )
27    {
28        height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
29    }
30

```

Fig. 9.30 Subclasse Cylinder de Circle – implementação indireta da interface Shape (parte 1 de 2).

```

31     // obtém a altura do Cylinder
32     public double getHeight()
33     {
34         return height;
35     }
36
37     // calcula a área do Cylinder (i.e., área da superfície do cilindro)
38     public double area()
39     {
40         return 2 * super.area() + 2 * Math.PI * radius * height;
41     }
42
43     // calcula o volume do Cylinder
44     public double volume()
45     {
46         return super.area() * height;
47     }
48
49     // converte o Cylinder para uma representação de String
50     public String toString()
51     {
52         return super.toString() + "; Height = " + height;
53     }
54
55     // devolve o nome da forma
56     public String getName()
57     {
58         return "Cylinder";
59     }
60
61 } // fim da classe Cylinder

```

Fig. 9.30 Subclasse Cylinder de Circle – implementação indireta da interface Shape (parte 2 de 2).

```

1 // Fig. 9.31: Test.java
2 // Testa a hierarquia Point, Circle, Cylinder com a interface Shape
3
4 // Pacotes do núcleo de Java
5 import java.text.DecimalFormat;
6
7 // Pacotes de extensão de Java
8 import javax.swing.JOptionPane;
9
10 public class Test {
11
12     // testa a hierarquia de formas
13     public static void main( String args[] )
14     {
15         // cria formas
16         Point point = new Point( 7, 11 );
17         Circle circle = new Circle( 3.5, 22, 8 );
18         Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );
19
20         // cria array de Shapes
21         Shape arrayOfShapes[] = new Shape[ 3 ];
22

```

Fig. 9.31 Hierarquia Shape, Point, Circle, Cylinder (parte 1 de 2).

```

23     // aponta arrayOfShapes[ 0 ] para o objeto da subclasse Point
24     arrayOfShapes[ 0 ] = point;
25
26     // aponta arrayOfShapes[ 1 ] para o objeto da subclasse Circle
27     arrayOfShapes[ 1 ] = circle;
28
29     // aponta arrayOfShapes[ 2 ] para o objeto da subclasse Cylinder
30     arrayOfShapes[ 2 ] = cylinder;
31
32     // obtém o nome e a representação como String de cada forma
33     String output =
34         point.getName() + ":" + point.toString() + "\n" +
35         circle.getName() + ":" + circle.toString() + "\n" +
36         cylinder.getName() + ":" + cylinder.toString();
37
38     DecimalFormat precision2 = new DecimalFormat( "0.00" );
39
40     // percorre com um laço arrayOfShapes e obtém nome,
41     // área e volume de cada forma em arrayOfShapes
42     for ( int i = 0; i < arrayOfShapes.length; i++ ) {
43         output += "\n\n" + arrayOfShapes[ i ].getName() +
44             ":" + arrayOfShapes[ i ].toString() +
45             "\nArea = " +
46             precision2.format( arrayOfShapes[ i ].area() ) +
47             "\nVolume = " +
48             precision2.format( arrayOfShapes[ i ].volume() );
49     }
50
51     JOptionPane.showMessageDialog( null, output,
52         "Demonstrating Polymorphism",
53         JOptionPane.INFORMATION_MESSAGE );
54
55     System.exit( 0 );
56 }
57
58 } // fim da classe Test

```

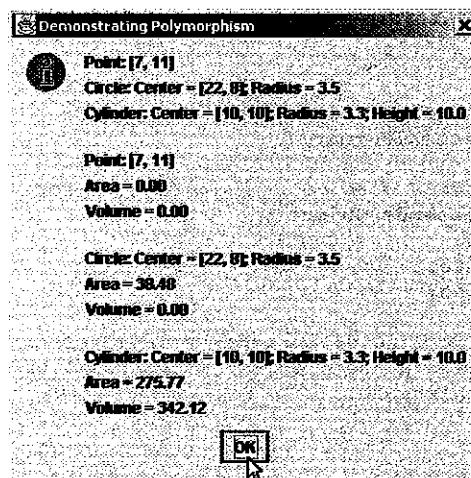


Fig. 9.31 Hierarquia Shape, Point, Circle, Cylinder (parte 2 de 2).

Outra utilização de interfaces é para definir um conjunto de constantes que podem ser utilizadas em muitas definições de classe. Considere a interface **Constants**

```

public interface Constants {
    public static final int ONE = 1;
    public static final int TWO = 2;
    public static final int THREE = 3;
}

```

As classes que implementam a interface `Constants` podem utilizar as constantes `ONE`, `TWO` e `THREE` em qualquer lugar na definição de classe. A classe pode até utilizar essas constantes simplesmente importando a interface e depois fazendo referência a cada constante como `Constants.ONE`, `Constants.TWO` e `Constants.THREE`. Como não há métodos declarados nessa interface, a classe que implementa a interface não é obrigada a fornecer qualquer implementação.

9.20 Definições de classe interna

Todas as definições de classe discutidas até agora foram definidas no escopo de arquivo. Por exemplo, se um arquivo continha duas classes, uma classe não estava aninhada no corpo da outra classe. Java oferece um recurso chamado de *classes internas* em que as classes podem ser definidas dentro de outras classes. Tais classes podem ser definições completas de classe ou definições de *classe interna anônima* (classes sem nome). As classes internas são utilizadas principalmente em tratamento de eventos. Entretanto, elas oferecem outras vantagens. Por exemplo, a implementação do tipo abstrato de dados de fila discutida na Seção 8.16.1 poderia utilizar uma classe interna para representar os objetos que armazenam cada item atualmente na fila. Só a estrutura de dados de fila exige conhecimento de como os objetos são armazenados internamente, de modo que se possa ocultar a implementação definindo-se uma classe interna como parte da classe `Queue`.

Como as classes internas são freqüentemente utilizadas com tratamento de eventos de GUI, aproveitamos essa oportunidade não apenas para lhe mostrar as definições de classe interna, mas também para demonstrar um aplicativo que é executado em sua própria janela. Depois de completar esse exemplo, você será capaz de utilizar em seus aplicativos as técnicas de GUI mostradas até agora apenas em *applets*.

Para demonstrar uma definição de classe interna, a Fig. 9.33 utiliza uma versão simplificada da classe `Time3` (renomeada `Time` na Fig. 9.32) da Fig. 8.8. A classe `Time` fornece um construtor *default*, os mesmos métodos *set/get* que os da Fig. 8.8 e um método `toString`. Além disso, esse programa define a classe `TimeTestWindow` como um aplicativo. O aplicativo é executado em sua própria janela. [Nota: não discutimos a classe `Time` aqui, porque todas as suas características foram discutidas no Capítulo 8.]

```

1 // Fig. 9.32: Time.java
2 // Definição da classe Time.
3
4 // Pacotes do núcleo de Java
5 import java.text.DecimalFormat;
6
7 // Esta classe mantém a hora no formato de 24 horas
8 public class Time extends Object {
9     private int hour;      // 0 a 23
10    private int minute;    // 0 a 59
11    private int second;    // 0 a 59
12
13    // Construtor Time inicializa cada variável de instância
14    // com zero. Assegura que o objeto Time inicia em
15    // um estado consistente.
16    public Time()
17    {
18        setTime( 0, 0, 0 );
19    }
20
21    // Configura um novo valor de hora usando hora universal. Faz a

```

Fig. 9.32 Classe `Time` (parte 1 de 2).

```

22  // verificação de validade da data. Configura valores inválidos como zero.
23  public void setTime( int hour, int minute, int second )
24  {
25      setHour( hour );
26      setMinute( minute );
27      setSecond( second );
28  }
29
30  // valida e configura a hora
31  public void setHour( int h )
32  {
33      hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
34  }
35
36  // valida e configura o minuto
37  public void setMinute( int m )
38  {
39      minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
40  }
41
42  // valida e configura o segundo
43  public void setSecond( int s )
44  {
45      second = ( ( s >= 0 && s < 60 ) ? s : 0 );
46  }
47
48  // obtém a hora
49  public int getHour()
50  {
51      return hour;
52  }
53
54  // obtém o minuto
55  public int getMinute()
56  {
57      return minute;
58  }
59
60  // obtém o segundo
61  public int getSecond()
62  {
63      return second;
64  }
65
66  // converte para String no formato de hora padrão
67  public String toString()
68  {
69      DecimalFormat twoDigits = new DecimalFormat( "00" );
70
71      return ( ( getHour() == 12 || getHour() == 0 ) ?
72          12 : getHour() % 12 ) + ":" +
73          twoDigits.format( getMinute() ) + ":" +
74          twoDigits.format( getSecond() ) +
75          ( getHour() < 12 ? " AM" : " PM" );
76  }
77
78 } // fim da classe Time

```

Fig. 9.32 Classe Time (parte 2 de 2).

```
1 // Fig. 9.33: TimeTestWindow.java
2 // Demonstrando os métodos set e get da classe Time
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class TimeTestWindow extends JFrame {
12     private Time time;
13     private JLabel hourLabel, minuteLabel, secondLabel;
14     private JTextField hourField, minuteField,
15         secondField, displayField;
16     private JButton exitButton;
17
18     // configura a GUI
19     public TimeTestWindow()
20     {
21         super( "Inner Class Demonstration" );
22
23         time = new Time();
24
25         // cria uma instância da classe interna ActionEventHandler
26         ActionEventHandler handler = new ActionEventHandler();
27
28         // configura a GUI
29         Container container = getContentPane();
30         container.setLayout( new FlowLayout() );
31
32         hourLabel = new JLabel( "Set Hour" );
33         hourField = new JTextField( 10 );
34         hourField.addActionListener( handler );
35         container.add( hourLabel );
36         container.add( hourField );
37
38         minuteLabel = new JLabel( "Set minute" );
39         minuteField = new JTextField( 10 );
40         minuteField.addActionListener( handler );
41         container.add( minuteLabel );
42         container.add( minuteField );
43
44         secondLabel = new JLabel( "Set Second" );
45         secondField = new JTextField( 10 );
46         secondField.addActionListener( handler );
47         container.add( secondLabel );
48         container.add( secondField );
49
50         displayField = new JTextField( 30 );
51         displayField.setEditable( false );
52         container.add( displayField );
53
54         exitButton = new JButton( "Exit" );
55         exitButton.addActionListener( handler );
56         container.add( exitButton );
57 }
```

Fig. 9.33 Demonstrando uma classe interna em um aplicativo com janela (parte 1 de 3).

```

58     } // fim do construtor
59
60     // exibe a hora no displayField
61     public void displayTime()
62     {
63         displayField.setText( "The time is: " + time );
64     }
65
66     // cria TimeTestWindow e a exibe
67     public static void main( String args[] )
68     {
69         TimeTestWindow window = new TimeTestWindow();
70
71         window.setSize( 400, 140 );
72         window.setVisible( true );
73     }
74
75     // definição de classe interna para tratar
76     // de eventos de JTextField e JButton
77     private class ActionEventHandler
78         implements ActionListener {
79
80             // método que trata de eventos de ação
81             public void actionPerformed( ActionEvent event )
82             {
83                 // usuário pressionou exitButton
84                 if ( event.getSource() == exitButton )
85                     System.exit( 0 ); // termina o aplicativo
86
87                 // usuário pressionou a tecla Enter dentro de hourField
88                 else if ( event.getSource() == hourField ) {
89                     time.setHour(
90                         Integer.parseInt( event.getActionCommand() ) );
91                     hourField.setText( "" );
92                 }
93
94                 // usuário pressionou a tecla Enter dentro de minuteField
95                 else if ( event.getSource() == minuteField ) {
96                     time.setMinute(
97                         Integer.parseInt( event.getActionCommand() ) );
98                     minuteField.setText( "" );
99                 }
100
101                // usuário pressionou a tecla Enter dentro de secondField
102                else if ( event.getSource() == secondField ) {
103                    time.setSecond(
104                        Integer.parseInt( event.getActionCommand() ) );
105                    secondField.setText( "" );
106                }
107
108                displayTime();
109            }
110
111        } // fim da classe interna ActionEventHandler
112    } // fim da classe TimeTestWindow

```

Fig. 9.33 Demonstrando uma classe interna em um aplicativo com janela (parte 2 de 3).

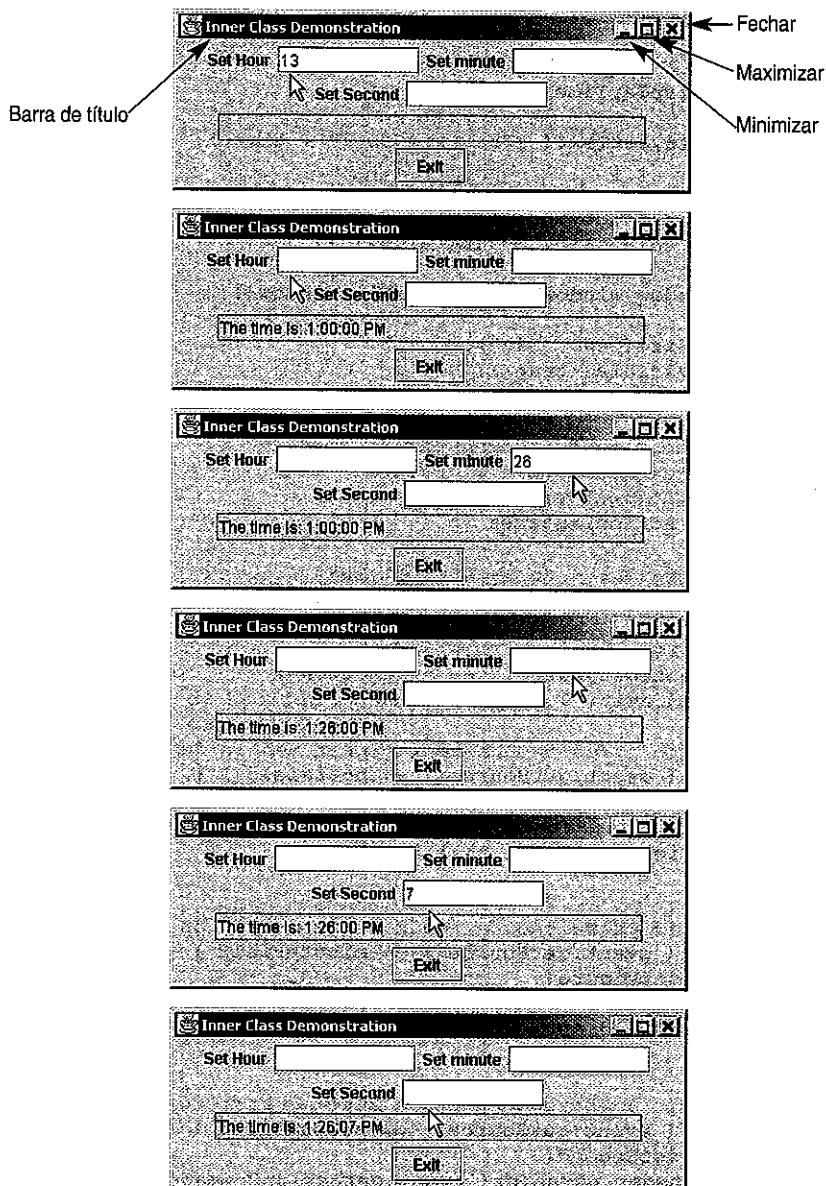


Fig. 9.33 Demonstrando uma classe interna em um aplicativo com janela (parte 3 de 3).

Na Fig. 9.33, a linha 11 indica que a classe `TimeTestWindow` estende a classe `JFrame` (do pacote `javax.swing`) em vez da classe `JApplet` (como mostrado na Fig. 8.8). A superclasse `JFrame` fornece os atributos e comportamentos básicos de uma janela – uma *barra de título* e botões para *minimizar*, *maximizar* e *fechar* a janela (todos rotulados na primeira captura de tela). A classe `TimeTestWindow` utiliza os mesmos componentes GUI que o *applet* da Fig. 8.8, exceto pelo fato de que o botão (linha 16) agora é chamado de `exitButton` e é utilizado para terminar o aplicativo.

O método `init` do *applet* foi substituído por um construtor (linhas 19 a 54) para garantir que os componentes GUI da janela sejam criados quando o aplicativo começar a ser executado. O método `main` (linhas 67 a 73) define um objeto `new` da classe `TimeTestWindow`, o que resulta em uma chamada para o construtor. Lembre-se, `init` é um método especial que é garantidamente chamado quando o *applet* começa a ser executado. Entretanto, esse programa não é um *applet*, de modo que, se tivéssemos definido o método `init` ele não seria chamado automaticamente.

Vários recursos novos aparecem no construtor. A linha 21 chama o construtor de superclasse `JFrame` com o string "Inner Class Demonstration". Esse string é exibido na barra de título da janela pelo construtor da classe `JFrame`. A linha 26 define uma instância de nossa classe interna `ActionEventHandler` (definida nas linhas 77 a 111) e a atribui a `handler`. Essa referência é passada para cada uma das quatro chamadas para `addActionListener` (linhas 34, 40, 46 e 55) que registra os tratadores de eventos para cada componente GUI que gera eventos nesse exemplo (`hourField`, `minuteField`, `secondField` e `exitButton`). Cada chamada a `addActionListener` exige que um objeto de tipo `ActionListener` seja passado como argumento. Na verdade, `handler` é um `ActionListener`. A linha 77 (a primeira linha da definição da classe interna) indica que a classe interna `ActionEventHandler` implementa `ActionListener`. Portanto, cada objeto do tipo `ActionEventHandler` é um `ActionListener`. O requisito de que `addActionListener` seja passado como objeto de tipo `ActionListener` é satisfeito! Utiliza-se o relacionamento é um extensamente no mecanismo de tratamento de eventos GUI, como você verá nos próximos capítulos. A classe interna é definida como `private` porque será utilizada somente nessa definição de classe. As classes internas podem ser `private`, `protected` ou `public`.

O objeto de classe interna tem um relacionamento especial com o objeto de classe externa que o cria. É permitido ao objeto de classe interna acessar diretamente todas as variáveis de instância e os métodos do objeto de classe externa. O método `actionPerformed` (linhas 77 a 105) da classe `ActionEventHandler` faz exatamente isso. Ao longo de todo o método, são utilizadas as variáveis de instância `time`, `exitButton`, `hourField`, `minuteField` e `secondField`, assim como o método `displayTime`. Repare que nenhuma delas precisa de um "handle" para o objeto de classe externa. Esse é um relacionamento livre criado pelo compilador entre a classe externa e suas classes internas.



Observação de engenharia de software 9.32

O objeto de classe interna tem permissão para acessar diretamente todas as variáveis de instância e os métodos do objeto da classe externa que o definiu.

Esse aplicativo deve ser terminado pressionando-se o botão `Exit`. Lembre-se, o aplicativo que exibe uma janela deve ser terminado com uma chamada a `System.exit(0)`. Também observe que, por default, a janela em Java tem 0 pixels de largura, 0 pixels de altura e não é exibida. As linhas 71 e 72 usam os métodos `resize` e `setVisible` para dimensionar a janela e exibi-la na tela. Estes métodos estão definidos originalmente na classe `java.awt.Component` e são herdados para a classe `JFrame`.

Uma classe interna também pode ser definida dentro de um método de uma classe. Essa classe interna tem acesso aos membros de sua classe externa. Entretanto, ela tem acesso limitado às variáveis locais do método em que é definida.



Observação de engenharia de software 9.33

Permite-se que uma classe interna definida em um método acesse diretamente todas as variáveis de instância e métodos do objeto da classe externa que a definiu e quaisquer variáveis locais final no método.

O aplicativo da Fig. 9.34 modifica a classe `TimeTestWindow` da Fig. 9.33 para utilizar *classes internas anônimas* definidas em métodos. Como uma classe interna anônima não tem nome, deve-se criar um objeto da classe interna anônima no ponto em que se define a classe no programa. Demonstramos as classes internas anônimas de duas maneiras nesse exemplo. Primeiro, utilizamos as classes internas anônimas separadas que implementam uma interface (`ActionListener`) para criar tratadores de eventos para cada um dos três `JTextFields`: `hourField`, `minuteField` e `secondField`. Também demonstramos como terminar um aplicativo quando o usuário clica no botão de fechamento na janela. O tratador de evento é definido como uma classe interna anônima que estende uma classe (`WindowAdapter`). A classe `Time` utilizada é idêntica à da Fig. 9.32. Por isso, não foi incluída aqui. Além disso, o botão `Exit` foi removido desse exemplo.

```

1 // Fig. 9.34: TimeTestWindow.java
2 // Demonstrando os métodos set e get da classe Time
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class TimeTestWindow extends JFrame {
12     private Time time;
13     private JLabel hourLabel, minuteLabel, secondLabel;
14     private JTextField hourField, minuteField,
15         secondField, displayField;
16
17     // configura a GUI
18     public TimeTestWindow()
19     {
20         super( "Inner Class Demonstration" );
21
22         // cria um objeto Time
23         time = new Time();
24
25         // cria a GUI
26         Container container = getContentPane();
27         container.setLayout( new FlowLayout() );
28
29         hourLabel = new JLabel( "Set Hour" );
30         hourField = new JTextField( 10 );
31
32         // registra tratador de eventos de hourField
33         hourField.addActionListener(
34
35             // classe interna anônima
36             new ActionListener() {
37
38                 public void actionPerformed( ActionEvent event )
39                 {
40                     time.setHour(
41                         Integer.parseInt( event.getActionCommand() ) );
42                     hourField.setText( "" );
43                     displayTime();
44                 }
45
46             } // fim da classe interna anônima
47
48         ); // fim da chamada para addActionListener
49
50         container.add( hourLabel );
51         container.add( hourField );
52
53         minuteLabel = new JLabel( "Set minute" );
54         minuteField = new JTextField( 10 );
55
56         // registra tratador de eventos de minuteField
57         minuteField.addActionListener(
58
59             // classe interna anônima
60             new ActionListener() {

```

Fig. 9.34 Demonstrando classes internas anônimas (parte 1 de 3).

```

61         public void actionPerformed( ActionEvent event )
62     {
63         time.setMinute(
64             Integer.parseInt( event.getActionCommand() ) );
65         minuteField.setText( "" );
66         displayTime();
67     }
68 }
69 }
70 } // fim da classe interna anônima
71 );
72 // fim da chamada para addActionListener
73
74 container.add( minuteLabel );
75 container.add( minuteField );
76
77 secondLabel = new JLabel( "Set Second" );
78 secondField = new JTextField( 10 );
79
80 secondField.addActionListener(
81
82     // classe interna anônima
83     new ActionListener() {
84
85         public void actionPerformed( ActionEvent event )
86     {
87         time.setSecond(
88             Integer.parseInt( event.getActionCommand() ) );
89         secondField.setText( "" );
90         displayTime();
91     }
92 }
93 } // fim da classe interna anônima
94 );
95 // fim da chamada para addActionListener
96
97 container.add( secondLabel );
98 container.add( secondField );
99
100 displayField = new JTextField( 30 );
101 displayField.setEditable( false );
102 container.add( displayField );
103 }
104
105 // exibe a hora em displayField
106 public void displayTime()
107 {
108     displayField.setText( "The time is: " + time );
109 }
110
111 // cria TimeTestWindow, registra seus eventos de janela
112 // e a exibe para iniciar a execução do aplicativo
113 public static void main( String args[] )
114 {
115     TimeTestWindow window = new TimeTestWindow();
116
117     // registra tratador para o evento windowClosing
118     window.addWindowListener(
119

```

Fig. 9.34 Demonstrando classes internas anônimas (parte 2 de 3).

```

120     // classe interna anônima para o evento windowClosing
121     new WindowAdapter() {
122
123         // termina o aplicativo quando o usuário fecha a janela
124         public void windowClosing( WindowEvent event )
125         {
126             System.exit( 0 );
127         }
128
129     } // fim da classe interna anônima
130
131 }; // fim da chamada para addWindowListener
132
133 window.setSize( 400, 120 );
134 window.setVisible( true );
135 }
136
137 } // fim da classe TimeTestWindow

```

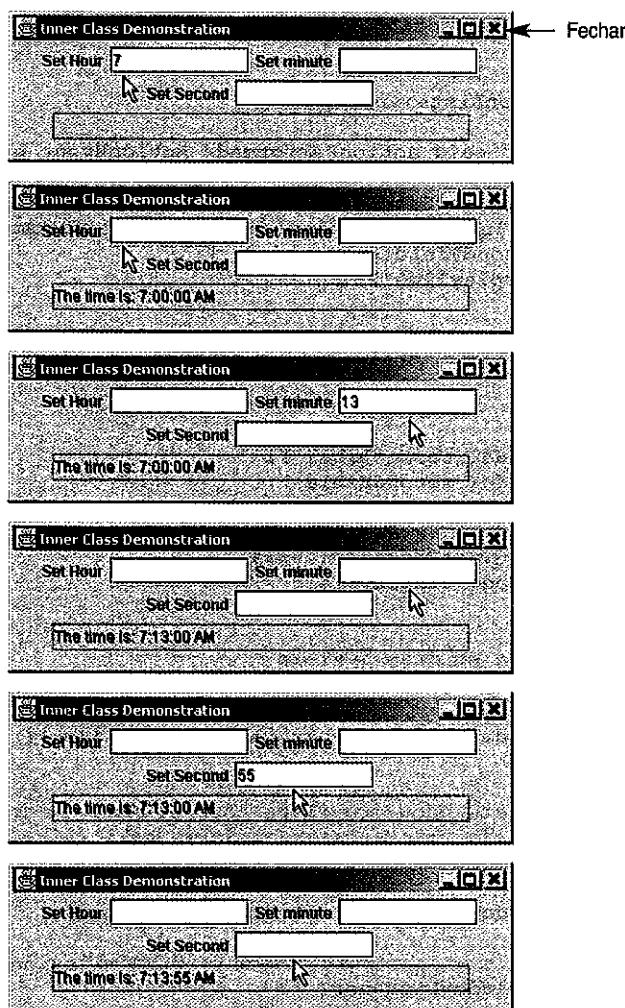


Fig. 9.34 Demonstrando classes internas anônimas (parte 3 de 3).

Cada um dos três `JTextFields` que gera eventos nesse programa tem uma classe interna anônima semelhante para tratar seus eventos, de modo que discutimos aqui apenas a classe interna anônima para `hourField`. As linhas 33 a 48 são uma chamada para o método `addActionListener` de `hourField`. O argumento para esse método deve ser um objeto que é *um ActionListener* (isto é, qualquer objeto de uma classe que implementa `ActionListener`). As linhas 36 a 46 utilizam sintaxe especial de Java para definir uma classe interna anônima e criar um objeto dessa classe que é passado como argumento para `addActionListener`. A linha 36 utiliza o operador `new` para criar um objeto. A sintaxe `ActionListener()` começa a definição de uma classe interna anônima que implementa a interface `ActionListener`. Isso é semelhante a começar uma definição de classe com

```
public class MyHandler implements ActionListener {
```

Os parênteses depois de `ActionListener` indicam uma chamada ao construtor *default* da classe interna anônima.

A chave esquerda de abertura (`{`) no fim da linha 36 e a chave direita de fechamento (`}`) na linha 46 definem o corpo da classe. As linhas 38 a 44 definem o método `actionPerformed` que é exigido em qualquer classe que implementa `ActionListener`. O método `actionPerformed` é chamado quando o usuário pressiona *Enter* enquanto está digitando em `hourField`.



Observação de engenharia de software 9.34

Quando uma classe interna anônima implementa uma interface, a classe deve definir cada método na interface.

O método `main` cria uma instância da classe `TimeTestWindow` (linha 115), dimensiona a janela (linha 133) e a exibe (linha 134).

As janelas geram diversos eventos que serão discutidos no Capítulo 13. Para esse exemplo, discutimos o evento gerado quando o usuário clica no botão de fechamento da janela – um *evento de fechamento de janela*. As linhas 118 a 131 permitem que o usuário encerre o aplicativo clicando no botão de fechamento da janela (rotulado na primeira captura de tela). O método `addWindowListener` registra um ouvinte de evento de janela. O argumento para `addWindowListener` deve ser uma referência a um objeto que é *um WindowListener* (pacote `java.awt.event`) (isto é, qualquer objeto de uma classe que implementa `WindowListener`). Entretanto, há sete métodos diferentes que devem ser definidos em toda a classe que implementa `WindowListener` e precisamos apenas de um nesse exemplo – `windowClosing`. Para interfaces de tratamento de eventos com mais de um método, Java fornece uma classe correspondente (chamada de *classe adaptadora*) que já implementa todos os métodos da interface para você. Tudo que você precisa fazer é estender a classe adaptadora e sobrescrever os métodos de que você precisa em seu programa.



Erro comum de programação 9.10

Estender uma classe adaptadora e cometer erros de ortografia no nome do método que você está sobrepondo é um erro de lógica.

As linhas 121 a 129 utilizam sintaxe especial de Java para definir uma classe interna anônima e criar um objeto dessa classe que é passado como argumento para `addWindowListener`. A linha 118 utiliza o operador `new` para criar um objeto. A sintaxe `WindowAdapter()` inicia a definição de uma classe interna anônima que estende a classe `WindowAdapter`. Isso é semelhante a começar uma definição de classe com

```
public class MyHandler extends WindowAdapter {
```

Os parênteses depois de `WindowAdapter` indicam uma chamada para o construtor *default* da classe interna anônima. A classe `WindowAdapter` implementa a interface `WindowListener`, de modo que cada objeto `WindowAdapter` é *um WindowListener* – o tipo exato exigido para o argumento de `addWindowListener`.

A chave esquerda de abertura (`{`) no fim da linha 121 e a chave direita de fechamento (`}`) na linha 129 definem o corpo da classe. As linhas 124 a 127 sobreponem o método `windowClosing` de `WindowAdapter` que é chamado quando o usuário clica no botão de fechamento da janela. Nesse exemplo, `windowClosing` encerra o aplicativo.

Nos últimos dois exemplos, vimos que se podem utilizar classes internas para criar tratadores de eventos e que se podem definir classes internas anônimas separadas para tratar eventos individualmente para cada compo-

nente GUI. Nos Capítulos 12 e 13, revisitaremos esse conceito ao discutir o mecanismo de tratamento de eventos em detalhes.

9.21 Notas sobre definições de classe interna

Esta seção apresenta várias notas de interesse para os programadores relacionadas à definição e utilização de classes internas.

1. Compilar uma classe que contém classes internas resulta em um arquivo `.class` separado para cada classe. As classes internas com nomes têm o nome de arquivo `NomeDaClasseExterna$NomeDaClasseInterna.class`. As classes internas anônimas têm o nome de arquivo `NomeDaClasseExterna$#.class`, onde # inicia em 1 e é incrementado para cada classe interna anônima encontrada durante a compilação.
2. As classes internas com nomes de classe podem ser definidas como `public`, `protected`, com acesso de pacote ou `private`, e estão sujeitas às mesmas restrições de uso que outros membros de uma classe.
3. Para acessar a referência `this` da classe externa, utilize `NomeDaClasseExterna.this`.
4. A classe externa é responsável por criar objetos de suas classes internas. Para criar um objeto de classe interna de outra classe, primeiro crie um objeto da classe externa e atribua-o a uma referência (que chamaremos de `ref`). Depois utilize uma instrução na seguinte forma para criar um objeto da classe interna:

```
NomeDaClasseExterna.NomeDaClasseInterna innerRef = ref.new NomeDaClasseInterna();
```

5. Uma classe interna pode ser declarada `static`. A classe interna `static` não exige que um objeto de sua classe externa seja definido (ao passo que uma classe não `static` interna, sim). A classe interna `static` não tem acesso a membros não-`static` da classe externa.

9.22 Classes invólucro de tipo para tipos primitivos

Cada um dos tipos primitivos tem uma classe invólucro de tipo. Essas classes se chamam `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` e `Boolean`. Cada classe invólucro de tipo permite manipular tipos primitivos como objetos da classe `Object`. Portanto, valores dos tipos primitivos de dados podem ser processados de maneira polimórfica se forem mantidos como objetos das classes invólucro de tipo. Muitas das classes que desenvolveremos ou reutilizaremos manipulam e compartilham `Objects`. Essas classes não podem manipular de maneira polimórfica as variáveis de tipos primitivos, mas podem manipular de maneira polimórfica objetos das classes invólucro de tipo, porque cada classe é, em última instância, derivada da classe `Object`.

Cada uma das classes numéricas – `Byte`, `Short`, `Integer`, `Long`, `Float` e `Double` – herda da classe `Number`. Cada um dos invólucros de tipo é declarado `final`, de modo que seus métodos são implicitamente `final` e não podem ser sobreescritos. Observe que muitos dos métodos que processam os tipos primitivos de dados são definidos como métodos `static` das classes invólucro de tipo. Se você precisar manipular um valor primitivo em seu programa, primeiro consulte a documentação das classes invólucro de tipo – o método de que você precisa pode já estar definido. Utilizaremos as classes invólucro de tipo de maneira polimórfica em nosso estudo de estruturas de dados nos Capítulos 22 e 23.

9.23 (Estudo de caso opcional) Pensando em objetos: incorporando herança à simulação do elevador

Agora examinamos nosso projeto de simulador de elevador para ver se ele pode se beneficiar da herança. Nos capítulos anteriores, tratamos `ElevatorButton` e `FloorButton` como classes separadas. Estas classes, no entanto, têm muito em comum – ambas têm o atributo `pressed` e os comportamentos `pressButton` e `resetButton`. Para aplicar herança, primeiro procuramos as coisas em comum entre estas classes. Extraímos o que há em comum, colocamos numa superclasse `Button` e depois derivamos subclasses `ElevatorButton` e `FloorButton` a partir de `Button`.

Examinemos agora as similaridades entre as classes `ElevatorButton` e `FloorButton`. A Fig. 9.35 mostra os atributos e as operações de cada classe. As duas classes têm seu atributo (`pressed`) e operações (`pressButton` e `resetButton`) em comum.

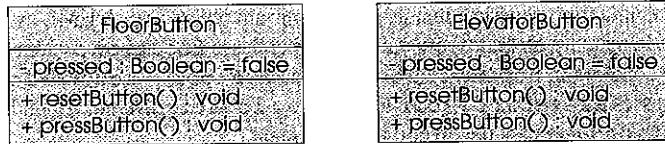


Fig. 9.35 Atributos e operações das classes `FloorButton` e `ElevatorButton`.

Precisamos agora analisar se estes objetos exibem comportamento distinto. Se os objetos `ElevatorButton` e `FloorButton` tiverem o mesmo comportamento, não podemos justificar o uso de duas classes separadas para instanciar estes objetos. entretanto, se estes objetos tiverem comportamentos distintos, podemos colocar os atributos e as operações comuns em uma superclasse `Button`; depois, `ElevatorButton` e `FloorButton` herdam tanto os atributos quanto as operações de `Button`.

Tratamos o `FloorButton` como se ele se comportasse de maneira diferente do `ElevatorButton` – o `FloorButton` pede ao `Elevator` para se movimentar para o `Floor` do pedido e o `ElevatorButton` sinaliza para o `Elevator` se mover para o `Floor` oposto. Examinando mais de perto, observamos que tanto o `FloorButton` quanto o `ElevatorButton` sinalizam para o `Elevator` se mover para um `Floor` e o `Elevator` decide se deve ou não se mover. O `Elevator` algumas vezes vai se mover em resposta a um sinal de `FloorButton` e o `Elevator` vai sempre se mover em resposta a um sinal do `ElevatorButton` – entretanto, nem o `FloorButton` nem o `ElevatorButton` decidem pelo `Elevator` que o `Elevator` deve se mover para o outro `Floor`. Concluímos que tanto o `FloorButton` quanto o `ElevatorButton` têm o mesmo comportamento – ambos sinalizam ao `Elevator` para se mover – e portanto *combinamos* (não herdamos) as classes para formar uma classe `Button` e descartamos as classes `FloorButton` e `ElevatorButton` de nosso estudo de caso.

Voltamos nossa atenção para as classes `ElevatorDoor` e `FloorDoor`. A Fig. 9.36 mostra os atributos e as operações das classes `ElevatorDoor` e `FloorDoor` – estas duas classes também são estruturalmente semelhantes uma à outra, porque as duas classes possuem o atributo `open` e as duas operações `openDoor` e `closeDoor`. Além disso, tanto a `ElevatorDoor` quanto a `FloorDoor` têm o mesmo comportamento – elas informam a uma `Person` que uma porta na simulação se abriu. A `Person` então decide se entra no `Elevator`, ou sai dele, dependendo de qual porta se abriu. Em outras palavras, nem a `ElevatorDoor` nem a `FloorDoor` decide pela `Per-`

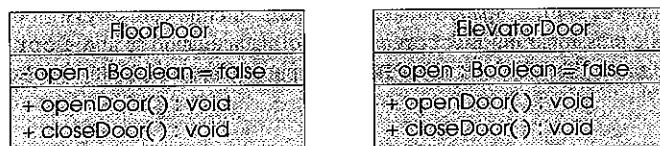


Fig. 9.36 Atributos e operações das classes `FloorDoor` e `ElevatorDoor`.

son que a **Person** deve entrar no **Elevator** ou sair dele. Combinamos as classes **ElevatorDoor** e **FloorDoor** em uma classe **Door** e eliminamos as classes **ElevatorDoor** e **FloorDoor** de nosso estudo de caso².

Na Seção 4.14, encontramos o problema de se representar a posição da **Person** – em que **Floor** a **Person** está localizada quando anda no **Elevator**? Usando herança, podemos agora modelar uma solução. Tanto o **Elevator** quanto os dois **Floors** são lugares nos quais a **Person** existe no simulador. Em outras palavras, o **Elevator** e os **Floors** são *tipos* de lugares. Portanto, as classes **Elevator** e **Floor** devem herdar de uma superclasse abstrata chamada **Location**. Declaramos a classe **Location** como abstrata, porque, para os fins de nossa simulação, um lugar é um termo genérico demais para definir um objeto real. A UML exige que coloquemos nomes de classes abstratas (e métodos abstratos) em itálico. A superclasse **Location** contém o atributo **protected locationName**, que contém um **String** com o valor “**firstFloor**”, “**secondFloor**” ou “**elevator**”. Portanto, somente as classes **Elevator** e **Floor** têm acesso a **locationName**. Além disso, incluímos o método **public getLocationName** para devolver o nome do lugar.

Procuramos mais semelhanças entre as classes **Floor** e **Elevator**. Antes de tudo, de acordo com o diagrama de classes da Fig. 3.23, **Elevator** contém uma referência tanto para um **Button** quanto para uma **Door** – o **ElevatorButton** (o **Button** do **Elevator**) e a **ElevatorDoor** (a **Door** do **Elevator**). A classe **Floor**, através de sua associação com a classe **ElevatorShaft** (a classe **ElevatorShaft** “conecta” a classe **Floor**), também contém uma referência para um **Button** e uma **Door** – o **FloorButton** (o **Button** daquele **Floor**) e a **FloorDoor** (a **Door** daquele **Floor**). Portanto, em nossa simulação, a classe **Location**, que é a superclasse das classes **Elevator** e **Floor**, irá conter os métodos **public getButton** e **getDoor**, que devolvem uma referência **Button** ou **Door**, respectivamente. A classe **Floor** sobrescreve estes métodos para devolver as referências para o **Button** e a **Door** daquele **Floor**, e a classe **Elevator** sobrescreve estes métodos para devolver as referências para o **Button** e a **Door** do **Elevator**. Em outras palavras, as classes **Floor** e **Elevator** exibem comportamentos distintos uma da outra, mas compartilham o atributo **locationName** e os métodos **getButton** e **getDoor** – portanto, podemos usar herança para estas classes.

A UML oferece um relacionamento chamado *generalização* para modelar herança. A Fig. 9.35 é o diagrama de generalização da superclasse **Location** e das subclasses **Elevator** e **Floor**. As setas com pontas vazadas especificam que as classes **Elevator** e **Floor** herdam da classe **Location**. Note que o atributo **locationName** tem um *modificador de acesso* que ainda não tínhamos visto – o sinal de sustenido (#), indicando que o **locationName** é um membro **protected**, de modo que as subclasses **Location** podem acessar este atributo. Note que as classes **Floor** e **Elevator** contêm atributos e métodos adicionais que diferenciam ainda mais estas classes.

As classes **Floor** e **Elevator** sobrescrevem os métodos **getButton** e **getDoor** de sua superclasse **Location**. Na Fig. 9.37, estes métodos estão em itálico na classe **Location**, indicando que eles são *métodos abstratos*. Entretanto, os métodos nas subclasses não estão em itálico – estes métodos se tornaram *métodos concretos* sobrepondo os métodos abstratos. A classe **Person** agora contém um objeto **Location** que representa se a **Person** está no primeiro ou segundo **Floor** ou dentro do **Elevator**. Removemos a associação entre **Person** e **Elevator** e a associação entre **Person** e **Floor** do diagrama de classes, porque o objeto **Location** age como a referência, tanto do **Elevator** quanto do **Floor**, para a **Person**. Uma **Person** configura seu objeto **Location** para fazer referência ao **Elevator** quando aquela **Person** entra no **Elevator**. Uma **Person** configura seu objeto **Location** para fazer referência a um **Floor** quando a **Person** está naquele **Floor**. Por último, atribuímos à classe **Elevator** dois objetos **Location** para representar as referências do **Elevator** para o **Floor** atual e o **Floor** de destino (originalmente usamos inteiros para descrever estas referências). A Fig. 9.38 mostra um diagrama de classes atualizado de nosso modelo, incorporando herança e eliminando as classes **FloorButton**, **ElevatorButton**, **FloorDoor** e **ElevatorDoor**.

Permitimos que uma **Person**, ocupando uma **Location**, interaja com diversos objetos no simulador. Por exemplo, a **Person** pode pressionar um **Button** ou ser informada quando se abre **Door** a partir da **Location** específica daquela **Person**. Além disso, como a **Person** pode ocupar somente uma **Location** de cada vez,

² À medida que continuamos a discutir herança ao longo dessa seção, referimo-nos ao diagrama de classes da Fig. 3.23 para determinar semelhanças entre objetos. Entretanto, este diagrama contém as classes **FloorButton** e **ElevatorButton**, que eliminamos recentemente do estudo de caso. Portanto, durante nossa discussão sobre herança, quando mencionarmos o **FloorButton**, referimo-nos ao objeto **Button** do **Floor**, e, quando mencionarmos o **ElevatorButton**, referimo-nos ao objeto **Button** do **Elevator**.

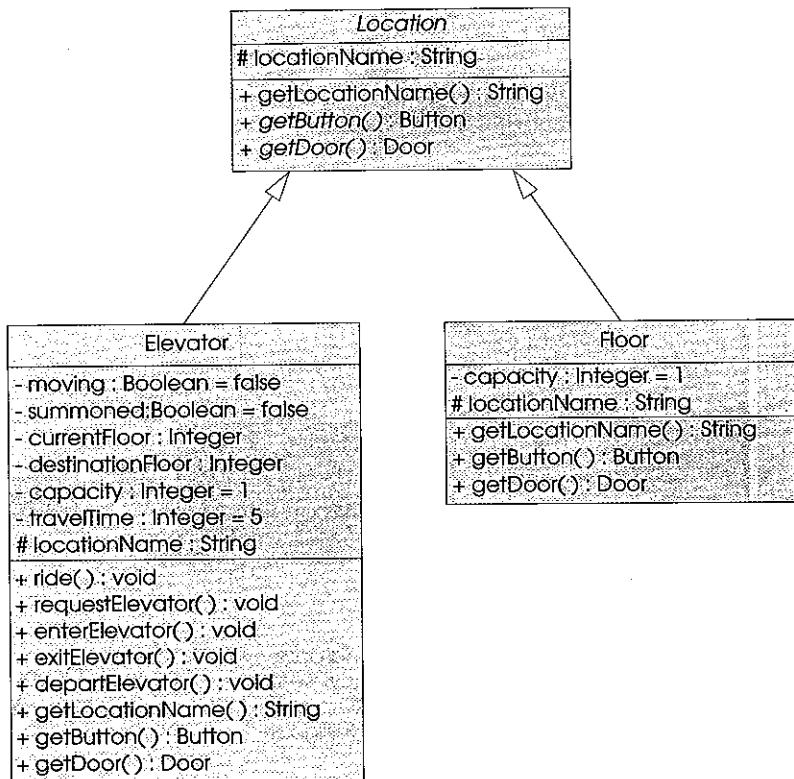


Fig. 9.37 Diagrama de generalização da superclasse **Location** e das subclasses **Elevator** e **Floor**.

aquela **Person** pode interagir somente com os objetos conhecidos por aquela **Location** – a **Person** nunca irá executar uma ação ilegal, como pressionar o **Button** do primeiro **Floor** enquanto estiver andando no **Elevator**.

Apresentamos os atributos de classe e as operações com modificadores de acesso na Fig. 8.22. Agora, apresentamos um diagrama modificado, que incorpora herança, na Fig. 9.39.

A classe **Elevator** agora contém dois objetos **Location**, chamados **currentFloor** e **destinationFloor**, que substituem os valores inteiros que usamos antes para descrever os **Floors**. Por último, **Person** contém um objeto **Location** chamado **location**, que indica se a **Person** está no **Floor** ou no **Elevator**.

Implementação: engenharia progressiva (incorporando herança)

A Seção 8.17 de “Pensando em objetos” usou a UML para expressar a estrutura de classes de Java para nossa simulação. Continuamos nossa implementação enquanto incorporamos herança, usando a classe E como exemplo. Para cada classe no diagrama de classes da Fig. 9.38,

1. Se uma classe **A** é uma subclasse da classe **B**, então **A** estende **B** na declaração de classe e chama o construtor de **B**. Por exemplo, a classe **Elevator** é uma subclasse da superclasse abstrata **Location**, de modo que a declaração da classe deve ser

```

public class Elevator extends Location {
    public Elevator {
  
```

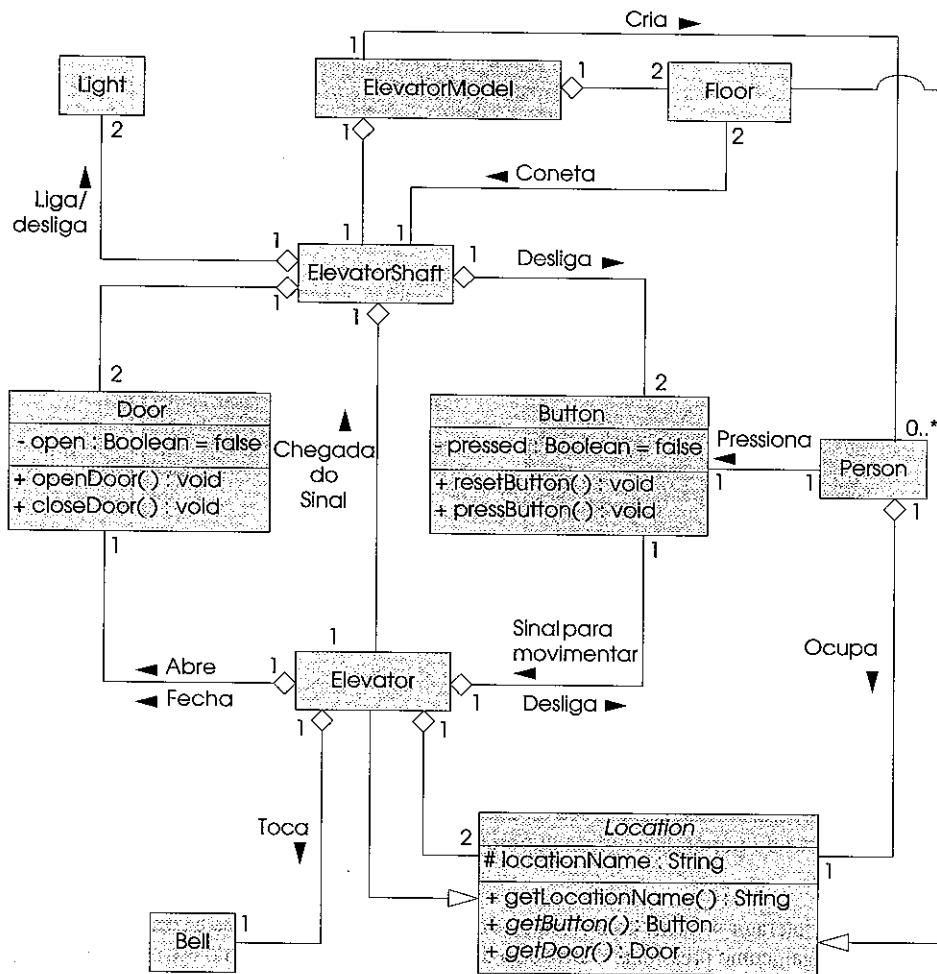


Fig. 9.38 Diagrama de classes de nosso simulador (incorporando herança).

```

{
    super();
}
...

```

2. Se a classe **B** é uma classe abstrata e a classe **A** é uma subclasse de **B**, então a classe **A** deve sobrescrever os métodos abstratos da classe **B** (se a classe **A** deve ser uma classe concreta). Por exemplo, a classe **Location** contém os métodos abstratos **getLocationName**, **getButton** e **getDoor**, de modo que a classe **Elevator** deve sobrescrever estes métodos (observe que **getButton** retorna o objeto **Button** do **Elevator** e **getDoor** retorna o objeto **Door** do **Elevator** – **Elevator** contém associações com os dois objetos, de acordo com o diagrama de classes da Fig. 9.38).

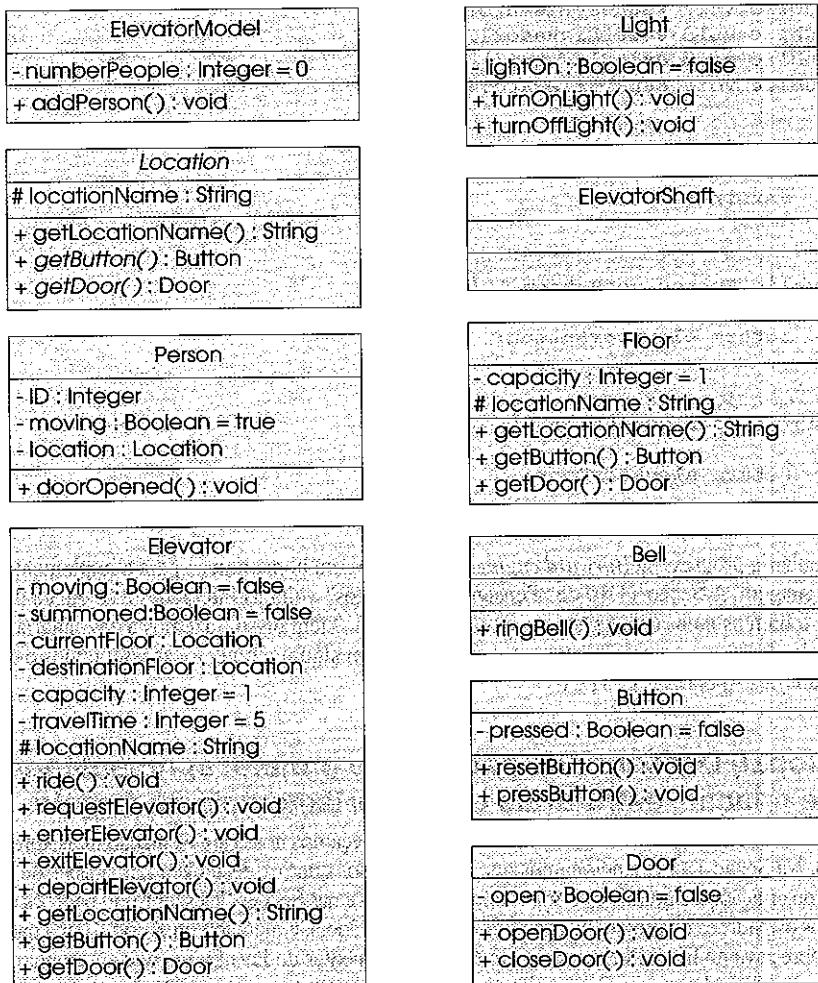


Fig. 9.39 Diagrama de classes com atributos e operações (incorporando herança).

```

public class Elevator extends Location {
    // atributos de classe
    private boolean moving;
    private boolean summoned;
    private Location currentFloor;
    private Location destinationFloor;
    private int capacity = 1;
    private int travelTime = 5;

    // objetos de classe
    private Button elevatorButton;
    private Door elevatorDoor;
    private Bell bell;

    // construtor de classe
    public Elevator()
    {
        super();
    }
}
  
```

```

// métodos de classe
public void ride() {}
public void requestElevator() {}
public void enterElevator() {}
public void exitElevator() {}
public void departElevator() {}

// método sobrescrevendo getLocationName
public String getLocationName()
{
    return "elevator";
}

// método sobrescrevendo getButton
public Button getButton() {}
{
    return elevatorButton;
}

// método sobrescrevendo getDoor
public Door getDoor() {}
{
    return elevatorDoor;
}
}

```

Usar a engenharia progressiva (*forward engineering*) nos dá um bom começo para a implementação de código em qualquer linguagem. A Seção 11.10 de “Pensando em objetos” volta às interações e se concentra em como os objetos geram e tratam as mensagens passadas em colaborações. Além disso, fazemos a engenharia progressiva de mais diagramas de classes para implementar estas interações. Em algum momento, apresentamos o código Java para nosso simulador nos Apêndices G, H e I.

9.24 (Opcional) Descobrindo padrões de projeto: apresentando os padrões de criação, estruturais e comportamentais de projeto

Agora que apresentamos a programação orientada a objetos, começamos nossa apresentação mais aprofundada de padrões de projeto. Na Seção 1.17, mencionamos que a “gangue dos quatro” descreveu 23 padrões de projeto usando três categorias – de criação, estruturais e comportamentais. Nessa e nas Seções “Descobrindo padrões de projeto” restantes, discutimos os padrões de projeto de cada tipo e sua importância, e mencionamos como cada padrão se relaciona ao material sobre Java no livro. Por exemplo, diversos componentes de Java Swing que apresentamos nos Capítulos 12 e 13 usam o padrão de projeto Composite, de modo que apresentamos o padrão de projeto Composite na Seção 13.18. A Fig. 9.40 identifica os 18 padrões de projeto da gangue dos quatro discutidos neste livro.

A Fig. 9.40 lista 18 dos padrões mais utilizados no setor de engenharia de *software*. Existem muitos padrões populares que foram documentados desde o livro da gangue dos quatro – trata-se dos *padrões de projeto concorrente*, que são especialmente úteis no projeto de sistemas com múltiplas *threads*. A Seção 15.13 discute alguns destes padrões usados nas empresas. Os padrões de arquitetura, como discutimos na Seção 17.11, especificam como os subsistemas interagem uns com os outros. A Fig. 9.41 especifica os padrões de concorrência e os padrões de arquitetura que discutimos neste livro.

Seção	Padrões de criação de projeto	Padrões estruturais de projeto	Padrões comportamentais de projeto
9.24	Singleton	Proxy	Memento State
13.18	Factory Model	Adapter Bridge	Chain-of-Responsibility Command

Fig. 9.40 Os 18 padrões de projeto da gangue dos quatro discutidos em *Java Como Programar 4^a Edição* (parte 1 de 2).

Seção	Padrões de criação de projeto	Padrões estruturais de projeto	Padrões comportamentais de projeto
		Composite	Observer Strategy Template Method
17.11	Abstract factory	Decorator Facade	
21.12	Prototype		Iterator

Fig. 9.40 Os 18 padrões de projeto da gangue dos quatro discutidos em *Java Como Programar 4^a Edição* (parte 2 de 2).

Seção	Padrões simultâneos de projeto	Padrões de arquitetura
15.13	Single-Threaded Execution Guarded Suspension Balking Read/Write Lock Two-Phase Termination	
17.11		Model-View-Controller Layers

Fig. 9.41 Padrões simultâneos e de arquitetura de projeto discutidos em *Java Como Programar 4^a Edição*.

9.24.1 Padrões de criação de projeto

Os padrões de criação de projeto abordam aspectos relacionados à criação de objetos, como evitar que um sistema crie mais do que um objeto de uma classe (o padrão de criação Singleton) ou deixar para a execução a decisão sobre que tipos de objetos serão criados (o propósito dos outros padrões de criação discutidos aqui). Por exemplo, suponha que estejamos projetando um programa para desenhar em 3D, no qual o usuário pode criar diversos objetos geométricos tridimensionais, como cilindros, esferas, cubos, tetraedros, etc. Durante a compilação, o programa não sabe que formas o usuário vai optar por desenhar. Com base na informação fornecida pelo usuário, este programa deve ser capaz de determinar a classe da qual instanciar um objeto. Se o usuário cria um cilindro na GUI, nosso programa deve “saber” instanciar um objeto da classe **Cylinder**. Quando o usuário decide qual objeto geométrico desenhar, o programa deve determinar a subclasse específica da qual aquele objeto deve ser instaciado.

A gangue dos quatro descreve cinco padrões de criação (quatro discutimos neste livro):

- Abstract Factory (Seção 17.11)
- Builder (não discutido)
- Factory Method (Seção 13.18)
- Prototype (Seção 21.12)
- Singleton (Seção 9.24)

Singleton

De vez em quando o sistema deve conter exatamente um objeto de uma classe – isto é, depois que o programa instancia aquele objeto, o programa não deve ser autorizado a criar objetos adicionais daquela classe. Por exemplo, alguns sistemas se conectam a um banco de dados com somente um objeto que gerencia conexões a banco de dados, o que assegura que outros objetos não podem inicializar conexões desnecessárias que iriam tornar o sistema mais lento. O *padrão de projeto Singleton* garante que um sistema instancia um máximo de um objeto de uma classe.

A Fig. 9.42 demonstra o código Java que usa o padrão de projeto Singleton. A linha 5 declara a classe **Singleton** como **final**, de modo que os métodos nesta classe não podem ser sobreescritos para manipular múltiplas instâncias. As linhas 11 a 14 declaram um construtor **private** – somente a classe **Singleton** pode instanciar um objeto **Singleton** usando este construtor. O método **static getSingletonInstance** (linhas 17 a 24) permite a instanciação uma única vez de um objeto **static Singleton** (declarado na linha 8) chamando o construtor **private**. Se o objeto **Singleton** já foi criado, a linha 23 simplesmente devolve uma referência para o objeto **Singleton** instaciado anteriormente.

As linhas 10 e 11 da classe **SingletonExample** (Fig. 9.43) declaram duas referências para os objetos **Singleton** – **firstSingleton** e **secondSingleton**. As linhas 14 e 15 chamam o método **getSingletonInstance** e atribuem referências **Singleton** a **firstSingleton** e **secondSingleton**, respectivamente. A linha 18 testa se estes objetos fazem referências ao mesmo objeto **Singleton**. A Fig. 9.44 mostra que **firstSingleton** e **secondSingleton** compartilham a mesma referência para o objeto **Singleton**, porque, cada vez que o método **getSingletonInstance** é chamado, ele devolve uma referência para o mesmo objeto **Singleton**.

```

1 // Singleton.java
2 // Demonstra o padrão de projeto Singleton
3 package com.deitel.jhtp4.designpattern0s;
4
5 public final class Singleton {
6
7     // objeto Singleton devolvido pelo método getSingletonInstance
8     private static Singleton singleton;
9
10    // construtor evita instanciação de outros objetos
11    private Singleton()
12    {
13        System.out.println( "Singleton object created." );
14    }
15
16    // cria Singleton e assegura somente uma instância de Singleton
17    public static Singleton getSingletonInstance()
18    {
19        // instancia Singleton se igual a null
20        if ( singleton == null )
21            singleton = new Singleton();
22
23        return singleton;
24    }
25 }
```

Fig. 9.42 Classe **Singleton** assegura que somente um objeto de sua classe é criado.

```

1 // SingletonExample.java
2 // Tenta criar dois objetos Singleton
3 package com.deitel.jhtp4.designpatterns;
4
```

Fig. 9.43 Classe **SingletonExample** tenta criar um objeto **Singleton** mais de uma vez (parte 1 de 2).

```

5  public class SingletonExample {
6
7      // executa SingletonExample
8      public static void main( String args[] ) {
9      {
10         Singleton firstSingleton;
11         Singleton secondSingleton;
12
13         // cria objetos Singleton
14         firstSingleton = Singleton.getSingletonInstance();
15         secondSingleton = Singleton.getSingletonInstance();
16
17         // os "dois" Singletons devem se referir ao mesmo Singleton
18         if ( firstSingleton == secondSingleton )
19             System.out.println( "firstSingleton and " +
20                     "secondSingleton refer to the same Singleton " +
21                     "object" );
22     }
23 }

```

Fig. 9.43 Classe `SingletonExample` tenta criar um objeto `Singleton` mais de uma vez (parte 2 de 2).



Fig. 9.44 Saída da classe `SingletonExample` mostra que o objeto `Singleton` só pode ser criado uma vez.

9.24.2 Padrões estruturais de projeto

Os *padrões estruturais de projeto* descrevem maneiras comuns de organizar classes e objetos em um sistema. A grandeza dos quatro descreve sete padrões estruturais (seis dos quais discutimos neste livro):

- Adapter (Seção 13.18)
- Bridge (Seção 13.18)
- Composite (Seção 13.18)
- Decorator (Seção 17.11)
- Facade (Seção 17.11)
- Flyweight (não discutido)
- Proxy (Seção 9.24)

Proxy

Um *applet* deve sempre exibir alguma coisa enquanto as imagens estão sendo carregadas. Se esta “qualquer coisa” for uma imagem menor ou um *string* de texto que informa ao usuário que as imagens estão sendo carregadas, é possível se aplicar o *padrão de projeto Proxy* para obter este efeito. Este padrão permite que um objeto aja como um substituto para outro. Pense na carga de diversas imagens grandes (muitos *megabytes*) em um *applet* Java. Idealmente, gostaríamos de ver estas imagens instantaneamente – entretanto, a carga de imagens grandes na memória pode levar tempo para ser completada (especialmente em um processador lento). O padrão de projeto Proxy permite que o sistema use um objeto – chamado de *objeto proxy* – no lugar de outro. Em nosso exemplo, o objeto *proxy* poderia

ser um indicador que informa ao usuário que percentual de uma imagem grande foi carregado. Quando a carga dessa imagem termina, o objeto *proxy* não é mais necessário – o *applet* então pode exibir uma imagem em vez do *proxy*.

9.24.3 Padrões comportamentais de projeto

Existem muitos exemplos diferentes de *padrões comportamentais de projeto*, que oferecem estratégias comprovadas para modelar como os objetos colaboram uns com os outros em um sistema e oferecem comportamentos especiais apropriados para uma grande variedade de aplicações. Vamos considerar o padrão comportamental *Observer* – um exemplo clássico de um padrão de projeto que ilustra a colaboração entre objetos. Por exemplo, os componentes GUI colaboram com seus tratadores de eventos para responder a interações do usuário. Os componentes GUI usam este padrão para processar eventos da interface do usuário. O tratador de eventos observa mudanças de estado em um componente GUI particular se registrando para tratar os eventos daquele componente GUI. Quando o usuário interage com aquele componente GUI, o componente notifica seus tratadores de eventos (também conhecidos como observadores) que o estado do componente GUI mudou (por exemplo, um botão foi pressionado).

Um outro padrão que consideramos é o padrão de projeto de comportamento *Memento* – um exemplo de oferecer comportamento especial para uma grande variedade de aplicativos. O padrão *Memento* permite que um sistema salve o estado de um objeto, de modo que o estado possa ser restaurado em um momento posterior. por exemplo, muitos aplicativos oferecem um recurso de “desfazer” que permite voltar a versões anteriores de seu trabalho.

A gangue dos quatro descreve 11 padrões de comportamento (oito dos quais discutimos neste livro):

- Chain-of-Responsibility (Seção 13.18)
- Command (Seção 13.18)
- Interpreter (não discutido)
- Iterator (Seção 21.12)
- Mediator (não discutido)
- Memento (Seção 9.24)
- Observer (Seção 13.18)
- State (Seção 9.24)
- Strategy (Seção 13.18)
- Template Method (Seção 13.18)
- Visitor (não discutido)

Memento

Imagine um programa de pintar. Este tipo de programa permite criar gráficos. De vez em quando, o usuário pode posicionar um gráfico impropriamente na área de desenho. Os programas de pintar oferecem um recurso “desfazer” que permite ao usuário voltar atrás num erro como este. Especificamente, o programa restaura o estado da área de desenho para aquele antes do usuário ter posicionado o gráfico. Os programas de pintar mais sofisticados oferecem um *histórico*, que armazena diversos estados em uma lista, de modo que o usuário pode restaurar o programa para qualquer estado que esteja no histórico. O *padrão de projeto Memento* permite que um objeto salve seu estado, de modo que – se necessário – o objeto possa ser restaurado para seu estado anterior.

O padrão de projeto *Memento* exige três tipos de objetos. O *objeto originador* ocupa algum *estado* – o conjunto de valores de atributos em um momento específico durante a execução do programa. Em nosso exemplo de programa de pintar, a área de desenho age como o originador, porque ela contém informações sobre atributos que descrevem seu estado – quando o programa é executado pela primeira vez, a área não contém nenhum elemento. O *objeto memento* armazena uma cópia de todos os atributos associados ao estado do originador – isto é, o memento salva o estado da área de desenho. O memento é armazenado como o primeiro item na lista do histórico, que age como o *objeto zelador* – o objeto que contém referências para todos os objetos memento associados ao originador.

Agora, suponha que o usuário desenhe um círculo na área de desenho. A área contém diversas informações que descrevem seu estado – um objeto círculo centrado em coordenadas x - y especificadas. A área de desenho então usa um outro memento para armazenar esta informação. Este memento é armazenado como o segundo item na lista de histórico. A lista de histórico mostra todos os mementos na tela, de modo que o usuário pode selecionar qual estado deve ser restaurado. Suponha que o usuário deseje remover o círculo – se o usuário selecionar o primeiro memento da lista, a área de desenho usa o primeiro memento para restaurar a si mesmo como uma área vazia.

State

Em certos projetos, precisamos transferir as informações sobre o estado de um objeto ou representar os vários estados que um objeto pode ocupar. Nossa estudo de caso opcional de simulação de elevador nas seções “Pensando em objetos” usa o *padrão de projeto State*. Nossa simulação inclui um elevador que se move entre andares em um prédio de dois pavimentos. A pessoa caminha através do andar e anda no elevador para o outro andar. Originalmente, usamos um valor inteiro para representar em que andar a pessoa está caminhando. Entretanto, encontramos um problema quando tentamos responder à pergunta “em que andar a pessoa está quando está andando no elevador ?”. Na verdade, a pessoa não está localizada em nenhum andar – em vez disso, a pessoa está localizada dentro do elevador. Também nos demos conta de que o elevador e os andares são lugares que a pessoa pode ocupar em nossa simulação. Criamos uma superclasse abstrata chamada **Location** para representar um “lugar”. As subclasses **Elevator** e **Floor** herdam da superclasse **Location**. A classe **Person** contém uma referência para um objeto **Location**, que representa o lugar atual – elevador, primeiro andar ou segundo andar – em que aquela pessoa está. Como referência para superclasse pode conter uma referência para subclasse, o atributo da pessoa faz referência ao objeto **Floor** apropriado quando aquela pessoa está num andar e faz referência ao objeto **Elevator** quando aquela pessoa está dentro do elevador.

O elevador e o andar contêm botões (o botão do elevador sinaliza ao elevador para se mover até o outro andar e o botão do andar chama o elevador para o andar do pedido). Como todos os lugares em nossa simulação contêm botões, a classe **Location** oferece o método abstrato **getButton**. A classe **Elevator** implementa o método **getButton** para devolver uma referência para o objeto **Button** dentro de elevador e a classe **Floor** implementa o método **getButton** para devolver uma referência para o objeto **Button** do andar. Usando sua referência **Location**, a pessoa é capaz de pressionar o botão correto – isto é, a pessoa não irá pressionar o botão de um andar quando estiver dentro do elevador e não vai pressionar o botão do elevador quando estiver em um andar.

O padrão de projeto State usa uma superclasse abstrata – chamada de *classe State* – que contém métodos que descrevem comportamentos para estados que um objeto (denominado *objeto de contexto*) pode ocupar. Em nossa simulação de elevador, a classe State é a superclasse **Location** e o objeto de contexto é o objeto da classe **P**. Note que a classe **Location** não descreve todos os estados da classe **Person** (por exemplo, se aquela pessoa está caminhando ou esperando pelo elevador) – a classe **Location** descreve somente o lugar em que a pessoa está e contém o método **getButton** de modo que a **Person** possa acessar o objeto **Button** nos diversos lugares.

Uma *subclasse State*, que estende a classe State, representa um estado individual que o contexto pode ocupar. As subclasses State em nossa simulação são as classes **Elevator** e **Floor**. Cada subclasse State contém métodos que implementam os métodos abstratos da classe State. Por exemplo, tanto a classe **Elevator** quanto a **Floor** implementam o método **getButton**.

O contexto contém exatamente uma referência para um objeto da classe State – esta referência é chamada de *objeto state*. Na simulação, o objeto *state* é o objeto da classe **Location**. Quando o contexto muda de estado, o objeto *state* faz referência ao objeto da subclasse State associado com aquele novo estado. Por exemplo, quando a pessoa caminha do andar para dentro do elevador, a **Location** do objeto **Person** é alterada de fazer referência a um dos objetos **Floor** para fazer referência ao objeto **Elevator**. Quando a pessoa caminha para o andar saindo do elevador, a **Location** do objeto **Person** faz referência ao objeto **Floor** apropriado.

9.24.4 Conclusão

Na Seção 9.24 de “Descobrindo padrões de projeto”, listamos três tipos de padrões de projeto apresentados no livro da gangue dos quatro, identificamos 18 destes padrões de projeto que discutimos neste livro e discutimos padrões de projeto específicos, incluindo Singleton, Proxy, Memento e State. Na Seção 13.18 de “Descobrindo padrões de projeto”, apresentamos alguns padrões de projeto associados aos componentes GUI AWT e Swing. Depois de ler esta seção, você deve entender melhor como os componentes GUI de Java tiram proveito dos padrões de projeto.

9.24.5 Recursos na Internet e na World Wide Web

Os URLs seguintes oferecem mais informações sobre a natureza, a importância e as aplicações de padrões de projeto.

Padrões de projeto

www.hillside.net/patterns

Esta página exibe *links* para informações sobre padrões de projeto e linguagens.

www.hillside.net/patterns/books/

Este site lista livros sobre padrões de projeto.

www.netobjectives.com/design.htm

Apresenta a importância dos padrões de projeto.

umbc7.umbc.edu/~tarr/dp/dp.html

Tem *links* para sites da Web, tutoriais e trabalhos publicados sobre padrões de projetos.

www.links2go.com/topic/Design_Patterns

Tem *links* para sites e informações sobre padrões de projeto.

www.c2.com/ppr/

Discute avanços recentes em padrões de projeto e idéias para futuros projetos.

Padrões de projeto em Java

www.research.umbc.edu/~tarr/cs491/fall00/cs491.html

Este site é para um curso sobre padrões de projeto na University of Maryland e contém inúmeros exemplos de como aplicar padrões de projeto em Java.

www.enteract.com/~bradapp/javapats.html

Discute padrões de projeto para Java e apresenta padrões de projeto em computação distribuída.

www.meurrens.org/ip-Links/java/designPatterns/

Exibe diversos *links* para recursos e informações sobre padrões de projeto em Java.

Padrões de projeto em C++ e Visual Basic

journal.iftech.com/articles/9904_shankel_patterns/

Fornece inspiração para padrões de projeto (o padrão de projeto Iterator em particular) em C++.

mspress.microsoft.com/prod/books/sampchap/2322.htm

Este site dá uma visão geral do livro *Microsoft Visual Basic Design Patterns* (Microsoft Press: 2000).

Padrões de arquitetura

compsci.about.com/science/compsci/library/weekly/aa030600a.htm

Fornece uma visão geral da arquitetura Model-View-Controller.

www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html

Contém um artigo que discute como os componentes Swing usam a arquitetura Model-View-Controller.

www.ootips.org/mvc-pattern.html

Fornece informações e dicas sobre o uso de MVC.

www.ftech.co.uk/~honeyg/articles/pda.htm

Contém um artigo sobre a importância de padrões de arquitetura em software.

www.tml.hut.fi/Opinnot/Tik-109.450/1998/niska/sld001.htm

Fornece informações sobre padrões de arquitetura, padrões de projeto e idiomas (padrões voltados para uma língua específica).

Resumo

- Uma das chaves para o poder da programação orientada a objetos é alcançar a capacidade de reutilização de software por herança.
- Através da herança, uma nova classe herda as variáveis de instância e métodos de uma superclasse previamente definida. Nesse caso, a nova classe é conhecida como subclasse.

- Com herança simples, a classe deriva-se apenas de uma superclasse. Com herança múltipla, uma subclasse herda de várias superclasses. Java não suporta herança múltipla, mas fornece a noção de interfaces, que oferece muitos dos benefícios da herança múltipla.
- A subclasse normalmente adiciona suas próprias variáveis de instância e seus próprios métodos; portanto, em geral, a subclasse é maior que sua superclasse. A subclasse representa um conjunto menor de objetos mais específicos do que os de sua superclasse.
- A subclasse não pode acessar os membros **private** de sua superclasse. A subclasse pode, entretanto, acessar membros **public**, **protected** e com acesso de pacote de sua superclasse. A subclasse deve estar no pacote da superclasse para utilizar membros de superclasse com acesso de pacote.
- O construtor de subclasse sempre chama o construtor da sua superclasse primeiro (explícita ou implicitamente) para criar e inicializar os membros da subclasse herdados da superclasse.
- A herança oferece a capacidade de reutilização de *software*, o que economiza tempo no desenvolvimento e incentiva o uso de *software* de alta qualidade previamente testado e depurado.
- O objeto de uma subclasse pode ser tratado como um objeto de sua superclasse correspondente, mas o contrário não é verdadeiro.
- A superclasse existe em um relacionamento hierárquico com suas subclasses.
- Quando uma classe é utilizada com o mecanismo de herança, ela se torna uma superclasse que fornece atributos e comportamentos para outras classes ou se torna uma subclasse que herda esses atributos e comportamentos.
- A hierarquia de herança pode ter uma profundidade arbitrária dentro das limitações físicas de um sistema particular, mas a maioria das hierarquias de herança tem apenas alguns níveis.
- As hierarquias são úteis para entender e gerenciar complexidade. Com o *software* tornando-se cada vez mais complexo, Java fornece mecanismos para suportar estruturas hierárquicas por herança e polimorfismo.
- O modificador **Protected** serve como um nível intermediário de proteção entre o acesso **public** e o acesso **private**. Os membros **Protected** de uma superclasse podem ser acessados por métodos da superclasse, por métodos de subclasses e por métodos de classes no mesmo pacote.
- A superclasse pode ser uma superclasse direta ou indireta de uma subclasse. A superclasse direta é a classe que uma subclasse estende (**extends**) explicitamente. A superclasse indireta é herdada de vários níveis acima na árvore da hierarquia de classes.
- Quando um membro de superclasse é inadequado para uma subclasse, o programador deve sobrescrever esse membro na subclasse.
- Em um relacionamento “tem um”, um objeto de classe tem uma referência para um objeto de outra classe como um membro. Em um relacionamento “é um”, um objeto de um tipo de subclasse também pode ser tratado como um objeto do tipo superclasse. “É um” é herança. “Tem um” é composição.
- Uma referência a um objeto de subclasse pode ser convertida implicitamente em uma referência a um objeto de superclasse.
- É possível converter uma referência de superclasse em uma referência de subclasse utilizando uma coerção explícita. Se o alvo não for um objeto de subclasse, uma **ClassCastException** é disparada.
- A superclasse especifica aspectos comuns. Todas as classes derivadas de uma superclasse herdam os recursos dessa superclasse. No processo de projeto orientado a objetos, o projetista procura aspectos comuns entre as classes e os fatores para formar as superclasses. As subclasses são então personalizadas além dos recursos herdados da superclasse.
- Ler um conjunto de declarações de subclasse pode ser confuso porque os membros herdados da superclasse não são listados nas declarações de subclasse, mas esses membros estão, de fato, presentes nas subclasses.
- Com o polimorfismo, torna-se possível projetar e implementar sistemas que são mais facilmente extensíveis. Os programas podem ser escritos para processar objetos de tipos que podem não existir quando o programa está em desenvolvimento.
- A programação polimórfica pode eliminar a necessidade de lógica **switch**, evitando, assim, os tipos de erros associados à lógica **switch**.
- O método abstrato é declarado antes da definição do método com a palavra-chave **abstract** na superclasse.
- Há muitas situações em que é útil definir classes para as quais o programador nunca pretenderá instanciar nenhum objeto. Essas classes são chamadas de classes **abstract**. Como essas classes são utilizadas somente como superclasses, normalmente elas são chamadas de superclasses **abstract**. O programa não pode instanciar objetos de uma classe **abstract**.
- As classes a partir das quais o programa pode instanciar objetos são chamadas de classes concretas.
- A classe se torna abstrata declarando-a com a palavra-chave **abstract**.

- Se uma subclasse é derivada de uma superclasse com um método **abstract** sem fornecer uma definição para esse método **abstract** na subclasse, esse método permanece **abstract** na subclasse. Conseqüentemente, a subclasse também é uma classe **abstract**.
- Quando se faz uma solicitação para utilizar um método através de uma referência de superclasse, Java escolhe o método sobrescrito correto na subclasse associada ao objeto.
- Pelo uso de polimorfismo, uma chamada de método pode fazer com que ocorram diferentes ações, dependendo do tipo do objeto que recebe a chamada.
- Apesar de não podermos instanciar objetos de superclasses **abstract**, podemos declarar referências a superclasses **abstract**. Essas referências então podem ser utilizadas para permitir manipulações polimórficas de objetos de subclasse quando esses objetos são instanciados de classes concretas.
- Novas classes são regularmente adicionadas aos sistemas. Novas classes são acomodadas por vinculação dinâmica de método (também chamada de vinculação tardia). O tipo de um objeto não precisa ser conhecido durante a compilação para que uma chamada de método seja compilada. Durante a execução, o método apropriado do objeto receptor é selecionado.
- Com a vinculação dinâmica de método, a chamada para um método é roteada, durante a execução, para a versão apropriada do método na classe do objeto que recebe a chamada.
- Quando uma superclasse fornece um método, as subclasses podem sobreescriver o método, mas não são obrigadas a sobreescrevê-lo. Assim, subclasse pode utilizar uma versão da superclasse de um método.
- A definição de interface inicia com a palavra-chave **interface** e contém um conjunto de métodos **public abstract**. As interfaces também podem conter os dados **public final static**.
- Para utilizar uma interface, a classe deve especificar que implementa (**implements**) a interface e ela deve definir cada método na interface com o número de argumentos e o tipo de retorno especificado na definição de interface.
- A interface é geralmente utilizada no lugar de uma classe abstrata quando não há implementação *default* a herdar.
- Quando a classe implementa uma interface aplica-se o mesmo relacionamento “é um” fornecido por herança.
- Para implementar mais de uma interface, simplesmente forneça uma lista de nomes de interface separados por vírgulas depois da palavra-chave **implements** na definição da classe.
- As classes internas são definidas dentro do escopo de outras classes.
- A classe interna também pode ser definida dentro de um método de uma classe. Tal classe interna tem acesso aos membros da sua classe externa e a variáveis locais **final** do método em que ela é definida.
- As definições de classe interna são utilizadas principalmente no tratamento de eventos.
- A classe **JFrame** fornece os atributos e os comportamentos básicos de uma janela – uma barra de título e botões para minimizar, maximizar e fechar a janela.
- O objeto de classe interna tem acesso a todas as variáveis e métodos do objeto da classe externa.
- Como uma classe interna anônima não tem nome, deve-se criar o objeto da classe interna anônima no ponto em que a classe é definida no programa.
- A classe interna anônima pode implementar uma interface ou estender uma classe.
- O evento gerado quando o usuário clica na caixa de fechamento da janela é um evento de fechar janela.
- O método **addWindowListener** registra um ouvinte de eventos de janela. O argumento para **addWindowListener** deve ser uma referência a um objeto que é um **WindowListener** (pacote `java.awt.event`).
- Para interfaces de tratamento de evento com mais de um método, Java fornece uma classe correspondente (chamada classe adaptadora) que já implementa todos os métodos na interface para você. A classe **WindowAdapter** implementa a interface **WindowListener**; portanto, cada objeto **WindowAdapter** é um **WindowListener**.
- Compilar uma classe que contém classes internas resulta em um arquivo `.class` separado para cada classe.
- As classes internas com nomes de classe podem ser definidas como **public**, **protected**, com acesso de pacote ou **private** e estão sujeitas às mesmas restrições de uso que outros membros de uma classe.
- Para acessar a referência **this** da classe externa, utilize `NomeDaClasseExterna.this`.
- A classe externa é responsável por criar objetos de suas classes internas **não-static**.
- A classe interna pode ser declarada **static**.

Terminologia

abstração
capacidade de reutilização de software
classe abstract

classe básica
classe Boolean
classe Character

<i>classe Double</i>	<i>lógica switch</i>
<i>classe final</i>	<i>membro protected de uma classe</i>
<i>classe Integer</i>	<i>método abstract</i>
<i>classe interna</i>	<i>método final</i>
<i>classe interna anônima</i>	<i>método setSize</i>
<i>classe invólucro de tipo</i>	<i>método setVisible</i>
<i>classe JFrame</i>	<i>método windowClosing</i>
<i>classe Long</i>	<i>objeto-membro</i>
<i>classe Number</i>	<i>polimorfismo</i>
<i>classe Object</i>	<i>programação orientada a objetos (OOP)</i>
<i>classe WindowAdapter</i>	<i>redefinir um método</i>
<i>classe WindowEvent</i>	<i>redefinir um método abstract</i>
<i>cliente de uma classe</i>	<i>referência a uma classe abstract</i>
<i>coleta de lixo</i>	<i>referência de subclasse</i>
<i>componentes padronizados de software</i>	<i>referência de superclasse</i>
<i>composição</i>	<i>relacionamento “é um”</i>
<i>construtor de subclasse</i>	<i>relacionamento “tem um”</i>
<i>construtor de superclasse</i>	<i>relacionamento “usa um”</i>
<i>controle de acesso de membro</i>	<i>relacionamento hierárquico</i>
<i>conversão de referência implícita</i>	<i>sobrescrever versus sobrecarregar</i>
<i>erro de recursão infinita</i>	<i>sobrescrita de método</i>
<i>extends</i>	<i>subclasse</i>
<i>herança</i>	<i>super</i>
<i>herança de implementação</i>	<i>superclasse</i>
<i>herança de interface</i>	<i>superclasse abstract</i>
<i>herança múltipla</i>	<i>superclasse direta</i>
<i>herança simples</i>	<i>superclasse indireta</i>
<i>hierarquia de classe</i>	<i>this</i>
<i>hierarquia de herança</i>	<i>variável de instância final</i>
<i>interface</i>	<i>vinculação dinâmica de método</i>
<i>interface WindowListener</i>	<i>vinculação tardia</i>

Exercícios de auto-revisão

- 9.1 Preencha as lacunas em cada uma das frases seguintes:
- Se a classe **Alpha** herda da classe **Beta**, a classe **Alpha** é chamada de _____ classe e a classe **Beta** é chamada de _____ classe.
 - A herança permite a _____, que economiza tempo no desenvolvimento e estimula a utilização de componentes de *software* previamente testados e de alta qualidade.
 - O objeto de uma _____ classe pode ser tratado como objeto de sua _____ classe correspondente.
 - Os quatro especificadores de acesso de membro são _____, _____, _____, e _____.
 - O relacionamento “tem um” entre as classes representa _____ e o relacionamento “é um” entre as classes representa _____.
 - Utilizar polimorfismo ajuda a eliminar a lógica _____.
 - Se uma classe contiver um ou mais métodos **abstract**, ela é uma classe _____.
 - A chamada de método resolvida durante a execução é chamada de vinculação _____.
 - A subclasse pode chamar qualquer método não-**private** de sua superclasse antes da chamada de método com _____.
- 9.2 Indique se cada uma das afirmações abaixo é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- A superclasse em geral representa um número maior de objetos que sua subclasse (*verdadeira/falsa*).
 - A subclasse em geral encapsula menos funcionalidades do que sua superclasse (*verdadeira/falsa*).

Respostas dos exercícios de auto-revisão

- 9.1** a) sub, super. b) reutilização de *software*. c) sub, super. d) **public**, **protected**, **private** e acesso de pacote. e) composição, herança. f) **switch**. g) **abstract**. h) dinâmica. i) **super**.
- 9.2** a) Verdadeira.
b) Falsa. A subclasse inclui toda a funcionalidade de sua superclasse.

Exercícios

- 9.3** Considere a classe **Bicycle**. Dado seu conhecimento de alguns componentes comuns de bicicletas, mostre uma hierarquia de classe em que a classe **Bicycle** herda de outras classes, que, por sua vez, herdam ainda de outras classes. Discuta a instanciação de vários objetos da classe **Bicycle**. Discuta a herança da classe **Bicycle** para outras subclases intimamente relacionadas.
- 9.4** Defina cada um dos seguintes termos: herança simples, herança múltipla, interface, superclasse e subclasse.
- 9.5** Discuta por que fazer coerção de uma referência de superclasse para uma referência de subclasse é potencialmente perigoso.
- 9.6** Distinga herança simples de herança múltipla. Por que Java não suporta herança múltipla? Que recurso de Java ajuda a explorar os benefícios da herança múltipla?
- 9.7** (*Verdadeiro/falso*) A subclasse é geralmente menor que sua superclasse.
- 9.8** (*Verdadeiro/falso*) O objeto de subclasse também é um objeto da superclasse dessa subclasse.
- 9.9** Alguns programadores preferem não utilizar acesso **protected** porque ele quebra o ocultamento de informações na superclasse. Discuta os méritos relativos de se utilizar acesso **protected** versus acesso **private** em superclasses.
- 9.10** Muitos programas escritos com herança podem ser resolvidos com composição e vice-versa. Discuta os méritos relativos dessas abordagens no contexto da hierarquia de classes **Point**, **Circle**, **Cylinder** neste capítulo. Rescreva o programa das Figs. 9.22 a 9.26 (e as classes de suporte) para utilizar composição em vez de herança. Depois de fazer isso, reveja os méritos relativos das duas abordagens para o problema de **Point**, **Circle**, **Cylinder** e para programas orientados a objetos em geral.
- 9.11** Rescreva o programa **Point**, **Circle**, **Cylinder** das Figs. 9.22 a 9.26 como um programa **Point**, **Square**, **Cube**. Faça isso de duas maneiras – uma vez com herança e outra vez com composição.
- 9.12** No capítulo, declaramos: “Quando um método de superclasse é inadequado para uma subclasse, esse método pode ser sobreescrito na subclasse com uma implementação adequada”. Se isso fosse feito, o relacionamento subclasse-é-um-objeto-da-superclasse ainda se manteria? Explique sua resposta.
- 9.13** Estude a hierarquia de herança da Fig. 9.2. Para cada classe, indique alguns atributos e comportamentos comuns consistentes com a hierarquia. Adicione classes (por exemplo, **AlunoDeGraduacao**, **AlunosDePosGraduacao**, **primeiranista**, **segundanista**, **terceiranista**, **quartanista**, etc.), para enriquecer a hierarquia.
- 9.14** Escreva uma hierarquia de herança para as classes **Quadrilateral**, **Trapezoid**, **Parallelogram**, **Rectangle** e **Square**. Utilize **Quadrilateral** como a superclasse da hierarquia. Faça a hierarquia o mais profunda (isto é, com muitos níveis) possível. Os dados **private** de **Quadrilateral** devem incluir os pares de coordenadas (x, y) para os quatro pontos que limitam o **Quadrilateral**. Escreva um programa de teste que instancia e exibe os objetos de cada uma dessas classes. [No Capítulo 11, você aprenderá a utilizar os recursos de desenho de Java.]
- 9.15** Liste todas as formas que você conseguir pensar – tanto bidimensionais como tridimensionais – e organize essas formas em uma hierarquia de forma. Sua hierarquia deve ter a superclasse **Shape**, a partir da qual a classe **TwoDimensionalShape** e a classe **ThreeDimensionalShape** são derivadas. Uma vez que você desenvolveu a hierarquia, defina cada uma das classes na hierarquia. Utilizaremos essa hierarquia nos exercícios para processar todas as formas como objetos da superclasse **Shape**.
- 9.16** Como é que o polimorfismo permite programar “no geral” em vez de “no específico?” Discuta as vantagens fundamentais da programação “no geral”.
- 9.17** Discuta os problemas de programação com a lógica **switch**. Explique por que o polimorfismo é uma alternativa efetiva à utilização da lógica **switch**.

- 9.18** Distinga herdar interface de herdar implementação. Como as hierarquias de herança projetadas para herdar interface diferem daquelas projetadas para herdar implementação?
- 9.19** Faça uma diferenciação entre os métodos não-**abstract** e os métodos **abstract**.
- 9.20** (*Verdadeiro/falso*) Todos os métodos em uma superclasse **abstract** devem ser declarados **abstract**.
- 9.21** Sugira um ou mais níveis de superclasses **abstract** para a hierarquia **Shape** discutida no início deste capítulo (o primeiro nível é **Shape** e o segundo nível consiste nas classes **TwoDimensionalShape** e **ThreeDimensionalShape**).
- 9.22** Como o polimorfismo promove a capacidade de expansão?
- 9.23** Pediram para você desenvolver um simulador de vôo que terá saídas gráficas elaboradas. Explique por que a programação polimórfica seria especialmente eficaz para um problema dessa natureza.
- 9.24** Desenvolva um pacote básico de imagens gráficas. Utilize a hierarquia de herança da classe **Shape** da Fig. 9.3. Limite-se a formas bidimensionais como quadrados, retângulos, triângulos e círculos. Interaja com o usuário. Deixe o usuário especificar a posição, o tamanho, a forma e as cores de preenchimento a utilizar no desenho de cada forma. O usuário pode especificar muitos itens da mesma forma. Ao criar cada forma, coloque uma referência **Shape** para cada novo objeto **Shape** em um *array*. Cada classe tem seu próprio método **draw**. Escreva um gerenciador de tela polimórfico que percorre o *array* enviando mensagens de **draw** para cada objeto no *array* para formar uma imagem na tela. Redesenhe a imagem na tela toda vez que o usuário especificar uma forma adicional. Investigue os métodos da classe **Graphics** para ajudar a desenhar cada forma.
- 9.25** Modifique o sistema de folha de pagamento das Figs. 9.16 a 9.21 para adicionar as variáveis de instância **private birthDate** (utilize a classe **Date** da Fig. 8.13) e **departmentCode** (um **int**) para classe **Employee**. Suponha que a folha de pagamento seja processada uma vez por mês. Então, quando seu programa calcular a folha de pagamento para cada **Employee** (de maneira polimórfica), adicione um bônus de US\$100,00 à quantidade da folha de pagamento da pessoa se esse for o mês de aniversário do funcionário.
- 9.26** No Exercício 9.15, você desenvolveu uma hierarquia da classe **Shape** e definiu as classes na hierarquia. Modifique a hierarquia de modo que a classe **Shape** seja uma superclasse **abstract** que contém a interface para a hierarquia. Derive **TwoDimensionalShape** e **ThreeDimensionalShape** da classe **Shape** – essas classes também devem ser **abstract**. Utilize um método **abstract print** para enviar para a saída o tipo e as dimensões de cada classe. Também inclua os métodos **area** e **volume** a fim de que esses cálculos possam ser realizados com objetos de cada classe concreta na hierarquia. Escreva um programa que testa a hierarquia da classe **Shape**.
- 9.27** Rescreva sua solução para o exercício 9.26 para utilizar uma interface **Shape** em vez de uma classe abstrata **Shape**.
- 9.28** (*Aplicativo de desenho*) Modifique o programa de desenho do Exercício 8.19 para criar um aplicativo de desenho que desenha linhas aleatórias, retângulos e elipses. [Nota: como o *applet*, o **JFrame** tem um método **paint** que você pode sobrepor para desenhar no fundo do **JFrame**.]

Para esse exercício, modifique as classes **MyLine**, **MyOval** e **MyRect** do Exercício 8.19 para criar a hierarquia de classe na Fig. 9.45. As classes da hierarquia **MyShape** devem ser classes “inteligentes”, de forma que os objetos dessas classes saibam desenhar a si próprios (se providas de um objeto **Graphics** que lhes diga onde desenhar). A única lógica **switch** ou **if/else** nesse programa deve ser determinar o tipo de objeto de forma a ser criado (utilize números aleatórios para selecionar o tipo de forma e as coordenadas de cada forma). Uma vez que um objeto dessa hierarquia é criado, ele será manipulado pelo resto de sua vida útil como uma referência de superclasse **MyShape**.

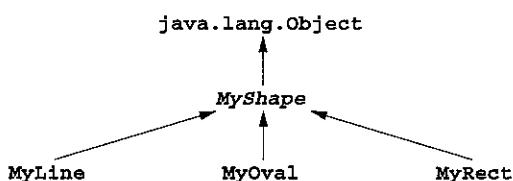


Fig. 9.45 A hierarquia **MyShape**.

A classe **MyShape** na Fig. 9.45 deve ser **abstract**. Os únicos dados que representam as coordenadas das formas na hierarquia devem ser definidos na classe **MyShape**. Todas as linhas, os retângulos e as elipses podem ser desenhados se você conhecer dois pontos no espaço. As linhas exigem as coordenadas *x1*, *y1*, *x2* e *y2*. O método **drawLine** da classe **Graphics** ligará os dois pontos fornecidos com uma linha. Se você tiver os mesmos quatro valores de coordenadas (*x1*, *y1*, *x2* e *y2*) para elipses e retângulos, você pode calcular os quatro argumentos necessários para desenhá-los. Cada um exige um valor da coordenada superior esquerda *x* (o menor dos dois valores de coordenada *x*), um valor da coordenada superior esquerda *y* (o menor dos dois valores de coordenada *y*), uma *largura* (a diferença entre os dois valores de coordenada *x*; deve ser não-negativa) e uma *altura* (a diferença entre os dois valores de coordenada *y*; deve ser não-negativa). [Nota: no Capítulo 12, cada par *x*, *y* será capturado com o mouse, numa interação entre o usuário e o fundo do programa. Essas coordenadas serão armazenadas em um objeto apropriado da forma selecionado pelo usuário. Quando começar o exercício, você utilizará valores de coordenadas aleatórios como argumentos para o construtor.]

Além dos dados para a hierarquia, a classe **MyShape** deve definir pelo menos os seguintes métodos:

- Um construtor sem argumentos que configura as coordenadas com 0.
- Um construtor com argumentos que configura as coordenadas com os valores fornecidos.
- Métodos *set* para cada parte individual dos dados que permitam configurar independentemente qualquer parte dos dados para uma forma na hierarquia (por exemplo, se você tiver uma variável de instância **x1**, deverá ter um método **setX1**).
- Métodos *get* para cada parte individual de dados que permitam recuperar independentemente qualquer parte dos dados para uma forma na hierarquia (por exemplo, se você tiver uma variável de instância **x1**, deverá ter um método **getX1**).
- O método **abstract**

```
public abstract void draw( Graphics g );
```

Esse método será chamado a partir do método **paint** do programa para desenhar uma forma na tela.

Os métodos precedentes são obrigatórios. Se você quiser fornecer mais métodos para obter maior flexibilidade, sinta-se à vontade. Entretanto, certifique-se de que qualquer método que você defina nessa classe seja um método que seria utilizado por *todas* as formas na hierarquia.

Todos os dados devem ser **private** para a classe **MyShape** nesse exercício (isso o força a utilizar um encapsulamento adequado dos dados e fornecer os métodos *set/get* adequados para manipular os dados). Você não tem permissão para definir novos dados que possam ser derivados das informações existentes. Como explicado anteriormente, o *x* superior esquerdo, o *y* superior esquerdo, a *largura* e a *altura* necessárias para desenhar uma elipse ou um retângulo podem ser calculadas se você já conhece dois pontos no espaço. Todas as subclasses de **MyShape** devem fornecer dois construtores que simulam aqueles fornecidos pela classe **MyShape**.

Os objetos das classes **MyOval** e **MyRect** não devem calcular sua coordenada superior esquerda *x*, a coordenada superior esquerda *y*, a *largura* e a *altura* até estarem próximos de ser desenhados. Nunca modifique as coordenadas *x1*, *y1*, *x2* e *y2* de um objeto **MyOval** ou **MyRect** para se preparar para desenhá-los. Em vez disso, utilize os resultados temporários dos cálculos descritos acima. Isso nos ajudará a aprimorar o programa no Capítulo 12, permitindo ao usuário selecionar cada par de coordenadas da forma com o mouse.

Não deve haver referências **MyLine**, **MyOval** ou **MyRect** no programa – apenas referências **MyShape** que fazem referência aos objetos **MyLine**, **MyOval** e **MyRect** são permitidas. O programa deve manter um *array* de referências **MyShape** contendo todas as formas. O método **paint** do programa deve percorrer o *array* de referências **MyShape** e desenhar cada forma (isto é, chamar o método de desenho, **draw**, de cada forma).

Comece definindo a classe **MyShape**, a classe **MyLine** e um aplicativo para testar suas classes. O aplicativo deve ter uma variável de instância **MyShape** que pode fazer referência a um objeto **MyLine** (criado no construtor do aplicativo). O método **paint** (para sua subclasse de **JFrame**) deve desenhar a forma com uma instrução como

```
currentShape.draw( g );
```

onde **currentShape** é a referência **MyShape** e **g** é o objeto **Graphics** que a forma utilizará para se desenhar no fundo da janela.

Em seguida, altere a única referência **MyShape** em um *array* de referências **MyShape** e inclua no código do programa (*hard code*) vários objetos **MyLine** para ser desenhados. O método **paint** do aplicativo deve percorrer o *array* de formas e desenhar todas as formas.

Depois que a parte precedente estiver funcionando, você deve definir as classes **MyOval** e **MyRect** e adicionar os objetos dessas classes ao *array* existente. Por enquanto, todos os objetos de forma devem ser criados no construtor para sua subclasse de **JFrame**. No Capítulo 12, criaremos os objetos quando o usuário escolher uma forma e começar a desenhá-la com o mouse.

9.29 No Exercício 9.28, você definiu uma hierarquia **MyShape** em que as classes **MyLine**, **MyOval** e **MyRect** são subclasses diretas de **MyShape**. Se a hierarquia foi projetada adequadamente, você deve ser capaz de ver as grandes semelhanças entre as classes **MyOval** e **MyRect**. Projete e implemente novamente o código para as classes **MyOval** e **MyRect** para “fatorar” os recursos comuns na classe **abstract MyBoundedShape** a fim de produzir a hierarquia na Fig. 9.46.

A classe **MyBoundedShape** deve definir dois construtores que simulam os construtores de classe **MyShape** e os métodos que calculam a coordenada superior esquerda *x*, a coordenada superior esquerda *y*, a *largura* e a *altura*. Nenhum dos novos dados que dizem respeito às dimensões das formas deve ser definido nessa classe. Lembre-se, os valores necessários para desenhar uma elipse ou um retângulo podem ser calculados a partir de duas coordenadas (*x*, *y*). Se você projetou adequadamente, as novas classes **MyOval** e **MyRect** devem ter dois construtores e um método **draw**.

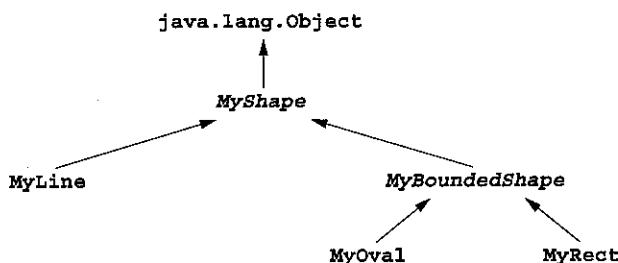


Fig. 9.46 A hierarquia **MyShape**.

10

Strings e caracteres

Objetivos

- Ser capaz de criar e manipular objetos não-modificáveis do tipo *string* de caracteres da classe **String**.
- Ser capaz de criar e manipular objetos modificáveis do tipo *string* de caracteres da classe **StringBuffer**.
- Ser capaz de criar e manipular objetos da classe **Character**.
- Ser capaz de utilizar um objeto **StringTokenizer** para dividir um objeto **String** em componentes individuais chamados *tokens*.

*The chief defect of Henry King
Was chewing little bits of string.*

Hilaire Belloc

Um texto vigoroso é conciso. Uma frase não deve conter nenhuma palavra desnecessária; um parágrafo, nenhuma frase desnecessária.

William Strunk, Jr.

Escrevi uma carta mais longa que o normal, porque me falta tempo para fazê-la mais curta.

Blaise Pascal

A diferença entre a palavra quase certa e a palavra certa é realmente uma questão importante – é a diferença entre um vaga-lume e um relâmpago.

Mark Twain

Mum's the word. ("Shhh"; "Silêncio", ou "Não diga nada sobre o segredo que você sabe.")

Miguel de Cervantes, Don Quixote de la Mancha



Sumário do capítulo

- 10.1 Introdução
- 10.2 Fundamentos de caracteres e *strings*
- 10.3 Construtores de *String*
- 10.4 Os métodos *length*, *charAt* e *getChars* de *String*
- 10.5 Comparando *Strings*
- 10.6 O método *hashCode* de *String*
- 10.7 Localizando caracteres e *substrings* em *Strings*
- 10.8 Extrairindo *substrings* a partir de *strings*
- 10.9 Concatenando *Strings*
- 10.10 Métodos diversos de *String*
- 10.11 Utilizando o método *valueOf* de *String*
- 10.12 O método *intern* de *String*
- 10.13 A classe *StringBuffer*
- 10.14 Construtores de *StringBuffer*
- 10.15 Os métodos *length*, *capacity*, *setLength* e *ensureCapacity* de *StringBuffer*
- 10.16 Os métodos *charAt*, *setCharAt*, *getChars* e *reverse* de *StringBuffer*
- 10.17 Os métodos *append* de *StringBuffer*
- 10.18 Métodos de inserção e de exclusão de *StringBuffer*
- 10.19 Exemplos da classe *Character*
- 10.20 A classe *StringTokenizer*
- 10.21 Simulação de embaralhamento e distribuição de cartas
- 10.22 (Estudo de caso opcional) Pensando em objetos: tratamento de eventos

Resumo • *Terminologia* • *Problemas de auto-revisão* • *Respostas* • *Exercícios de aplicação* • *Exercícios* • *Sessões especiais: exercícios de manipulação de uma fila de strings* • *Atividade de projeto* • *Propriedades desafiadoras de manipulação de strings*

10.1 Introdução

Neste capítulo, apresentamos os recursos de Java para processamento de *strings* e caracteres. As técnicas aqui discutidas são apropriadas para validar os dados de entrada de programas, exibir informações para usuários e outras manipulações baseadas em texto. As técnicas também são apropriadas para desenvolver editores e processadores de texto, *software* de leiaute de página, sistemas computadorizados de composição e outros tipos de *software* de processamento de textos. Já apresentamos vários recursos de processamento de *strings* no texto. Neste capítulo, discutimos em detalhes os recursos da classe *String*, da classe *StringBuffer* e da classe *Character* do pacote *java.lang*, e da classe *StringTokenizer* do pacote *java.util*. Estas classes oferecem os fundamentos para a manipulação de *strings* e caracteres em Java.

10.2 Fundamentos de caracteres e *strings*

Os caracteres são os blocos de construção fundamentais dos programa-fonte Java. Cada programa é composto por uma seqüência de caracteres que – quando agrupados de maneira significativa – é interpretada pelo computador como uma série de instruções utilizadas para realizar uma tarefa. O programa pode conter *caracteres constantes*. O caractere constante é um valor inteiro representado como caractere entre aspas simples. Como declaramos previamen-

te, o valor de um caractere constante é o valor inteiro do caractere no *conjunto de caracteres Unicode*. Por exemplo, '**z**' representa o valor inteiro de **z** e '\n' representa o valor inteiro de nova linha. Veja o Apêndice D para obter os equivalentes em inteiro desses caracteres.

O **String** é uma seqüência de caracteres tratada como uma só unidade. Pode incluir letras, dígitos e vários *caracteres especiais*, como +, -, *, /, \$ e outros. Trata-se de um objeto da classe **String**. Os *literais string* ou *constants string* (frequentemente chamados de *objetos anônimos String*) são escritos como uma seqüência de caracteres entre aspas duplas como segue:

"John Q. Doe"	(<i>nome</i>)
"9999 Main Street"	(<i>endereço numa rua</i>)
"Waltham, Massachusetts"	(<i>cidade e estado</i>)
"(201) 555-1212"	(<i>número de telefone</i>)

O **String** pode ser atribuído em uma declaração a uma referência de **String**. A declaração

```
String color = "blue";
```

inicializa a referência **String color** para fazer referência ao objeto anônimo **String "blue"**.



Dica de desempenho 10.1

Java trata todos os Strings anônimos que tenham o mesmo conteúdo como um objeto anônimo String que tem muitas referências. Isso economiza memória.

10.3 Construtores de String

A classe **String** fornece nove construtores para inicializar objetos **String** de diferentes maneiras. Sete dos construtores são demonstrados na Fig. 10.1. Todos os construtores são utilizados no método **main** do aplicativo **StringConstructors**.

A linha 25 instancia um novo objeto **String** e o atribui à referência **s1** utilizando o construtor *default* da classe **String**. O novo objeto **String** não contém nenhum caractere (*string vazio*) e tem um comprimento de 0.

A linha 26 instancia um novo objeto **String** e o atribui à referência **s2**, utilizando o construtor de cópia da classe **String**. O novo objeto **String** contém uma cópia dos caracteres no objeto **String s**, que é passado como argumento para o construtor.

```

1 // Fig. 10.1: StringConstructors.java
2 // Este programa demonstra os construtores da classe String.
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class StringConstructors {
8
9     // testa os construtores de String
10    public static void main( String args[] )
11    {
12        char charArray[] = { 'b', 'i', 'r', 't', 'h', ' ', ' ',
13                           'd', 'a', 'y' };
14        byte byteArray[] = { ( byte ) 'n', ( byte ) 'e',
15                           ( byte ) 'w', ( byte ) ' ', ( byte ) 'y',
16                           ( byte ) 'e', ( byte ) 'a', ( byte ) 'r' };
17
18        StringBuffer buffer;
19        String s, s1, s2, s3, s4, s5, s6, s7, output,
20
21        s = new String( "hello" );
22        buffer = new StringBuffer( "Welcome to Java Programming!" );
23

```

Fig. 10.1 Demonstrando os construtores da classe **String** (parte 1 de 2).

```

24      // usa os construtores de String
25      s1 = new String();
26      s2 = new String( s );
27      s3 = new String( charArray );
28      s4 = new String( charArray, 6, 3 );
29      s5 = new String( byteArray, 4, 4 );
30      s6 = new String( byteArray );
31      s7 = new String( buffer );
32
33      // acrescenta os Strings à saída
34      output = "s1 = " + s1 + "\ns2 = " + s2 + "\ns3 = " + s3 +
35          "\ns4 = " + s4 + "\ns5 = " + s5 + "\ns6 = " + s6 +
36          "\ns7 = " + s7;
37
38      JOptionPane.showMessageDialog( null, output,
39          "Demonstrating String Class Constructors",
40          JOptionPane.INFORMATION_MESSAGE );
41
42      System.exit( 0 );
43  }
44
45 } // fim da classe StringConstructors

```

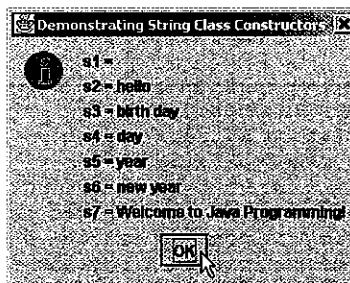


Fig. 10.1 Demonstrando os construtores da classe **String** (parte 2 de 2).



Observação de engenharia de software 10.1

*Na maioria dos casos, não é necessário fazer uma cópia de um objeto **String** existente. Os objetos **String** são imutáveis – seu conteúdo de caracteres não pode ser alterado depois de criado. Além disso, se houver uma ou mais referências a um objeto **String** (ou qualquer objeto, para este assunto) o objeto não pode ser reivindicado pelo coletor de lixo. Portanto, não se pode utilizar uma referência **String** para modificar um objeto **String** ou excluir um objeto **String** da memória como em outras linguagens de programação, como C ou C++.*

A linha 27 instancia um novo objeto **String** e o atribui à referência **s3** utilizando o construtor da classe **String** que recebe um *array* de caracteres como argumento. O novo objeto **String** contém uma cópia dos caracteres do *array*.

A linha 28 instancia um novo objeto **String** e o atribui à referência **s4** utilizando o construtor da classe **String** que recebe um *array* **char** e dois inteiros como argumentos. O segundo argumento especifica a posição inicial (o **offset**) a partir da qual os caracteres do *array* são copiados. O terceiro argumento especifica o número de caracteres (o **count**) que serão copiados do *array*. O novo objeto **String** contém uma cópia dos caracteres especificados no *array*. Se o **offset** ou o **count** especificado como argumento resultar no acesso a um elemento além dos limites do *array* de caracteres, é disparada uma **StringIndexOutOfBoundsException**. Discutimos as exceções em detalhes no Capítulo 14.

A linha 29 instancia um novo objeto **String** e o atribui à referência **s5** utilizando construtor da classe **String** que recebe um *array* **byte** e dois inteiros como argumentos. O segundo e o terceiro argumentos especificam o **offset** e o **count**, respectivamente. O novo objeto **String** contém uma cópia dos **bytes** especificados

no *array*. Se o **offset** ou o **count** especificado como argumento resultar no acesso a um elemento fora dos limites do *array* de caracteres, é disparada uma **StringIndexOutOfBoundsException**.

A linha 30 instancia um novo objeto **String** e o atribui à referência **s6** utilizando construtor da classe **String** que recebe um *array byte* como argumento. O novo objeto **String** contém uma cópia dos *bytes* do *array*.

A linha 31 instancia um novo objeto **String** e o atribui à referência **s7** utilizando o construtor da classe **String** que recebe um **StringBuffer** como argumento. O **StringBuffer** é um *string* dinamicamente modificável e redimensionável. O novo objeto **String** contém uma cópia dos caracteres no **StringBuffer**. A linha 22 cria um novo objeto da classe **StringBuffer** utilizando o construtor que recebe um argumento **String** (neste caso, "Welcome to Java Programming") e atribui o novo objeto à referência **buffer**. Discutimos **StringBuffers** em detalhes mais adiante neste capítulo. A captura de tela do programa exibe o conteúdo de cada **String**.

10.4 Os métodos **length**, **charAt** e **getChars** de **String**

O aplicativo da Fig. 10.2 apresenta os métodos **length**, **charAt** e **getChars** de **String**, que determinam o comprimento de um **String**, obtêm o caractere em uma localização específica em um **String** e obtêm todo o conjunto de caracteres em um **String**, respectivamente.

A linha 28 utiliza o método **length** de **String** para determinar o número de caracteres no **String** **s1**. Assim como *arrays*, **Strings** sempre conhecem seu próprio tamanho. Entretanto, diferentemente dos *arrays*, os **Strings** não têm uma variável de instância **length** que especifica o número de elementos em um **String**.

```

1 // Fig. 10.2: StringMiscellaneous.java
2 // Este programa demonstra os métodos length,
3 // charAt e getChars da classe String.
4 //
5 // Nota: o método getChars requer uma posição inicial e uma
6 // posição final no String. A posição inicial é o subscrito
7 // de onde a cópia realmente inicia. A posição final é uma
8 // após o subscrito no qual a cópia termina.
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 public class StringMiscellaneous {
14
15     // testa diversos métodos de String
16     public static void main( String args[] ) {
17         {
18             String s1, output;
19             char charArray[];
20
21             s1 = new String( "hello there" );
22             charArray = new char[ 5 ];
23
24             // gera o string como saída
25             output = "s1: " + s1;
26
27             // testa o método length
28             output += "\nLength of s1: " + s1.length();
29
30             // percorre com um laço os caracteres em s1 e os exibe na ordem inversa
31             output += "\nThe string reversed is: ";
32
33             for ( int count = s1.length() - 1; count >= 0; count-- )
34                 output += s1.charAt( count ) + " ";

```

Fig. 10.2 Os métodos de manipulação de caracteres da classe **String** (parte 1 de 2).

```

35      // copia os caracteres do string para um array de caracteres
36      s1.getChars( 0, 5, charArray, 0 );
37      output += "\nThe character array is: ";
38
39      for ( int count = 0; count < charArray.length; count++ )
40          output += charArray[ count ];
41
42
43      JOptionPane.showMessageDialog( null, output,
44          "Demonstrating String Class Constructors",
45          JOptionPane.INFORMATION_MESSAGE );
46
47      System.exit( 0 );
48  }
49
50 } // fim da classe StringMiscellaneous

```

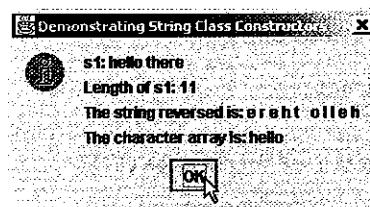


Fig. 10.2 Os métodos de manipulação de caracteres da classe **String** (parte 2 de 2).



Erro comum de programação 10.1

Tentar determinar o comprimento de um **String** através de uma variável de instância chamada **length** (por exemplo, **s1.length**) é um erro de sintaxe. Deve-se utilizar o método **length** de **String** (por exemplo, **s1.length()**).

A estrutura **for** nas linhas 33 e 34 acrescenta a **output** os caracteres do **String** **s1** em ordem inversa. O método **charAt** de **String** devolve o caractere em uma posição específica no **String**. O método **charAt** recebe um argumento inteiro que é utilizado como o *número de posição* (ou *índice*) e retorna o caractere que está nessa posição. De maneira semelhante aos **arrays**, considera-se que o primeiro elemento de um **String** esteja na posição 0.



Erro comum de programação 10.2

Tentar acessar um caractere que está além dos limites de um **String** (isto é, um índice menor que 0 ou um índice maior que ou igual ao comprimento do **String**) resulta em uma **StringIndexOutOfBoundsException**.

A linha 37 utiliza o método **getChars** de **String** para copiar os caracteres de um **String** para um **array** de caracteres. O primeiro argumento é o índice inicial a partir do qual os caracteres são copiados do **String**. O segundo argumento é o índice que está após o último caractere a ser copiado do **String**. O terceiro argumento é o **array** de caracteres para o qual os caracteres são copiados. O último argumento é o índice inicial no qual os caracteres copiados são colocados no **array** de caracteres. A seguir, o conteúdo do **array** **char** é acrescentado, um caractere por vez, ao **String** **output**, sendo que a estrutura **for** nas linhas 40 e 41 serve apenas para fins de exibição.

10.5 Comparando Strings

Java fornece diversos métodos para comparar objetos **String**; esses métodos são demonstrados nos dois exemplos a seguir. Para entender exatamente o que significa um *string* ser “maior que” ou “menor que” outro *string*, pense no processo de colocar em ordem alfabética uma série de sobrenomes. O leitor iria, sem dúvida, colocar “Jones” antes de “Smith” porque a primeira letra de “Jones” vem antes da primeira letra de “Smith” no alfabeto. Mas o alfabeto é

mais que uma simples lista de 26 letras – é uma lista ordenada de caracteres. Cada letra ocupa uma posição específica dentro da lista. O “Z” é mais que apenas uma letra do alfabeto; “Z” é especificamente a vigésima sexta letra do alfabeto.

Como o computador sabe que uma letra vem antes de outra? Todos os caracteres são representados dentro do computador como códigos numéricos (veja o Apêndice D). Quando o computador compara dois *strings*, ele na verdade compara os códigos numéricos dos caracteres nos *strings*.

A Fig.10.3 demonstra os métodos *equals*, *equalsIgnoreCase*, *compareTo* e *regionMatches* de *String* e demonstra a utilização do operador de igualdade *==* para comparar objetos *String*.

```

1 // Fig. 10.3: StringCompare.java
2 // Este programa demonstra os métodos equals,
3 // equalsIgnoreCase, compareTo e regionMatches
4 // da classe String.
5
6 // Pacotes de extensão de Java
7 import javax.swing.JOptionPane;
8
9 public class StringCompare {
10
11     // testa os métodos de comparação da classe String
12     public static void main( String args[] )
13     {
14         String s1, s2, s3, s4, output;
15
16         s1 = new String( "hello" );
17         s2 = new String( "good bye" );
18         s3 = new String( "Happy Birthday" );
19         s4 = new String( "happy birthday" );
20
21         output = "s1 = " + s1 + "\ns2 = " + s2 +
22             "\n\ns3 = " + s3 + "\ns4 = " + s4 + "\n\n";
23
24         // testa igualdade
25         if ( s1.equals( "hello" ) )
26             output += "s1 equals \"hello\"\n";
27         else
28             output += "s1 does not equal \"hello\"\n";
29
30         // testa igualdade com ==
31         if ( s1 == "hello" )
32             output += "s1 equals \"hello\"\n";
33         else
34             output += "s1 does not equal \"hello\"\n";
35
36         // testa igualdade (ignora caixa)
37         if ( s3.equalsIgnoreCase( s4 ) )
38             output += "s3 equals s4\n";
39         else
40             output += "s3 does not equal s4\n";
41
42         // testa compareTo
43         output +=
44             "\ns1.compareTo( s2 ) is " + s1.compareTo( s2 ) +
45             "\ns2.compareTo( s1 ) is " + s2.compareTo( s1 ) +
46             "\ns1.compareTo( s1 ) is " + s1.compareTo( s1 ) +
47             "\ns3.compareTo( s4 ) is " + s3.compareTo( s4 ) +
48             "\ns4.compareTo( s3 ) is " + s4.compareTo( s3 ) +
49             "\n\n";

```

Fig. 10.3 Demonstrando comparações de *Strings* (parte 1 de 2).

```

50
51     // testa regionMatches (diferencia maiúsculas e minúsculas)
52     if ( s3.regionMatches( 0, s4, 0, 5 ) )
53         output += "First 5 characters of s3 and s4 match\n";
54     else
55         output +=
56             "First 5 characters of s3 and s4 do not match\n";
57
58     // testa regionMatches (ignora caixa)
59     if ( s3.regionMatches( true, 0, s4, 0, 5 ) )
60         output += "First 5 characters of s3 and s4 match";
61     else
62         output +=
63             "First 5 characters of s3 and s4 do not match";
64
65     JOptionPane.showMessageDialog( null, output,
66         "Demonstrating String Class Constructors",
67         JOptionPane.INFORMATION_MESSAGE );
68
69     System.exit( 0 );
70 }
71
72 } // fim da classe StringCompare

```

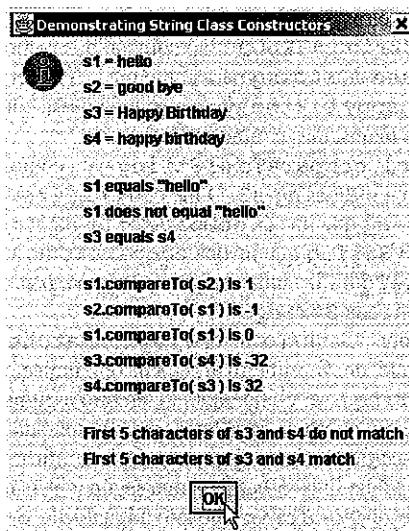


Fig. 10.3 Demonstrando comparações de `String`s (parte 2 de 2).

A condição na estrutura `if` na linha 25 utiliza o método `equals` para comparar o `String` `s1` e o `String` anônimo `"hello"` quanto à igualdade. O método `equals` (herdado pelo `String` de sua superclasse `Object`) testa dois objetos quaisquer quanto à igualdade – o conteúdo de dois objetos é idêntico. O método retorna `true` se os objetos forem iguais e `false` caso contrário. A condição precedente é `true` porque o `String` `s1` foi inicializado com uma cópia do `String` anônimo `"hello"`. O método `equals` utiliza uma *comparação lexicográfica* – os valores inteiros do Unicode que representam cada caractere em cada `String` são comparados. Portanto, se o `String` `"hello"` é comparado como `String` `"HELLO"`, o resultado é `false` porque a representação inteira de uma letra minúscula é diferente daquela da letra maiúscula correspondente.

A condição na estrutura `if` na linha 31 utiliza o operador de igualdade `==` para comparar o `String` `s1` quanto à igualdade com o `String` anônimo `"hello"`. O operador `==` tem funcionalidade diferente quando é utilizado para comparar referências e quando é utilizado para comparar valores de tipos primitivos de dados. Quando valores do tipo primitivo de dados são comparados com `==`, o resultado é `true` se os dois valores forem idênticos.

Quando as referências são comparadas com `==`, o resultado é `true` se as duas referências *se referirem ao mesmo objeto na memória*. Para comparar o conteúdo real (ou informações de estado) de objetos quanto à igualdade, devem-se invocar métodos (como `equals`). A condição precedente é avaliada como `false` nesse programa porque a referência `s1` foi inicializada com a instrução

```
s1 = new String( "hello" );
```

que cria um novo objeto `String` com uma cópia do `String` anônimo `"hello"` e atribui o novo objeto à referência `s1`. Se `s1` tivesse sido inicializada com a instrução

```
s1 = "hello";
```

que diretamente atribui o `String` anônimo `"hello"` à referência `s1`, a condição seria `true`. Lembre-se de que Java trata todos os objetos anônimos `String` com o mesmo conteúdo como um objeto anônimo `String` que tem muitas referências. Portanto, as linhas 16, 25 e 31 fazem referência ao mesmo objeto anônimo `String` `"hello"` na memória.



Erro comum de programação 10.3

Comparar referências com `==` pode levar a erros de lógica, porque `==` compara as referências para determinar se elas se referem ao mesmo objeto, não se dois objetos têm o mesmo conteúdo. Quando dois objetos idênticos (mas separados) são comparados com `==`, o resultado será `false`. Ao comparar objetos para determinar se eles têm o mesmo conteúdo, use o método `equals`.

Se estiver classificando `Strings`, você pode compará-los quanto à igualdade com o método `equalsIgnoreCase`, que ignora diferenças entre letras maiúsculas e minúsculas em cada `String` ao realizar a comparação. Portanto, o `String` `"hello"` e o `String` `"HELLO"` são comparados como iguais. A estrutura `if` na linha 37 utiliza o método `equalsIgnoreCase` de `String` para comparar o `String` `s3 - Happy Birthday` – quanto à igualdade com o `String` `s4 - happy birthday`. O resultado dessa comparação é `true`, porque a comparação ignora as diferenças entre letras maiúsculas e minúsculas.

As linhas 44 a 48 utilizam o método `compareTo` de `String` para comparar objetos `String`. Por exemplo, a linha 44 compara o `String` `s1` com o `String` `s2`. O método `compareTo` devolve 0 se os `Strings` forem iguais, um número negativo se o `String` que invoca `compareTo` for menor que o `String` que é passado como argumento e um número positivo se o `String` que invoca `compareTo` for maior que o `String` que é passado como argumento. O método `compareTo` utiliza uma *comparação lexicográfica* – ele compara os valores numéricos dos caracteres correspondentes em cada `String`.

A condição na estrutura `if` na linha 52 utiliza o método `regionMatches` de `String` para comparar partes de dois objetos `String` quanto à igualdade. O primeiro argumento é o índice inicial no `String` que invoca o método. O segundo argumento é um `String` a ser comparado. O terceiro argumento é o índice inicial no `String` a ser comparado. O último argumento é o número de caracteres a ser comparados entre os dois `Strings`. O método retorna `true` apenas se o número especificado de caracteres for lexicograficamente igual.

Por fim, a condição na estrutura `if` na linha 59 utiliza uma segunda versão do método `regionMatches` de `String` para comparar partes de dois objetos `String` quanto à igualdade. Se o primeiro argumento for `true`, o método ignora as diferenças entre maiúsculas e minúsculas nos caracteres que estão sendo comparados. Os argumentos restantes são idênticos àqueles descritos para o método `regionMatches` com quatro argumentos.

O segundo exemplo desta seção (Fig. 10.4) demonstra os métodos `startsWith` e `endsWith` da classe `String`. O método `main` do aplicativo `StringStartEnd` define um *array* de `Strings` chamado `strings` que contém `"started"`, `"starting"`, `"ended"` e `"ending"`. O restante do método `main` consiste em três estruturas `for` que testam os elementos do *array* para determinar se eles iniciam ou terminam com um conjunto particular de caracteres.

```

1 // Fig. 10.4: StringStartEnd.java
2 // Este programa demonstra os métodos startsWith
3 // e endsWith da classe String.
4
```

Fig. 10.4 Métodos `startsWith` e `endsWith` da classe `String` (parte 1 de 2).

```

5  // Pacotes de extensão de Java
6  import javax.swing.*;
7
8  public class StringStartEnd {
9
10 // testa os métodos de comparação de Strings
11 // para o inicio e o fim de um String
12 public static void main( String args[] )
13 {
14     String strings[] =
15         { "started", "starting", "ended", "ending" };
16     String output = "";
17
18     // testa o método startsWith
19     for ( int count = 0; count < strings.length; count++ )
20
21         if ( strings[ count ].startsWith( "st" ) )
22             output += "\n" + strings[ count ] +
23                 "\n starts with \"st\"\n";
24
25     output += "\n";
26
27     // testa o método startsWith, começando
28     // na posição 2 do string
29     for ( int count = 0; count < strings.length; count++ )
30
31         if ( strings[ count ].startsWith( "art", 2 ) )
32             output += "\n" + strings[ count ] +
33                 "\n starts with \"art\" at position 2\n";
34
35     output += "\n";
36
37     // testa o método endsWith
38     for ( int count = 0; count < strings.length; count++ )
39
40         if ( strings[ count ].endsWith( "ed" ) )
41             output += "\n" + strings[ count ] +
42                 "\n ends with \"ed\"\n";
43
44 JOptionPane.showMessageDialog( null, output,
45                             "Demonstrating String Class Comparisons",
46                             JOptionPane.INFORMATION_MESSAGE );
47
48 System.exit( 0 );
49 }
50 } // fim da classe StringStartEnd

```

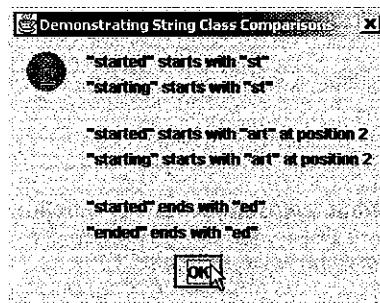


Fig. 10.4 Métodos `startsWith` e `endsWith` da classe `String` (parte 2 de 2).

A primeira estrutura **for** (linhas 19 a 23) utiliza a versão do método **startsWith** que recebe um argumento **String**. A condição na estrutura **if** (linha 21) determina se o **String** na posição **count** do **array** inicia com os caracteres "st". Se iniciar, o método retorna **true**, e **strings[count]** é acrescentado a **output** para fins de exibição.

A segunda estrutura **for** (linhas 29 a 33) utiliza a versão do método **startsWith** que recebe um **String** e um inteiro como argumentos. O argumento inteiro especifica o índice em que a comparação deve iniciar no **String**. A condição na estrutura **if** (linha 31) determina se o **String** na posição **count** do **array** inicia com os caracteres "art", começando com o caractere no índice 2 em cada **String**. Se iniciar, o método retorna **true** e o programa acrescenta **strings[count]** a **output** para fins de exibição.

A terceira estrutura **for** (linhas 38 a 42) utiliza o método **endsWith** que recebe um argumento **String**. A condição na estrutura **if** (linha 40) determina se o **string** na posição **count** do **array** termina com os caracteres "ed". Se terminar, o método retorna **true** e o programa acrescenta **strings[count]** a **output** para fins de exibição.

10.6 O método **hashCode** de **String**

Freqüentemente, é necessário armazenar **Strings** e outros tipos de dados de uma maneira que permita localizar rapidamente as informações. Uma das melhores maneiras de armazenar as informações para pesquisa rápida é uma tabela de *hash*. A tabela de *hash* armazena informações fazendo um cálculo especial com o objeto que será armazenado, que produz um *código de hash*. O código de *hash* é utilizado para escolher a posição na tabela na qual armazenar o objeto. Quando é necessário recuperar as informações, realiza-se o mesmo cálculo, determina-se o código de *hash* e uma pesquisa nessa posição na tabela resulta no valor que foi ali armazenado antes. Qualquer objeto tem a capacidade de ser armazenado em uma tabela de *hash*. A classe **Object** define o método **hashCode** para fazer o cálculo do código de *hash*. Esse método é herdado por todas as subclasses de **Object**. O método **hashCode** é sobreescrito pela classe **String** para fornecer uma boa distribuição de código de *hash* com base no conteúdo do **String**. Falaremos mais sobre *hashing* no Capítulo 20.

O exemplo na Fig. 10.5 demonstra o método **hashCode** para dois **Strings** que contêm "hello" e "Hello". Observe que o valor de código de *hash* para cada **String** é diferente. Isso porque os próprios **Strings** são lexicograficamente diferentes.

```

1 // Fig. 10.5: StringHashCode.java
2 // Este programa demonstra o método
3 // hashCode da classe String.
4
5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class StringHashCode {
9
10    // testa o método hashCode de String
11    public static void main( String args[] )
12    {
13        String s1 = "hello", s2 = "Hello";
14
15        String output =
16            "The hash code for \"\" + s1 + "\" is " +
17            s1.hashCode() +
18            "\nThe hash code for \"\" + s2 + "\" is " +
19            s2.hashCode();
20
21        JOptionPane.showMessageDialog( null, output,
22            "Demonstrating String Method hashCode",
23            JOptionPane.INFORMATION_MESSAGE );
24

```

Fig. 10.5 Método **hashCode** da classe **String** (parte 1 de 2).

```

25     System.exit( 0 );
26 }
27
28 } // fim da classe StringHashCode

```

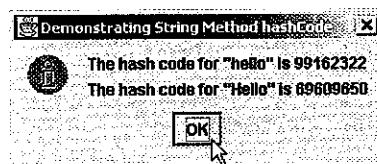


Fig. 10.5 Método hashCode da classe String (parte 2 de 2).

10.7 Localizando caracteres e substrings em Strings

Em geral, é útil procurar um caractere ou um conjunto de caracteres em um **String**. Por exemplo, se estiver criando seu próprio processador de texto, você pode querer fornecer um recurso para pesquisar ao longo do documento. O aplicativo da Fig. 10.6 demonstra as muitas versões dos métodos **indexOf** e **lastIndexOf** de **String** que procuram um caractere ou um *substring* especificado em um **String**. Todas as pesquisas nesse exemplo são feitas no **String letters** (initializado com "abcdefghijklmabcdefghijklm") no método **main** da classe **StringIndexMethods**.

```

1 // Fig. 10.6: StringIndexMethods.java
2 // Este programa demonstra os métodos
3 // "index" da classe String.
4
5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class StringIndexMethods {
9
10    // Métodos de pesquisa em Strings
11    public static void main( String args[] )
12    {
13        String letters = "abcdefghijklmabcdefghijklm";
14
15        // testa indexOf para encontrar um substring em um string
16        String output = "\n'c' is located at index " +
17                      letters.indexOf( 'c' );
18
19        output += "\n'a' is located at index " +
20                      letters.indexOf( 'a', 1 );
21
22        output += "\n'$' is located at index " +
23                      letters.indexOf( '$' );
24
25        // testa lastIndexOf para encontrar um substring em um string
26        output += "\n\nLast 'c' is located at index " +
27                      letters.lastIndexOf( 'c' );
28
29        output += "\nLast 'a' is located at index " +
30                      letters.lastIndexOf( 'a', 25 );
31
32        output += "\nLast '$' is located at index " +
33                      letters.lastIndexOf( '$' );

```

Fig. 10.6 Os métodos de pesquisa da classe String (parte 1 de 2).

```

34
35     // testa indexOf para encontrar um substring em um string
36     output += "\n\n\"def\" is located at index " +
37         letters.indexOf( "def" );
38
39     output += "\n\"def\" is located at index " +
40         letters.indexOf( "def", 7 );
41
42     output += "\n\"hello\" is located at index " +
43         letters.indexOf( "hello" );
44
45     // testa lastIndexOf para encontrar um substring em um string
46     output += "\n\nLast \"def\" is located at index " +
47         letters.lastIndexOf( "def" );
48
49     output += "\nLast \"def\" is located at index " +
50         letters.lastIndexOf( "def", 25 );
51
52     output += "\nLast \"hello\" is located at index " +
53         letters.lastIndexOf( "hello" );
54
55     JOptionPane.showMessageDialog( null, output,
56         "Demonstrating String Class \"index\" Methods",
57         JOptionPane.INFORMATION_MESSAGE );
58
59     System.exit( 0 );
60 }
61
62 } // fim da classe StringTokenizer

```

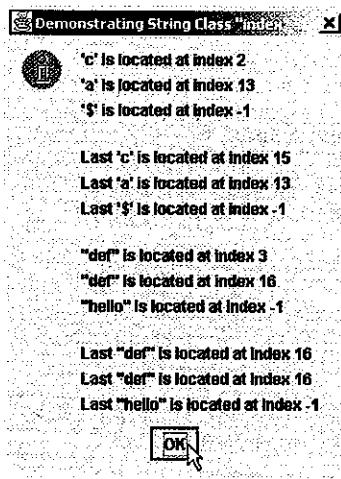


Fig. 10.6 Os métodos de pesquisa da classe **String** (parte 2 de 2).

As instruções nas linhas 16 a 23 utilizam o método **indexOf** para localizar a primeira ocorrência de um caractere em um **String**. Se **indexOf** localizar o caractere, **indexOf** retorna o índice daquele caractere no **String**; caso contrário, **indexOf** retorna **-1**. Há duas versões de **indexOf** que procuram caracteres em um **String**. A expressão na linha 17 utiliza o método **indexOf** que recebe um argumento inteiro que é a representação inteira do caractere. Lembre-se de que uma constante caractere entre apóstrofes é do tipo **char** e especifica a representação inteira do caractere no conjunto de caracteres Unicode. A expressão na linha 20 utiliza a segunda versão do método **indexOf**, que recebe dois argumentos inteiros – a representação inteira de um caractere e o índice inicial em que a pesquisa do **String** deve começar.

As instruções nas linhas 26 a 33 utilizam o método `lastIndexOf` para localizar a última ocorrência de um caractere em um `String`. O método `lastIndexOf` faz a pesquisa a partir do fim do `String` em direção ao início do `String`. Se o método `lastIndexOf` encontrar o caractere, `lastIndexOf` retorna o índice daquele caractere no `String`; caso contrário, `lastIndexOf` retorna -1. Há duas versões de `lastIndexOf` que procuram caracteres em um `String`. A expressão na linha 27 utiliza a versão do método `lastIndexOf` que recebe um argumento inteiro que é a representação inteira de um caractere. A expressão na linha 30 utiliza a versão do método `lastIndexOf` que recebe dois argumentos inteiros – a representação inteira de um caractere e o índice mais alto, a partir do qual inicia a pesquisa do caractere do final para o início.

As linhas 36 a 53 do programa demonstram versões dos métodos `indexOf` e `lastIndexOf`—cada uma delas recebe um `String` como seu primeiro argumento. Essas versões dos métodos trabalham de maneira idêntica à aquelas descritas acima, exceto pelo fato de que procuram seqüências de caracteres (ou *substrings*) que são especificadas por seus argumentos `String`.

10.8 Extraíndo *substrings* a partir de `Strings`

A classe `String` fornece dois métodos `substring` para permitir que um novo objeto `String` seja criado copiando parte de um objeto `String` existente. Cada método retorna um novo objeto `String`. Ambos os métodos são demonstrados na Fig. 10.7.

```

1 // Fig. 10.7: SubString.java
2 // Este programa demonstra os
3 // métodos substring da classe String
4
5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class SubString {
9
10    // testa os métodos substring de String
11    public static void main( String args[] )
12    {
13        String letters = "abcdefghijklmabcdefghijklm";
14
15        // testa os métodos substring
16        String output = "Substring from index 20 to end is " +
17            "\n" + letters.substring( 20 ) + "\n";
18
19        output += "Substring from index 0 up to 6 is " +
20            "\n" + letters.substring( 0, 6 ) + "\n";
21
22        JOptionPane.showMessageDialog( null, output,
23            "Demonstrating String Class Substring Methods",
24            JOptionPane.INFORMATION_MESSAGE );
25
26        System.exit( 0 );
27    }
28
29 } // fim da classe SubString

```

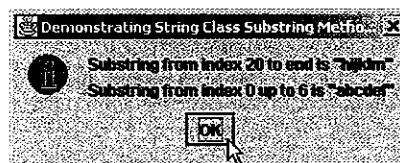


Fig. 10.7 Métodos `substring` da classe `String`.

A expressão `letters.substring(20)` na linha 17 utiliza o método `substring` que recebe um argumento inteiro. O argumento especifica o índice inicial a partir do qual os caracteres são copiados do `String` original. O `substring` devolvido contém uma cópia dos caracteres, a partir do índice inicial, até o final do `String`. Se o índice especificado como argumento estiver fora dos limites do `String`, o programa gera uma `StringIndexOutOfBoundsException`.

A expressão `letters.substring(0, 6)` na linha 20 utiliza o método `substring` que recebe dois argumentos inteiros. O primeiro argumento especifica o índice inicial a partir do qual os caracteres são copiados do `String` original. O segundo argumento especifica o índice que existe além do último caractere que será copiado (isto é, copiar até, mas não incluindo, esse índice no `String`). O `substring` devolvido contém cópias dos caracteres especificados do `String` original. Se os argumentos estiverem fora dos limites do `String`, o programa gera uma `StringIndexOutOfBoundsException`.

10.9 Concatenando Strings

O método `concat` de `String` (Fig. 10.8) concatena dois objetos `String` e retorna um novo objeto `String` que contém os caracteres de ambos os `Strings` originais. Se o argumento `String` não tiver nenhum caractere nele, o `String` original é devolvido. A expressão `s1.concat(s2)` na linha 20 acrescenta os caracteres do `String` `s2` ao final do `String` `s1`. Os `Strings` originais aos quais `s1` e `s2` se referem não são modificados.

Dica de desempenho 10.2



Nos programas que realizam as concatenações de `Strings` ou outras modificações de `Strings` freqüentemente, é mais eficiente implementar estas modificações com a classe `StringBuffer` (explicada nas Seções 10.13 a 10.18).

```

1 // Fig. 10.8: StringConcatenation.java
2 // Este programa demonstra o método concat da classe String.
3 // Note que o método concat retorna um objeto String novo.
4 // Ele não modifica o objeto que invocou o método concat.
5
6 // Pacotes de extensão de Java
7 import javax.swing.*;
8
9 public class StringConcatenation {
10
11     // testa o método concat de String
12     public static void main( String args[] )
13     {
14         String s1 = new String( "Happy " ),
15             s2 = new String( "Birthday" );
16
17         String output = "s1 = " + s1 + "\ns2 = " + s2;
18
19         output += "\n\nResult of s1.concat( s2 ) = " +
20             s1.concat( s2 );
21
22         output += "\n\ns1 after concatenation = " + s1;
23
24         JOptionPane.showMessageDialog( null, output,
25             "Demonstrating String Method concat",
26             JOptionPane.INFORMATION_MESSAGE );
27
28         System.exit( 0 );
29     }
30
31 } // fim da classe StringConcatenation

```

Fig. 10.8 Método `concat` de `String` (parte 1 de 2).

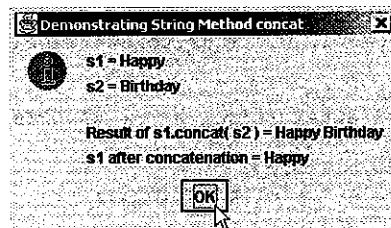


Fig. 10.8 Método `concat` de `String` (parte 2 de 2).

10.10 Métodos diversos de `String`

A classe `String` fornece vários métodos que retornam cópias modificadas de `Strings` ou que retornam um *array* de caracteres. Esses métodos são demonstrados no aplicativo da Fig. 10.9.

A linha 22 utiliza o método `replace` de `String` para retornar um novo objeto `String` no qual o método substitui cada ocorrência do caractere 'l' ('ele') no `String` `s1` pelo caractere 'L'. O método `replace` deixa o `String` original inalterado. Se não houver nenhuma ocorrência do primeiro argumento no `String`, o método `replace` devolve o `String` original.

A linha 26 utiliza o método `toUpperCase` de `String` para gerar um novo objeto `String` com letras maiúsculas onde existem letras minúsculas correspondentes em `s1`. O método retorna um novo objeto `String` que contém o `String` convertido e deixa o `String` original inalterado. Se não houver nenhum caractere para converter em letras maiúsculas, o método `toUpperCase` devolve o `String` original.

A linha 27 utiliza o método `toLowerCase` de `String` para retornar um novo objeto `String` com letras minúsculas onde existem letras maiúsculas correspondentes em `s1`. O `String` original permanece inalterado. Se não houver nenhum caractere para converter em letras minúsculas, o método `toLowerCase` retorna o `String` original.

```

1 // Fig. 10.9: StringMiscellaneous2.java
2 // Este programa demonstra os métodos replace, toLowerCase,
3 // toUpperCase, trim, toString e toCharArray de String.
4
5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class StringMiscellaneous2 {
9
10    // testa diversos métodos de String
11    public static void main( String args[] ) {
12    {
13        String s1 = new String( "hello" ),
14            s2 = new String( "GOOD BYE" ),
15            s3 = new String( " spaces " );
16
17        String output = "s1 = " + s1 + "\ns2 = " + s2 +
18            "\ns3 = " + s3;
19
20        // testa o método replace
21        output += "\n\nReplace 'l' with 'L' in s1: " +
22            s1.replace( 'l', 'L' );
23
24        // testa toLowerCase e toUpperCase
25        output +=

```

Fig. 10.9 Métodos diversos de `String` (parte 1 de 2).

```

26      "\n\ns1.toUpperCase() = " + s1.toUpperCase() +
27      "\ns2.toLowerCase() = " + s2.toLowerCase();
28
29      // testa o método trim
30      output += "\n\ns3 after trim = \n" + s3.trim() + "\n";
31
32      // testa o método toString
33      output += "\n\ns1 = " + s1.toString();
34
35      // testa o método toCharArray
36      char charArray[] = s1.toCharArray();
37
38      output += "\n\ns1 as a character array = ";
39
40      for ( int count = 0; count < charArray.length; ++count )
41          output += charArray[ count ];
42
43      JOptionPane.showMessageDialog( null, output,
44          "Demonstrating Miscellaneous String Methods",
45          JOptionPane.INFORMATION_MESSAGE );
46
47      System.exit( 0 );
48  }
49
50 } // fim da classe StringMiscellaneous2

```

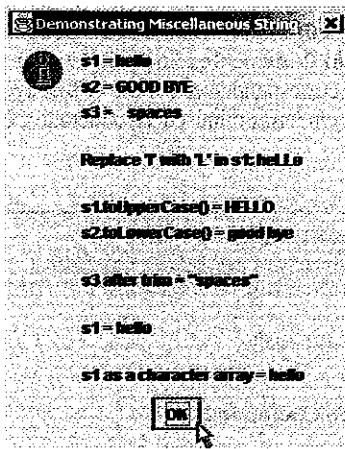


Fig. 10.9 Métodos diversos de **String** (parte 2 de 2).

A linha 30 utiliza o método **trim** de **String**, que remove todos os caracteres de espaço em branco que aparecem no início ou no final do **String** para o qual a mensagem **trim** é enviada, a fim de gerar um novo objeto **String**. O método retorna um novo objeto **String** que contém o **String** sem caracteres de espaço em branco iniciais ou finais. O **String** original permanece inalterado.

A linha 33 usa o método **toString** de **String** para devolver o **String** **s1**. Por que o método **toString** é oferecido para a classe **String**? Todos os objetos podem ser convertidos em **Strings** em Java utilizando o método **toString**, que se origina na classe **Object**. Se uma classe que herda de **Object** (como **String**) não sobrescreve o método **toString**, é utilizada a versão original da classe **Object**. A versão original cria um **String** que consiste no nome da classe do objeto e no código de *hash* para o objeto. Normalmente usa-se o método **toString** para representar o conteúdo de um objeto como texto. O método **toString** é fornecido na classe **String** para assegurar que o valor **String** adequado seja devolvido.

A linha 36 cria um novo *array* de caracteres que contém uma cópia dos caracteres do **String** **s1** e o atribui a **charArray**.

10.11 Utilizando o método valueOf de String

A classe **String** fornece um conjunto de métodos de classe **static** que recebem argumentos de vários tipos, convertem esses argumentos em *strings* e os devolvem como objetos **String**. A classe **StringValueOf** (Fig. 10.10) demonstra os métodos **valueOf** da classe **String**.

```

1 // Fig. 10.10: StringValueOf.java
2 // Este programa demonstra os métodos valueOf da classe String.
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class StringValueOf {
8
9     // testa os métodos valueOf de String
10    public static void main( String args[] )
11    {
12        char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
13        boolean b = true;
14        char c = 'Z';
15        int i = 7;
16        long l = 10000000;
17        float f = 2.5f;
18        double d = 33.333;
19
20        Object o = "hello";    // atribui a uma referência para Object
21        String output;
22
23        output = "char array = " + String.valueOf( charArray ) +
24            "\npart of char array = " +
25            String.valueOf( charArray, 3, 3 ) +
26            "\nboolean = " + String.valueOf( b ) +
27            "\nchar = " + String.valueOf( c ) +
28            "\nint = " + String.valueOf( i ) +
29            "\nlong = " + String.valueOf( l ) +
30            "\nfloat = " + String.valueOf( f ) +
31            "\ndouble = " + String.valueOf( d ) +
32            "\nObject = " + String.valueOf( o );
33
34        JOptionPane.showMessageDialog( null, output,
35            "Demonstrating String Class valueOf Methods",
36            JOptionPane.INFORMATION_MESSAGE );
37
38        System.exit( 0 );
39    }
40
41 } // fim da classe StringValueOf

```

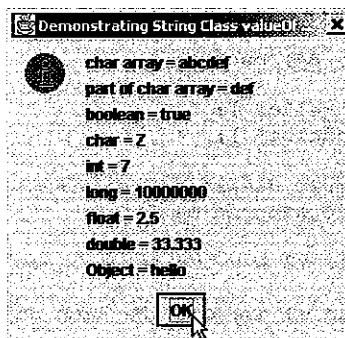


Fig. 10.10 Métodos **valueOf** da classe **String**.

A expressão `String.valueOf(charArray)` da linha 23 copia o conteúdo do `array` de caracteres `charArray` para um novo objeto `String` e devolve o novo `String`. A expressão `String.valueOf(charArray, 3, 3)` da linha 25 copia uma parte do conteúdo do `array` de caracteres `charArray` para um novo objeto `String` e retorna o novo `String`. O segundo argumento especifica o índice inicial a partir do qual os caracteres são copiados. O terceiro argumento especifica o número de caracteres a copiar.

Existem outras sete versões do método `valueOf`, que recebem argumentos do tipo `boolean`, `char`, `int`, `long`, `float`, `double` e `Object`, respectivamente. Elas são demonstradas nas linhas 26 a 32 do programa. Observe que a versão de `valueOf` que aceita um `Object` como um argumento pode fazer isso porque todos os `Objects` podem ser convertidos para `Strings` com o método `toString`.

10.12 O método `intern` de `String`

Comparar grandes objetos `String` é uma operação relativamente lenta. O método `intern` de `String` pode melhorar o desempenho da comparação de `Strings`. Quando o método `intern` de `String` é invocado sobre um objeto `String`, ele retorna uma referência a um objeto `String` que seguramente tem o mesmo conteúdo do `String` original. A classe `String` mantém o `String` resultante durante a execução do programa. Subseqüentemente, se o programa invocar o método `intern` sobre outros objetos `String` cujo conteúdo é idêntico ao do `String` original, o método `intern` retorna uma referência para a cópia do `String` mantido na memória pela classe `String`. Se um programa usar `intern` sobre `Strings` muito grandes, o programa pode comparar esses `Strings` mais rapidamente com o operador `==`, o qual simplesmente compara duas referências – uma operação rápida. Usar métodos `String` padrão, como `equals` e `equalsIgnoreCase` pode ser mais lento, visto que eles comparam caracteres correspondentes em cada `String`. Para grandes `Strings`, esta é uma operação demorada. O programa da Fig. 10.11 demonstra o método `intern`.

O programa declara cinco referências `String` – `s1`, `s2`, `s3`, `s4` e `output`. Os `Strings` `s1` e `s2` são inicializados com novos objetos `String`, cada um contendo uma cópia de "hello". A primeira estrutura `if` (linhas 20 e 23) utiliza o operador `==` para determinar se os `Strings` `s1` e `s2` são o mesmo objeto. As referências `s1` e `s2` se referem a objetos diferentes, porque eles foram inicializados com novos objetos `String`.

A segunda estrutura `if` (linhas 26 a 29) utiliza o método `equals` para determinar se o conteúdo dos `Strings` `s1` e `s2` é igual. Eles foram inicializados com cópias de "hello", de modo que possuem o mesmo conteúdo.

```

1 // Fig. 10.11: StringIntern.java
2 // Este programa demonstra o método
3 // intern da classe String.
4
5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class StringIntern {
9
10    // testa o método intern de String
11    public static void main( String args[] )
12    {
13        String s1, s2, s3, s4, output;
14
15        s1 = new String( "hello" );
16        s2 = new String( "hello" );
17
18        // testa strings para determinar se eles são
19        // o mesmo objeto String na memória
20        if ( s1 == s2 )
21            output = "s1 and s2 are the same object in memory";
22        else
23            output = "s1 and s2 are not the same object in memory";
24

```

Fig. 10.11 Método `intern` da classe `String` (parte 1 de 2).

```

25     // testa a igualdade de conteúdo do String
26     if ( s1.equals( s2 ) )
27         output += "\ns1 and s2 are equal";
28     else
29         output += "\ns1 and s2 are not equal";
30
31     // usa o método intern de String para obter uma cópia única
32     // de "hello" mencionada tanto por s3 quanto por s4
33     s3 = s1.intern();
34     s4 = s2.intern();
35
36     // testa strings para determinar se eles são
37     // o mesmo objeto String na memória
38     if ( s3 == s4 )
39         output += "\ns3 and s4 are the same object in memory";
40     else
41         output +=
42             "\ns3 and s4 are not the same object in memory";
43
44     // determina se s1 e s3 se referem ao mesmo objeto
45     if ( s1 == s3 )
46         output +=
47             "\ns1 and s3 are the same object in memory";
48     else
49         output +=
50             "\ns1 and s3 are not the same object in memory";
51
52     // determina se s2 e s4 se referem ao mesmo objeto
53     if ( s2 == s4 )
54         output += "\ns2 and s4 are the same object in memory";
55     else
56         output +=
57             "\ns2 and s4 are not the same object in memory";
58
59     // determina se s1 e s4 se referem ao mesmo objeto
60     if ( s1 == s4 )
61         output += "\ns1 and s4 are the same object in memory";
62     else
63         output +=
64             "\ns1 and s4 are not the same object in memory";
65
66     JOptionPane.showMessageDialog( null, output,
67         "Demonstrating String Method intern",
68         JOptionPane.INFORMATION_MESSAGE );
69
70     System.exit( 0 );
71 }
72
73 } // fim da classe StringIntern

```

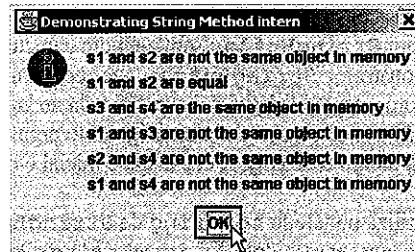


Fig. 10.11 Método `intern` da classe `String` (parte 2 de 2).

A linha 33 utiliza o método `intern` para obter uma referência para um `String` com o mesmo conteúdo do objeto `s1` e a atribui a `s3`. O `String` ao qual `s3` se refere é mantido na memória pela classe `String`. A linha 34 também utiliza o método `intern` para obter uma referência para um objeto `String`. Entretanto, visto que `String s1` e `String s2` têm o mesmo conteúdo, a referência devolvida por essa chamada a `intern` é uma referência ao mesmo objeto `String` devolvido por `s1.intern()`. A terceira estrutura `if` (linhas 38 a 42) utiliza o operador `==` para determinar se os `Strings` `s3` e `s4` se referem ao mesmo objeto.

A quarta estrutura `if` (linhas 45 a 50) utiliza o operador `==` para determinar se os `Strings` `s1` e `s3` não são o mesmo objeto. Tecnicamente, elas poderiam fazer referência ao mesmo objeto, mas não há garantia de que elas fazer referência ao mesmo objeto, a menos que os objetos aos quais elas fazem referência fossem devolvidos por chamadas a `intern` sobre `Strings` com o mesmo conteúdo. Nesse caso, `s1` faz referência ao `String` que foi atribuído no método `main` e `s3` faz referência ao `String` com o mesmo conteúdo mantido pela classe `String`.

A quinta estrutura `if` (linhas 53 a 57) utiliza o operador `==` para determinar se os `Strings` `s2` e `s4` não são o mesmo objeto, porque a segunda chamada `intern` resulta em uma referência ao mesmo objeto devolvido por `s1.intern()`, e não a `s2`. De maneira semelhante, a sexta estrutura `if` (linhas 60 a 64) utiliza operador `==` para determinar se os `Strings` `s1` e `s4` não são o mesmo objeto, porque a segunda chamada a `intern` resulta em uma referência ao mesmo objeto devolvido por `s1.intern()`, e não a `s1`.

10.13 A classe `StringBuffer`

A classe `String` fornece muitos recursos para processamento de `Strings`. Entretanto, uma vez que se cria um objeto `String`, seu conteúdo nunca é alterado. As várias seções a seguir discutem os recursos da classe `StringBuffer` para criar e manipular informações em `strings` dinâmicos – isto é, `Strings` que podem ser modificados. Cada `StringBuffer` pode armazenar uma quantidade de caracteres especificada por sua capacidade. Se a capacidade de um `StringBuffer` for excedida, a capacidade automaticamente é expandida para acomodar os caracteres adicionais. Como veremos, a classe `StringBuffer` também é utilizada na implementação de operadores `+` e `+=` para concatenação de `Strings`.



Dica de desempenho 10.3

Objetos `String` são strings constantes e objetos `StringBuffer` são strings que podem ser modificados. Java distingue strings constantes e strings que podem ser modificados para fins de otimização; em particular, Java pode executar certas otimizações que envolvem objetos `String` (como compartilhar um objeto `String` entre múltiplas referências) porque sabe que esses objetos não serão alterados.



Dica de desempenho 10.4

Quando se oferece a opção de utilizar um objeto `String` para representar um string ou um objeto `StringBuffer` para representar esse string, sempre utilize um objeto `String` se, de fato, o objeto não será alterado; isso melhora o desempenho.



Erro comum de programação 10.4

Invocar os métodos de `StringBuffer` que não são métodos da classe `String` sobre objetos `String` é um erro de sintaxe.

10.14 Construtores de `StringBuffer`

A classe `StringBuffer` fornece três construtores (demonstrados na Fig. 10.12). A linha 14 utiliza o construtor `default StringBuffer` para criar um `StringBuffer` sem nenhum caractere e com uma capacidade inicial de 16 caracteres. A linha 15 utiliza o construtor `StringBuffer` que recebe um argumento inteiro para criar um `StringBuffer` sem caracteres e com a capacidade inicial especificada no argumento inteiro (isto é, 10). A linha 16 utiliza o construtor `StringBuffer`, que recebe um `String` como argumento, para criar um `StringBuffer` que contém os caracteres do argumento `String`. A capacidade inicial é o número de caracteres no argumento `String` mais 16.

A instrução nas linhas 18 a 21 utiliza o método `toString` de `StringBuffer` para converter os `StringBuffers` em objetos `String` que podem ser exibidos com `drawString`. Observe o uso do operador `+` para concatenar `Strings` para saída. Na Seção 10.17, discutimos como Java utiliza `StringBuffers` para implementar os operadores `+` e `+=` para concatenação de `Strings`.

```

1 // Fig. 10.12: StringBufferConstructors.java
2 // Este programa demonstra os construtores StringBuffer.
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class StringBufferConstructors {
8
9     // testa construtores de StringBuffer
10    public static void main( String args[] )
11    {
12        StringBuffer buffer1, buffer2, buffer3;
13
14        buffer1 = new StringBuffer();
15        buffer2 = new StringBuffer( 10 );
16        buffer3 = new StringBuffer( "hello" );
17
18        String output =
19            "buffer1 = \\" + buffer1.toString() + "\\n" +
20            "\nbuffer2 = \\" + buffer2.toString() + "\\n" +
21            "\nbuffer3 = \\" + buffer3.toString() + "\\n";
22
23        JOptionPane.showMessageDialog( null, output,
24            "Demonstrating StringBuffer Class Constructors",
25            JOptionPane.INFORMATION_MESSAGE );
26
27        System.exit( 0 );
28    }
29
30 } // fim da classe StringBufferConstructors

```

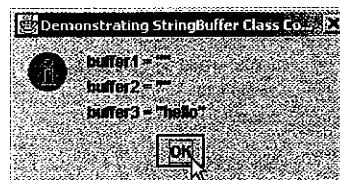


Fig. 10.12 Construtores da classe `StringBuffer`.

10.15 Os métodos `length`, `capacity`, `setLength` e `ensureCapacity` de `StringBuffer`

A classe `StringBuffer` fornece os métodos `length` e `capacity` para se retornar a quantidade de caracteres existentes em um `StringBuffer` e a quantidade de caracteres que pode ser armazenada em um `StringBuffer` sem alocar mais memória, respectivamente. Fornece-se o método `ensureCapacity` para permitir que o programador garanta que um `StringBuffer` tenha uma capacidade mínima. Fornece-se o método `setLength` para permitir que o programador aumente ou diminua o tamanho de um `StringBuffer`. O programa da Fig. 10.13 demonstra esses métodos.

```

1 // Fig. 10.13: StringBufferCapLen.java
2 // Este programa demonstra os métodos length
3 // e capacity da classe StringBuffer.
4

```

Fig. 10.13 Métodos `length` e `capacity` de `StringBuffer` (parte 1 de 2).

```

5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class StringBufferCapLen {
9
10    // testa a capacidade e o tamanho dos métodos de StringBuffer
11    public static void main( String args[] ) {
12    {
13        StringBuffer buffer =
14            new StringBuffer( "Hello, how are you?" );
15
16        String output = "buffer = " + buffer.toString() +
17            "\nlength = " + buffer.length() +
18            "\ncapacity = " + buffer.capacity();
19
20        buffer.ensureCapacity( 75 );
21        output += "\n\nNew capacity = " + buffer.capacity();
22
23        buffer.setLength( 10 );
24        output += "\n\nNew length = " + buffer.length() +
25            "\nbuf = " + buffer.toString();
26
27        JOptionPane.showMessageDialog( null, output,
28            "StringBuffer length and capacity Methods",
29            JOptionPane.INFORMATION_MESSAGE );
30
31        System.exit( 0 );
32    }
33
34 } // fim da classe StringBufferCapLen

```

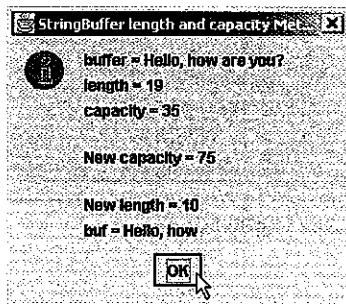


Fig. 10.13 Métodos `length` e `capacity` de `StringBuffer` (parte 2 de 2).

O programa contém um `StringBuffer` chamado `buffer`. As linhas 13 e 14 do programa utilizam o construtor `StringBuffer`, que recebe um argumento `String`, para instanciar e inicializar o `StringBuffer` com `"Hello, how are you?"`. As linhas 16 a 18 acrescentam a `output` o conteúdo, o tamanho e a capacidade do `StringBuffer`. Repare na janela de saída que a capacidade do `StringBuffer` é inicialmente 35. Lembre-se de que o construtor `StringBuffer` que aceita um argumento `String` cria um objeto `StringBuffer` com uma capacidade inicial que é o comprimento do `String` passado como argumento mais 16.

A linha 20 expande a capacidade do `StringBuffer` para um mínimo de 75 caracteres. Na verdade, se a capacidade original for menor que o argumento, o método assegura uma capacidade que é o maior valor entre o número especificado como argumento e o dobro da capacidade original mais 2. Se a capacidade atual do `StringBuffer` for maior que a capacidade especificada, a capacidade do `StringBuffer` permanece inalterada.

A linha 23 utiliza o método `setLength` para estabelecer o tamanho do `StringBuffer` como 10. Se o tamanho especificado for menor que o número atual de caracteres do `StringBuffer`, os caracteres são truncados para o tamanho especificado (isto é, os caracteres do `StringBuffer` que ultrapassam o tamanho especificado são

descartados). Se o tamanho especificado for maior que o número de caracteres existente no `StringBuffer`, acrescentam-se caracteres nulos (caracteres com a representação numérica 0) ao `StringBuffer` até que o número total de caracteres no `StringBuffer` seja igual ao tamanho especificado.

10.16 Os métodos `charAt`, `setCharAt`, `getChars` e `reverse` de `StringBuffer`

A classe `StringBuffer` fornece os métodos `charAt`, `setCharAt`, `getChars` e `reverse` para manipular os caracteres em um `StringBuffer`. O método `charAt` aceita um argumento inteiro e retorna o caractere do `StringBuffer` naquele índice. O método `setCharAt` recebe um inteiro e um caractere como argumentos e coloca o argumento caractere na posição especificada. O índice especificado nos métodos `charAt` e `setCharAt` deve ser maior que ou igual a 0 e menor que o tamanho do `StringBuffer`; caso contrário, gera-se uma `StringIndexOutOfBoundsException`.



Erro comum de programação 10.5

Tentar acessar um caractere que está fora dos limites de um `StringBuffer` (isto é, um índice menor que 0 ou um índice maior que ou igual ao tamanho do `StringBuffer`) resulta em uma `StringIndexOutOfBoundsException`.

O método `getChars` retorna um *array* de caracteres que contém uma cópia dos caracteres no `StringBuffer`. Esse método recebe quatro argumentos – o índice inicial a partir do qual os caracteres devem ser copiados do `StringBuffer`, o índice que excede o último caractere que será copiado do `StringBuffer`, o *array* de caracteres no qual os caracteres devem ser copiados e a posição inicial no *array* de caracteres na qual o primeiro caractere deve ser colocado. O método `reverse` inverte o conteúdo de um `StringBuffer`. Cada um desses métodos é demonstrado na Fig. 10.14.

```

1 // Fig. 10.14: StringBufferChars.java
2 // Os métodos charAt, setCharAt, getChars, e reverse
3 // da classe StringBuffer.
4
5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class StringBufferChars {
9
10    // testa os métodos de StringBuffer para caracteres
11    public static void main( String args[] )
12    {
13        StringBuffer buffer = new StringBuffer( "hello there" );
14
15        String output = "buffer = " + buffer.toString() +
16                    "\nCharacter at 0: " + buffer.charAt( 0 ) +
17                    "\nCharacter at 4: " + buffer.charAt( 4 );
18
19        char charArray[] = new char[ buffer.length() ];
20        buffer.getChars( 0, buffer.length(), charArray, 0 );
21        output += "\n\nThe characters are: ";
22
23        for ( int count = 0; count < charArray.length; ++count )
24            output += charArray[ count ];
25
26        buffer.setCharAt( 0, 'H' );
27        buffer.setCharAt( 6, 'T' );
28        output += "\n\nbuf = " + buffer.toString();
29

```

Fig. 10.14 Métodos de manipulação de caracteres da classe `StringBuffer` (parte 1 de 2).

```

30     buffer.reverse();
31     output += "\n\nbuf = " + buffer.toString();
32
33     JOptionPane.showMessageDialog( null, output,
34         "Demonstrating StringBuffer Character Methods",
35         JOptionPane.INFORMATION_MESSAGE );
36
37     System.exit( 0 );
38 }
39
40 } // fim da classe StringBufferChars

```

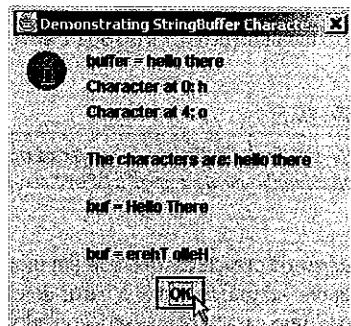


Fig. 10.14 Métodos de manipulação de caracteres da classe `StringBuffer` (parte 2 de 2).

10.17 Os métodos `append` de `StringBuffer`

A classe `StringBuffer` fornece 10 métodos `append` sobrecarregados para permitir adicionar diversos tipos de dados ao final de um `StringBuffer`. As versões são oferecidas para cada um dos tipos primitivos de dados e para *arrays* de caracteres, *Strings* e *Objects* (lembre-se de que o método `toString` produz uma representação `String` de qualquer `Object`). Cada um dos métodos recebe seu argumento, converte-o em um `String` e acrescenta-o ao `StringBuffer`. Os métodos `append` são demonstrados na Fig. 10.15. [Nota: a linha 20 especifica o valor literal `2.5f` como valor inicial de uma variável `float`. Normalmente, Java trata um valor literal em ponto flutuante como do tipo `double`. Acrescentar a letra `f` ao literal `2.5`, indica ao compilador que `2.5` deve ser tratado como do tipo `float`. Sem essa indicação, o compilador gera um erro de sintaxe, porque um valor `double` não pode ser atribuído diretamente a uma variável `float` em Java.]

```

1 // Fig. 10.15: StringBufferAppend.java
2 // Este programa demonstra os métodos
3 // append da classe StringBuffer.
4
5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class StringBufferAppend {
9
10    // testa os métodos append de StringBuffer
11    public static void main( String args[] )
12    {
13        Object o = "hello";
14        String s = "good bye";
15        char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
16        boolean b = true;

```

Fig. 10.15 Métodos `append` da classe `StringBuffer` (parte 1 de 2).

```

17     char c = 'Z';
18     int i = 7;
19     long l = 10000000;
20     float f = 2.5f;
21     double d = 33.333;
22     StringBuffer buffer = new StringBuffer();
23
24     buffer.append( o );
25     buffer.append( " " );
26
27     buffer.append( s );
28     buffer.append( " " );
29     buffer.append( charArray );
30     buffer.append( " " );
31     buffer.append( charArray, 0, 3 );
32     buffer.append( " " );
33     buffer.append( b );
34     buffer.append( " " );
35     buffer.append( c );
36     buffer.append( " " );
37     buffer.append( i );
38     buffer.append( " " );
39     buffer.append( l );
40     buffer.append( " " );
41     buffer.append( f );
42     buffer.append( " " );
43     buffer.append( d );
44
45     JOptionPane.showMessageDialog( null,
46         "buffer = " + buffer.toString(),
47         "Demonstrating StringBuffer append Methods",
48         JOptionPane.INFORMATION_MESSAGE );
49
50     System.exit( 0 );
51 }
52
53 } // fim da classe StringBufferAppend

```

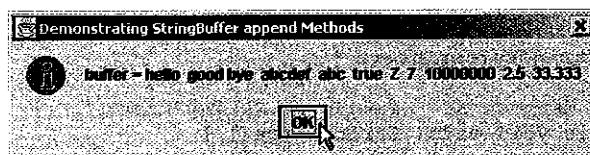


Fig. 10.15 Métodos `append` da classe `StringBuffer` (parte 2 de 2).

Na verdade, `StringBuffers` e os métodos `append` são utilizados pelo compilador para implementar os operadores `+` e `+=` para concatenar `Strings`. Por exemplo, considerando as seguintes declarações:

```

String string1 = "hello";
String string2 = "BC";
int value = 22;

a instrução

String s = string1 + string2 + 22;

concatena "hello", "BC" e 22. A concatenação é realizada da seguinte maneira:

new StringBuffer().append( "hello" ).append( BC ).append(
22 ).toString();

```

Primeiro, Java cria um **StringBuffer**, depois acrescenta ao **StringBuffer** o **String "hello"**, o **String "BC"** e o inteiro **22**. A seguir, o **StringBuffer** é convertido para uma representação **String** com o método **toString** e o resultado é atribuído ao **String s**. A instrução

```
s += "!" ;
```

é executada da seguinte forma:

```
s = new StringBuffer().append( s ).append( "!" ).toString()
```

Primeiro, Java cria um **StringBuffer**, depois acrescenta ao **StringBuffer** o conteúdo atual de **s** seguido pelo **"!"**. A seguir, **toString** de **StringBuffer** converte o **StringBuffer** para uma representação de **String** e o resultado é atribuído a **s**.

10.18 Métodos de inserção e de exclusão de **StringBuffer**

A classe **StringBuffer** fornece nove métodos **insert** sobrecarregados para permitir que valores de vários tipos de dados sejam inseridos em qualquer posição em um **StringBuffer**. São oferecidas versões para cada um dos tipos primitivos de dados e para **Strings**, **Objects** e **arrays** de caracteres (lembre-se de que o método **toString** produz uma representação de **String** de qualquer **Object**). Cada um dos métodos recebe seu segundo argumento, converte-o em um **String** e o insere precedendo o índice especificado pelo primeiro argumento. O índice especificado pelo primeiro argumento deve ser maior que ou igual a 0 e menor que o tamanho de **StringBuffer**; caso contrário, gera-se uma **StringIndexOutOfBoundsException**. A classe **StringBuffer** também fornece métodos **delete** e **deleteCharAt** para excluir caracteres de qualquer posição em um **StringBuffer**. O método **delete** recebe dois argumentos – o subscrito inicial e o subscrito que ultrapassa o final dos caracteres a excluir. Todos os caracteres iniciais, do subscrito inicial até, mas não incluindo, o subscrito final, são excluídos. O método **deleteCharAt** recebe um argumento – o subscrito do caractere a excluir. Os subscritos inválidos fazem com que ambos os métodos disparem uma **StringIndexOutOfBoundsException**. Os métodos **insert** e **delete** são demonstrados na Fig. 10.16.

```

1 // Fig. 10.16: StringBufferInsert.java
2 // Este programa demonstra os métodos
3 // insert e delete da classe StringBuffer.
4
5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class StringBufferInsert {
9
10    // testa os métodos de inserção de StringBuffer
11    public static void main( String args[] )
12    {
13        Object o = "hello";
14        String s = "good bye";
15        char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
16        boolean b = true;
17        char c = 'K';
18        int i = 7;
19        long l = 10000000;
20        float f = 2.5f;
21        double d = 33.333;
22        StringBuffer buffer = new StringBuffer();
23
24        buffer.insert( 0, o );
25        buffer.insert( 0, " " );
26        buffer.insert( 0, s );
27        buffer.insert( 0, " " );
28        buffer.insert( 0, charArray );

```

Fig. 10.16 Métodos **insert** e **delete** da classe **StringBuffer** (parte 1 de 2).

```

29     buffer.insert( 0, " " );
30     buffer.insert( 0, b );
31     buffer.insert( 0, " " );
32     buffer.insert( 0, c );
33     buffer.insert( 0, " " );
34     buffer.insert( 0, i );
35     buffer.insert( 0, " " );
36     buffer.insert( 0, l );
37     buffer.insert( 0, " " );
38     buffer.insert( 0, f );
39     buffer.insert( 0, " " );
40     buffer.insert( 0, d );
41
42     String output =
43         "buffer after inserts:\n" + buffer.toString();
44
45     buffer.deleteCharAt( 10 ); // apaga o 5 em 2.5
46     buffer.delete( 2, 6 ); // apaga .333 em 33.333
47
48     output +=
49         "\n\nbuffer after deletes:\n" + buffer.toString();
50
51     JOptionPane.showMessageDialog( null, output,
52         "Demonstrating StringBufferer Inserts and Deletes",
53         JOptionPane.INFORMATION_MESSAGE );
54
55     System.exit( 0 );
56 }
57
58 } // fim da classe StringBufferInsert

```

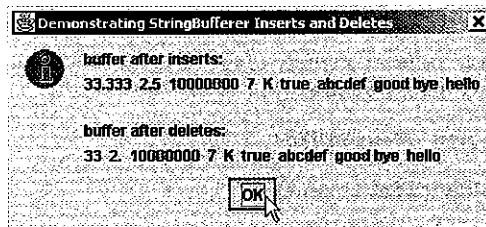


Fig. 10.16 Métodos `insert` e `delete` da classe `StringBuffer` (parte 2 de 2).

10.19 Exemplos da classe Character

Java fornece diversas classes que permitem que variáveis primitivas sejam tratadas como objetos. As classes são `Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` e `Long`. Essas classes (exceto `Boolean` e `Character`) derivam-se de `Number`. Essas oito classes são conhecidas como *invólucro de tipo* e fazem parte do pacote `java.lang`. Os objetos dessas classes podem ser utilizados em qualquer lugar em um programa em que possa se esperar um `Object` ou um `Number`. Nesta seção, apresentamos `Character` – a classe invólucro de tipo para caracteres.

A maioria dos métodos da classe `Character` são `static`, recebem pelo menos um argumento caractere e fazem um teste ou uma manipulação de caractere. Essas classes também contêm um construtor que recebe um argumento `char` para inicializar um objeto `Character` e vários métodos não-`static`. A maioria dos métodos da classe `Character` será apresentada nos próximos três exemplos. Para obter mais informações sobre a classe `Character` (e todas as classes invólucro), veja o pacote `java.lang` na documentação de Java API.

A Fig. 10.17 demonstra alguns métodos `static` que testam caracteres para determinar se eles são um tipo específico de caractere e os métodos `static` que executam conversões de caracteres para letras maiúsculas/minúsculas. Cada método é utilizado no método `buildOutput` da classe `StaticCharMethods`. Você pode inserir

qualquer caractere e aplicar os métodos precedentes ao caractere. Observe o uso de classes internas para o tratamento de eventos, como demonstrado no Capítulo 9.

A linha 63 utiliza o método `isDefined` de `Character` para determinar se o caractere `c` está definido no conjunto de caracteres Unicode. Se estiver, o método retorna `true`; caso contrário, `false`.

```

1 // Fig. 10.17: StaticCharMethods.java
2 // Demonstra o caractere static que testa os métodos e métodos
3 // de conversão de maiúsculas/minúsculas da classe
4 // character a partir do pacote Java.lang.
5
6 // Pacotes do núcleo de Java
7 import java.awt.*;
8 import java.awt.event.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 public class StaticCharMethods extends JFrame {
14     private char c;
15     private JLabel promptLabel;
16     private JTextField inputField;
17     private JTextArea outputArea;
18
19     // configurar GUI
20     public StaticCharMethods()
21     {
22         super( "Static Character Methods" );
23
24         Container container = getContentPane();
25         container.setLayout( new FlowLayout() );
26
27         promptLabel =
28             new JLabel( "Enter a character and press Enter" );
29         container.add( promptLabel );
30
31         inputField = new JTextField( 5 );
32
33         inputField.addActionListener(
34
35             // classe interna anônima
36             new ActionListener() {
37
38                 // Trata o evento do campo de texto
39                 public void actionPerformed( ActionEvent event )
40                 {
41                     String s = event.getActionCommand();
42                     c = s.charAt( 0 );
43                     buildOutput();
44                 }
45
46             } // fim da classe interna anônima
47
48         ); // chamada final para addActionListener
49
50         container.add( inputField );
51
52         outputArea = new JTextArea( 10, 20 );
53         container.add( outputArea );
54

```

Fig. 10.17 Métodos `static` da classe `Character` para testar caracteres e converter para maiúsculas/minúsculas (parte 1 de 3).

```

55     setSize( 300, 250 ); // configura o tamanho da janela
56     show(); // mostra a janela
57 }
58
59 // exibe as informações de character em outputArea
60 public void buildOutput()
61 {
62     outputArea.setText(
63         "is defined: " + Character.isDefined( c ) +
64         "\nis digit: " + Character.isDigit( c ) +
65         "\nis Java letter: " +
66         Character.isJavaIdentifierStart( c ) +
67         "\nis Java letter or digit: " +
68         Character.isJavaIdentifierPart( c ) +
69         "\nis letter: " + Character.isLetter( c ) +
70         "\nis letter or digit: " +
71         Character.isLetterOrDigit( c ) +
72         "\nis lower case: " + Character.isLowerCase( c ) +
73         "\nis upper case: " + Character.isUpperCase( c ) +
74         "\nто upper case: " + Character.toUpperCase( c ) +
75         "\nто lower case: " + Character.toLowerCase( c ) );
76 }
77
78 // executa o aplicativo
79 public static void main( String args[] )
80 {
81     StaticCharMethods application = new StaticCharMethods();
82
83     application.addWindowListener(
84
85         // classe interna anônima
86         new WindowAdapter() {
87
88             // trata o evento quando o usuário fecha a janela
89             public void windowClosing( WindowEvent windowEvent )
90             {
91                 System.exit( 0 );
92             }
93
94         } // fim da classe interna anônima
95
96     ); // chamada final para addWindowListener
97
98 } // fim do método main
99
100 } // fim da classe StaticCharMethods

```

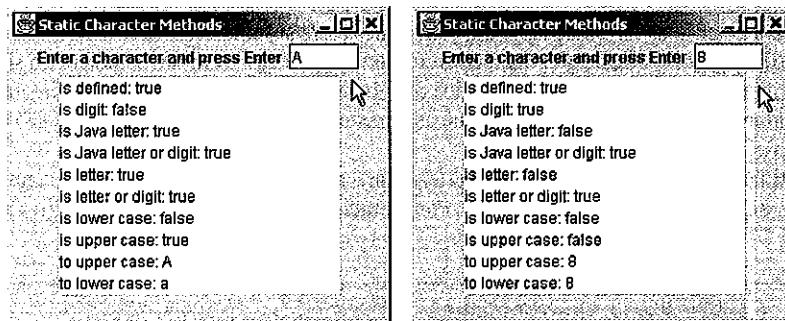


Fig. 10.17 Métodos `static` da classe `Character` para testar caracteres e converter para maiúsculas/minúsculas (parte 2 de 3).

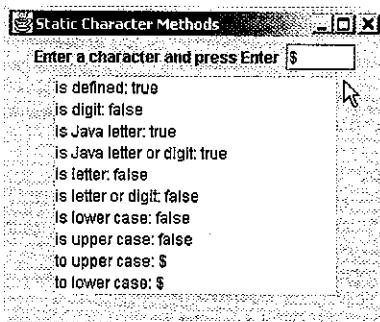


Fig. 10.17 Métodos `static` da classe `Character` para testar caracteres e converter para maiúsculas/minúsculas (parte 3 de 3).

A linha 64 utiliza o método `isDigit` de `Character` para determinar se o caractere `c` é um dígito Unicode válido. Se for, o método retorna `true`; caso contrário, `false`.

A linha 66 utiliza o método `isJavaIdentifierStart` de `Character` para determinar se `c` é um caractere que pode ser utilizado como primeiro caractere de um identificador em Java – isto é, uma letra, um sublinhado () ou um sinal de cifrão (\$). Se for, o método retorna `true`; caso contrário, `false`.

A linha 68 utiliza método `isJavaIdentifierPart` de `Character` para determinar se o caractere `c` é um caractere que pode ser utilizado em um identificador em Java – isto é, um dígito, uma letra, um sublinhado () ou um sinal de cifrão (\$). Se puder, o método retorna `true`; caso contrário, `false`.

A linha 69 utiliza o método `isLetter` de `Character` para determinar se o caractere `c` é uma letra. Se for, o método retorna `true`; caso contrário, retorna `false`. A linha 71 utiliza método `isLetterOrDigit` de `Character` para determinar se o caractere `c` é uma letra ou um dígito. Se for, o método retorna `true`; caso contrário, `false`.

A linha 72 utiliza o método `isLowerCase` de `Character` para determinar se o caractere `c` é uma letra minúscula. Se for, o método retorna `true`; caso contrário, `false`. A linha 73 utiliza o método `isUpperCase` de `Character` para determinar se o caractere `c` é uma letra maiúscula. Se for, o método retorna `true`; caso contrário, `false`.

A linha 74 utiliza o método `toUpperCase` de `Character` para converter o caractere `c` em seu equivalente em letras maiúsculas. O método retorna o caractere convertido se o caractere tiver um equivalente em letras maiúsculas; caso contrário, o método retorna seu argumento original. A linha 75 utiliza o método `toLowerCase` de `Character` para converter o caractere `c` em seu equivalente em letras minúsculas. O método retorna o caractere convertido se o caractere tiver um equivalente em letras minúsculas; caso contrário, o método retorna seu argumento original.

A Fig. 10.18 demonstra os métodos `static digit` e `forDigit` de `Character`, que fazem conversões entre caracteres e dígitos em sistemas de numeração diferentes. Os sistemas numéricos mais comuns incluem decimal (base 10), octal (base 8), hexadecimal (base 16) e binário (base 2). A base de um número também é conhecida como seu *radical*. Para obter mais informações sobre conversões entre sistemas numéricos, veja o Apêndice E.

A linha 52 utiliza método `forDigit` para converter o inteiro `digit` em um caractere no sistema numérico especificado pelo inteiro `radix` (também conhecido como base do número). Por exemplo, o inteiro `13` na base 16 (o `radix`) tem valor de caractere '`d`'. Observe que as letras maiúsculas e letras minúsculas são equivalentes em sistemas numéricos.

A linha 77 utiliza o método `digit` para converter o caractere `c` em um inteiro no sistema numérico especificado pelo inteiro `radix` (isto é, a base do número). Por exemplo, o caractere '`A`' na base 16 (o `radix`) tem o valor inteiro 10.

```

1 // Fig. 10.18: StaticCharMethods2.java
2 // Demonstra os métodos de conversão static de caracteres
3 // da classe Character do pacote java.lang.

```

Fig. 10.18 Métodos de conversão `static` da classe `Character` (parte 1 de 4).

```

4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class StaticCharMethods2 extends JFrame {
13     private char c;
14     private int digit, radix;
15     private JLabel prompt1, prompt2;
16     private JTextField input, radixField;
17     private JButton toChar, toInt;
18
19     public StaticCharMethods2()
20     {
21         super( "Character Conversion Methods" );
22
23         // configura GUI e tratamento de eventos
24         Container container = getContentPane();
25         container.setLayout( new FlowLayout() );
26
27         prompt1 = new JLabel( "Enter a digit or character" );
28         input = new JTextField( 5 );
29         container.add( prompt1 );
30         container.add( input );
31
32         prompt2 = new JLabel( "Enter a radix" );
33         radixField = new JTextField( 5 );
34         container.add( prompt2 );
35         container.add( radixField );
36
37         toChar = new JButton( "Convert digit to character" );
38
39         toChar.addActionListener(
40
41             // classe interna anônima
42             new ActionListener() {
43
44                 // trata evento toChar de JButton
45                 public void actionPerformed( ActionEvent actionEvent )
46                 {
47                     digit = Integer.parseInt( input.getText() );
48                     radix =
49                         Integer.parseInt( radixField.getText() );
50                     JOptionPane.showMessageDialog( null,
51                         "Convert digit to character: " +
52                         Character.forDigit( digit, radix ) );
53                 }
54
55             } // fim da classe interna anônima
56
57         ); // fim da chamada para addActionListener
58
59         container.add( toChar );
60
61         toInt = new JButton( "Convert character to digit" );
62
63         toInt.addActionListener(

```

Fig. 10.18 Métodos de conversão static da classe Character (parte 2 de 4).

```

64
65      // classe interna anônima
66      new ActionListener() {
67
68          // trata evento toInt de JButton
69          public void actionPerformed( ActionEvent actionEvent )
70          {
71              String s = input.getText();
72              c = s.charAt( 0 );
73              radix =
74                  Integer.parseInt( radixField.getText() );
75              JOptionPane.showMessageDialog( null,
76                  "Convert character to digit: " +
77                  Character.digit( c, radix ) );
78          }
79
80      } // fim da classe interna anônima
81
82  ); // fim da chamada para addActionListener
83
84  container.add(ToInt);
85
86  setSize( 275, 150 ); // configura o tamanho da janela
87  show(); // mostra a janela
88
89
90 // executa o aplicativo
91 public static void main( String args[] )
92 {
93     StaticCharMethods2 application = new StaticCharMethods2();
94
95     application.addWindowListener(
96
97         // classe interna anônima
98         new WindowAdapter() {
99
100            // trata evento quando o usuário fecha a janela
101            public void windowClosing( WindowEvent windowEvent )
102            {
103                System.exit( 0 );
104            }
105
106        } // fim da classe interna anônima
107
108    ); // fim da chamada para addWindowListener
109
110 }
111
112 } // fim da classe StaticCharMethods2

```

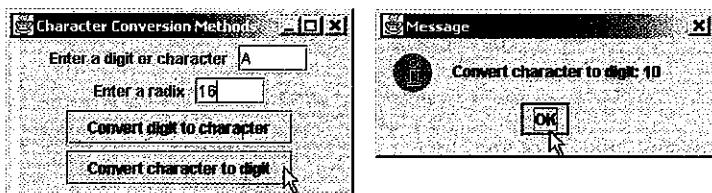


Fig. 10.18 Métodos de conversão static da classe Character (parte 3 de 4).

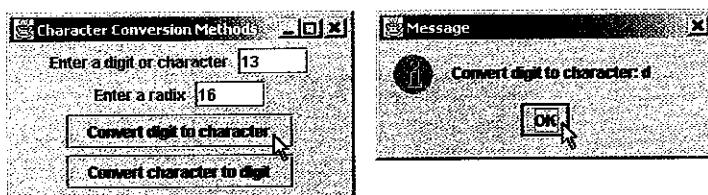


Fig. 10.18 Métodos de conversão `static` da classe `Character` (parte 4 de 4).

O programa da Fig. 10.19 demonstra os métodos não-`static` da classe `Character` – o construtor, `charValue`, `toString`, `hashCode` e `equals`.

As linhas 15 e 16 instanciam dois objetos `Character` e passam literais de caractere para o construtor inicializar esses objetos.

A linha 18 utiliza o método `charValue` de `Character` para devolver o valor `char` armazenado no objeto `Character` `c1`. A linha 19 retorna uma representação `String` do objeto `Character` `c2` utilizando o método `toString`.

As linhas 20 e 21 calculam o `hashCode` dos objetos `Character` `c1` e `c2`, respectivamente. Lembre-se de que os valores do código de `hash` são utilizados para armazenar objetos em tabelas de `hash` para acelerar as capacidades de pesquisa (veja o Capítulo 20).

A condição da estrutura `if` na linha 23 utiliza o método `equals` para determinar se o objeto `c1` tem o mesmo conteúdo que o objeto `c2` (isto é, os caracteres dentro de cada objeto são iguais).

```

1 // Fig. 10.19: OtherCharMethods.java
2 // Demonstra os métodos não-static da
3 // classe Character do pacote java.lang.
4
5 // Pacotes de extensão de Java
6 import javax.swing.*;
7
8 public class OtherCharMethods {
9
10    // testa os métodos não-static de Character
11    public static void main( String args[] ) {
12        {
13            Character c1, c2;
14
15            c1 = new Character( 'A' );
16            c2 = new Character( 'a' );
17
18            String output = "c1 = " + c1.charValue() +
19                         "\nc2 = " + c2.toString() +
20                         "\n\nhash code for c1 = " + c1.hashCode() +
21                         "\nhash code for c2 = " + c2.hashCode();
22
23            if ( c1.equals( c2 ) )
24                output += "\n\n c1 and c2 are equal";
25            else
26                output += "\n\n c1 and c2 are not equal";
27
28            JOptionPane.showMessageDialog( null, output,
29                                         "Demonstrating Non-Static Character Methods",
30                                         JOptionPane.INFORMATION_MESSAGE );
31
32            System.exit( 0 );
33        }

```

Fig. 10.19 Métodos não-`static` da classe `Character` (parte 1 de 2).

```

34
35 } // fim da classe OtherCharMethods

```

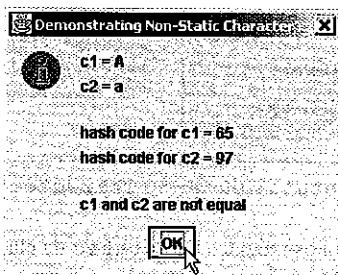


Fig. 10.19 Métodos não-static da classe Character (parte 2 de 2).

10.20 A classe StringTokenizer

Quando você lê uma frase, sua mente divide a frase em palavras e marcas de pontuação ou *tokens*, e cada uma delas possui um significado para você. Os compiladores também realizam a “tokenização”. Eles dividem as instruções em pedaços individuais como palavras-chave, identificadores, operadores e outros elementos de uma linguagem de programação. Nesta seção, estudamos a classe **StringTokenizer** de Java (do pacote **java.util**) a qual divide um *string* nos *tokens* que o compõem. Os *tokens* são separados entre si por delimitadores, em geral caracteres de espaçamento como espaço em branco, tabulação, nova linha e retorno de carro. Outros caracteres também podem ser utilizados como delimitadores para separar *tokens*. O programa da Fig. 10.20 demonstra a classe **StringTokenizer**. A janela para a classe **TokenTest** exibe um **JTextField** em que o usuário digita uma frase a ser separada em *tokens*. A saída desse programa é exibida em uma **JTextArea**.

Quando o usuário pressiona a tecla *Enter* no **JTextField**, o método **actionPerformed** (nas linhas 37 a 49) é invocado. As linhas 39 e 40 atribuem à referência **String stringToTokenize** o valor do texto no **JTextField** devolvido pela chamada a **event.getActionCommand()**. Em seguida, as linhas 41 e 42 criam uma instância de classe **StringTokenizer**. Este construtor **StringTokenizer** recebe um argumento **String** e cria um **StringTokenizer** para **stringToTokenize** que utilizará o *string* delimitador *default* " \n\t\r" que consiste em um espaço, uma nova linha, uma tabulação e um retorno de carro para separação em *tokens*. Existem dois outros construtores na classe **StringTokenizer**. Na versão que recebe dois argumentos **String**, o segundo **String** é o **String** delimitador. Na versão que recebe três argumentos, o segundo **String** é o **String** delimitador e o terceiro argumento (um **boolean**) determina se os delimitadores também são devolvidos como *tokens* (sómente se o argumento for **true**). Isso é útil se você precisar saber quais são os delimitadores.

```

1 // Fig. 10.20: TokenTest.java
2 // Testando a classe StringTokenizer do pacote java.util
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6 import java.awt.*;
7 import java.awt.event.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class TokenTest extends JFrame {
13     private JLabel promptLabel;
14     private JTextField inputField;
15     private JTextArea outputArea;
16

```

Fig. 10.20 Separando strings em tokens com um objeto StringTokenizer (parte 1 de 3).

```

17 // configura GUI e tratamento de eventos
18 public TokenTest()
19 {
20     super( "Testing Class StringTokenizer" );
21
22     Container container = getContentPane();
23     container.setLayout( new FlowLayout() );
24
25     promptLabel =
26         new JLabel( "Enter a sentence and press Enter" );
27     container.add( promptLabel );
28
29     inputField = new JTextField( 20 );
30
31     inputField.addActionListener(
32
33         // classe interna anônima
34         new ActionListener() {
35
36             // trata o evento do campo de texto
37             public void actionPerformed( ActionEvent event )
38             {
39                 String stringToTokenize =
40                     event.getActionCommand();
41                 StringTokenizer tokens =
42                     new StringTokenizer( stringToTokenize );
43
44                 outputArea.setText( "Number of elements: " +
45                     tokens.countTokens() + "\nThe tokens are:\n" );
46
47                 while ( tokens.hasMoreTokens() )
48                     outputArea.append( tokens.nextToken() + "\n" );
49             }
50
51         } // fim da classe interna anônima
52
53     ); // fim da chamada para addActionListener
54
55     container.add( inputField );
56
57     outputArea = new JTextArea( 10, 20 );
58     outputArea.setEditable( false );
59     container.add( new JScrollPane( outputArea ) );
60
61     setSize( 275, 260 ); // configura o tamanho da janela
62     show(); // mostra a janela
63 }
64
65 // executa o aplicativo
66 public static void main( String args[] )
67 {
68     TokenTest application = new TokenTest();
69
70     application.addWindowListener(
71
72         // classe interna anônima
73         new WindowAdapter() {
74
75             // trata evento quando o usuário fecha a janela
76             public void windowClosing( WindowEvent windowEvent )

```

Fig. 10.20 Separando strings em tokens com um objeto StringTokenizer (parte 2 de 3).

```

77         {
78             System.exit( 0 );
79         }
80     } // fim da classe interna anônima
81
82     ); // fim da chamada para addWindowListener
83
84 }
85 // fim do método main
86
87 } // fim da classe TokenTest

```

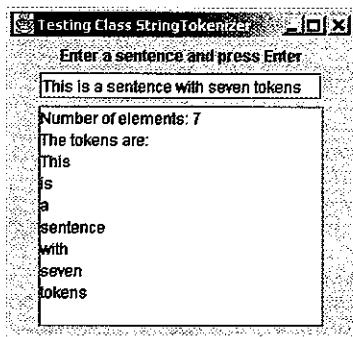


Fig. 10.20 Separando *strings* em *tokens* com um objeto **StringTokenizer** (parte 3 de 3).

A instrução nas linhas 33 a 35 utiliza o método **countTokens** de **StringTokenizer** para determinar o número de *tokens* no **String** a ser separado em *tokens*.

A condição na estrutura **while** nas linhas 47 e 48 utiliza o método **hasMoreTokens()** de **StringTokenizer** para determinar se existem mais *tokens* no **String** que está sendo separado. Se houver, o método **append** é invocado para a **JTextArea outputArea** para acrescentar o próximo *token* ao **String** na **JTextArea**. Obtém-se próximo *token* pela chamada do método **nextToken()** de **StringTokenizer**, que retorna um **string**. O *token* enviado para a saída é seguido por um caractere de nova linha, de modo que os *tokens* subsequentes apareçam em linhas separadas.

Se quisesse alterar o **String** delimitador ao separar em *tokens* um **String**, você poderia fazer isso especificando um novo **string** delimitador em uma chamada a **nextToken** da seguinte maneira:

```
tokens.nextToken( newDelimiterString );
```

Esse recurso não é demonstrado no programa.

10.21 Simulação de embaralhamento e distribuição de cartas

Nesta seção, utilizamos a geração de números aleatórios para desenvolver um programa que simula o embaralhamento e a distribuição de cartas. Esse programa pode então ser utilizado para implementar programas que simulam jogos de cartas específicos.

Desenvolvemos o aplicativo **DeckOfCards** (Fig. 10.21), que cria um baralho de 52 cartas utilizando objetos **Card**, e depois permite que o usuário distribua cada carta clicando no botão “**Deal card**”. Cada carta distribuída é exibida em um **JTextField**. O usuário pode embaralhar as cartas a qualquer hora, clicando no botão “**Shuffle cards**”.

```

1 // Fig. 10.21: DeckOfCards.java
2 // Programa que embaralha e distribui cartas
3

```

Fig. 10.21 Programa que distribui cartas (parte 1 de 4).

```

4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class DeckOfCards extends JFrame {
12     private Card deck[];
13     private int currentCard;
14     private JButton dealButton, shuffleButton;
15     private JTextField displayField;
16     private JLabel statusLabel;
17
18     // configura o baralho e a GUI
19     public DeckOfCards()
20     {
21         super( "Card Dealing Program" );
22
23         String faces[] = { "Ace", "Deuce", "Three", "Four",
24                            "Five", "Six", "Seven", "Eight", "Nine", "Ten",
25                            "Jack", "Queen", "King" };
26         String suits[] =
27             { "Hearts", "Diamonds", "Clubs", "Spades" };
28
29         deck = new Card[ 52 ];
30         currentCard = -1;
31
32         // preenche o baralho com objetos Card
33         for ( int count = 0; count < deck.length; count++ )
34             deck[ count ] = new Card( faces[ count % 13 ],
35                                     suits[ count / 13 ] );
36
37         // configura GUI e tratamento de eventos
38         Container container = getContentPane();
39         container.setLayout( new FlowLayout() );
40
41         dealButton = new JButton( "Deal card" );
42         dealButton.addActionListener(
43
44             // classe interna anônima
45             new ActionListener() {
46
47                 // distribui uma carta
48                 public void actionPerformed( ActionEvent actionEvent )
49                 {
50                     Card dealt = dealCard();
51
52                     if ( dealt != null ) {
53                         displayField.setText( dealt.toString() );
54                         statusLabel.setText( "Card #: " + currentCard );
55                     }
56                     else {
57                         displayField.setText(
58                             "NO MORE CARDS TO DEAL" );
59                         statusLabel.setText(
60                             "Shuffle cards to continue" );
61                     }
62                 }
63             }
64         );
65     }
66
67     // classe interna anônima
68     class Card implements Comparable<Card> {
69
70         String face;
71         String suit;
72
73         public Card( String face, String suit ) {
74             this.face = face;
75             this.suit = suit;
76         }
77
78         public String toString() {
79             return face + " of " + suit;
80         }
81
82         public int compareTo( Card card ) {
83             int value = face.compareTo( card.face );
84             if ( value == 0 )
85                 value = suit.compareTo( card.suit );
86             return value;
87         }
88     }
89
90     // método para gerar uma carta aleatória
91     Card dealCard() {
92         if ( currentCard >= 51 )
93             shuffle();
94
95         currentCard++;
96
97         return deck[ currentCard ];
98     }
99
100    // método para embaralhar o baralho
101    void shuffle() {
102        Card temp;
103
104        for ( int i = 0; i < 52; i++ ) {
105            int index = (int)( Math.random() * 52 );
106
107            temp = deck[ i ];
108            deck[ i ] = deck[ index ];
109            deck[ index ] = temp;
110        }
111    }
112
113    // método para limpar a tela
114    void clear() {
115        displayField.setText( "" );
116        statusLabel.setText( "" );
117    }
118
119    // método para sair do programa
120    void exit() {
121        System.exit( 0 );
122    }
123
124    // método para gerar uma carta aleatória
125    Card dealCard() {
126        if ( currentCard >= 51 )
127            shuffle();
128
129        currentCard++;
130
131        return deck[ currentCard ];
132    }
133
134    // método para embaralhar o baralho
135    void shuffle() {
136        Card temp;
137
138        for ( int i = 0; i < 52; i++ ) {
139            int index = (int)( Math.random() * 52 );
140
141            temp = deck[ i ];
142            deck[ i ] = deck[ index ];
143            deck[ index ] = temp;
144        }
145    }
146
147    // método para limpar a tela
148    void clear() {
149        displayField.setText( "" );
150        statusLabel.setText( "" );
151    }
152
153    // método para sair do programa
154    void exit() {
155        System.exit( 0 );
156    }
157
158    // método para gerar uma carta aleatória
159    Card dealCard() {
160        if ( currentCard >= 51 )
161            shuffle();
162
163        currentCard++;
164
165        return deck[ currentCard ];
166    }
167
168    // método para embaralhar o baralho
169    void shuffle() {
170        Card temp;
171
172        for ( int i = 0; i < 52; i++ ) {
173            int index = (int)( Math.random() * 52 );
174
175            temp = deck[ i ];
176            deck[ i ] = deck[ index ];
177            deck[ index ] = temp;
178        }
179    }
180
181    // método para limpar a tela
182    void clear() {
183        displayField.setText( "" );
184        statusLabel.setText( "" );
185    }
186
187    // método para sair do programa
188    void exit() {
189        System.exit( 0 );
190    }
191
192    // método para gerar uma carta aleatória
193    Card dealCard() {
194        if ( currentCard >= 51 )
195            shuffle();
196
197        currentCard++;
198
199        return deck[ currentCard ];
200    }
201
202    // método para embaralhar o baralho
203    void shuffle() {
204        Card temp;
205
206        for ( int i = 0; i < 52; i++ ) {
207            int index = (int)( Math.random() * 52 );
208
209            temp = deck[ i ];
210            deck[ i ] = deck[ index ];
211            deck[ index ] = temp;
212        }
213    }
214
215    // método para limpar a tela
216    void clear() {
217        displayField.setText( "" );
218        statusLabel.setText( "" );
219    }
220
221    // método para sair do programa
222    void exit() {
223        System.exit( 0 );
224    }
225
226    // método para gerar uma carta aleatória
227    Card dealCard() {
228        if ( currentCard >= 51 )
229            shuffle();
230
231        currentCard++;
232
233        return deck[ currentCard ];
234    }
235
236    // método para embaralhar o baralho
237    void shuffle() {
238        Card temp;
239
240        for ( int i = 0; i < 52; i++ ) {
241            int index = (int)( Math.random() * 52 );
242
243            temp = deck[ i ];
244            deck[ i ] = deck[ index ];
245            deck[ index ] = temp;
246        }
247    }
248
249    // método para limpar a tela
250    void clear() {
251        displayField.setText( "" );
252        statusLabel.setText( "" );
253    }
254
255    // método para sair do programa
256    void exit() {
257        System.exit( 0 );
258    }
259
260    // método para gerar uma carta aleatória
261    Card dealCard() {
262        if ( currentCard >= 51 )
263            shuffle();
264
265        currentCard++;
266
267        return deck[ currentCard ];
268    }
269
270    // método para embaralhar o baralho
271    void shuffle() {
272        Card temp;
273
274        for ( int i = 0; i < 52; i++ ) {
275            int index = (int)( Math.random() * 52 );
276
277            temp = deck[ i ];
278            deck[ i ] = deck[ index ];
279            deck[ index ] = temp;
280        }
281    }
282
283    // método para limpar a tela
284    void clear() {
285        displayField.setText( "" );
286        statusLabel.setText( "" );
287    }
288
289    // método para sair do programa
290    void exit() {
291        System.exit( 0 );
292    }
293
294    // método para gerar uma carta aleatória
295    Card dealCard() {
296        if ( currentCard >= 51 )
297            shuffle();
298
299        currentCard++;
300
301        return deck[ currentCard ];
302    }
303
304    // método para embaralhar o baralho
305    void shuffle() {
306        Card temp;
307
308        for ( int i = 0; i < 52; i++ ) {
309            int index = (int)( Math.random() * 52 );
310
311            temp = deck[ i ];
312            deck[ i ] = deck[ index ];
313            deck[ index ] = temp;
314        }
315    }
316
317    // método para limpar a tela
318    void clear() {
319        displayField.setText( "" );
320        statusLabel.setText( "" );
321    }
322
323    // método para sair do programa
324    void exit() {
325        System.exit( 0 );
326    }
327
328    // método para gerar uma carta aleatória
329    Card dealCard() {
330        if ( currentCard >= 51 )
331            shuffle();
332
333        currentCard++;
334
335        return deck[ currentCard ];
336    }
337
338    // método para embaralhar o baralho
339    void shuffle() {
340        Card temp;
341
342        for ( int i = 0; i < 52; i++ ) {
343            int index = (int)( Math.random() * 52 );
344
345            temp = deck[ i ];
346            deck[ i ] = deck[ index ];
347            deck[ index ] = temp;
348        }
349    }
350
351    // método para limpar a tela
352    void clear() {
353        displayField.setText( "" );
354        statusLabel.setText( "" );
355    }
356
357    // método para sair do programa
358    void exit() {
359        System.exit( 0 );
360    }
361
362    // método para gerar uma carta aleatória
363    Card dealCard() {
364        if ( currentCard >= 51 )
365            shuffle();
366
367        currentCard++;
368
369        return deck[ currentCard ];
370    }
371
372    // método para embaralhar o baralho
373    void shuffle() {
374        Card temp;
375
376        for ( int i = 0; i < 52; i++ ) {
377            int index = (int)( Math.random() * 52 );
378
379            temp = deck[ i ];
380            deck[ i ] = deck[ index ];
381            deck[ index ] = temp;
382        }
383    }
384
385    // método para limpar a tela
386    void clear() {
387        displayField.setText( "" );
388        statusLabel.setText( "" );
389    }
390
391    // método para sair do programa
392    void exit() {
393        System.exit( 0 );
394    }
395
396    // método para gerar uma carta aleatória
397    Card dealCard() {
398        if ( currentCard >= 51 )
399            shuffle();
400
401        currentCard++;
402
403        return deck[ currentCard ];
404    }
405
406    // método para embaralhar o baralho
407    void shuffle() {
408        Card temp;
409
410        for ( int i = 0; i < 52; i++ ) {
411            int index = (int)( Math.random() * 52 );
412
413            temp = deck[ i ];
414            deck[ i ] = deck[ index ];
415            deck[ index ] = temp;
416        }
417    }
418
419    // método para limpar a tela
420    void clear() {
421        displayField.setText( "" );
422        statusLabel.setText( "" );
423    }
424
425    // método para sair do programa
426    void exit() {
427        System.exit( 0 );
428    }
429
430    // método para gerar uma carta aleatória
431    Card dealCard() {
432        if ( currentCard >= 51 )
433            shuffle();
434
435        currentCard++;
436
437        return deck[ currentCard ];
438    }
439
440    // método para embaralhar o baralho
441    void shuffle() {
442        Card temp;
443
444        for ( int i = 0; i < 52; i++ ) {
445            int index = (int)( Math.random() * 52 );
446
447            temp = deck[ i ];
448            deck[ i ] = deck[ index ];
449            deck[ index ] = temp;
450        }
451    }
452
453    // método para limpar a tela
454    void clear() {
455        displayField.setText( "" );
456        statusLabel.setText( "" );
457    }
458
459    // método para sair do programa
460    void exit() {
461        System.exit( 0 );
462    }
463
464    // método para gerar uma carta aleatória
465    Card dealCard() {
466        if ( currentCard >= 51 )
467            shuffle();
468
469        currentCard++;
470
471        return deck[ currentCard ];
472    }
473
474    // método para embaralhar o baralho
475    void shuffle() {
476        Card temp;
477
478        for ( int i = 0; i < 52; i++ ) {
479            int index = (int)( Math.random() * 52 );
480
481            temp = deck[ i ];
482            deck[ i ] = deck[ index ];
483            deck[ index ] = temp;
484        }
485    }
486
487    // método para limpar a tela
488    void clear() {
489        displayField.setText( "" );
490        statusLabel.setText( "" );
491    }
492
493    // método para sair do programa
494    void exit() {
495        System.exit( 0 );
496    }
497
498    // método para gerar uma carta aleatória
499    Card dealCard() {
500        if ( currentCard >= 51 )
501            shuffle();
502
503        currentCard++;
504
505        return deck[ currentCard ];
506    }
507
508    // método para embaralhar o baralho
509    void shuffle() {
510        Card temp;
511
512        for ( int i = 0; i < 52; i++ ) {
513            int index = (int)( Math.random() * 52 );
514
515            temp = deck[ i ];
516            deck[ i ] = deck[ index ];
517            deck[ index ] = temp;
518        }
519    }
520
521    // método para limpar a tela
522    void clear() {
523        displayField.setText( "" );
524        statusLabel.setText( "" );
525    }
526
527    // método para sair do programa
528    void exit() {
529        System.exit( 0 );
530    }
531
532    // método para gerar uma carta aleatória
533    Card dealCard() {
534        if ( currentCard >= 51 )
535            shuffle();
536
537        currentCard++;
538
539        return deck[ currentCard ];
540    }
541
542    // método para embaralhar o baralho
543    void shuffle() {
544        Card temp;
545
546        for ( int i = 0; i < 52; i++ ) {
547            int index = (int)( Math.random() * 52 );
548
549            temp = deck[ i ];
550            deck[ i ] = deck[ index ];
551            deck[ index ] = temp;
552        }
553    }
554
555    // método para limpar a tela
556    void clear() {
557        displayField.setText( "" );
558        statusLabel.setText( "" );
559    }
560
561    // método para sair do programa
562    void exit() {
563        System.exit( 0 );
564    }
565
566    // método para gerar uma carta aleatória
567    Card dealCard() {
568        if ( currentCard >= 51 )
569            shuffle();
570
571        currentCard++;
572
573        return deck[ currentCard ];
574    }
575
576    // método para embaralhar o baralho
577    void shuffle() {
578        Card temp;
579
580        for ( int i = 0; i < 52; i++ ) {
581            int index = (int)( Math.random() * 52 );
582
583            temp = deck[ i ];
584            deck[ i ] = deck[ index ];
585            deck[ index ] = temp;
586        }
587    }
588
589    // método para limpar a tela
590    void clear() {
591        displayField.setText( "" );
592        statusLabel.setText( "" );
593    }
594
595    // método para sair do programa
596    void exit() {
597        System.exit( 0 );
598    }
599
600    // método para gerar uma carta aleatória
601    Card dealCard() {
602        if ( currentCard >= 51 )
603            shuffle();
604
605        currentCard++;
606
607        return deck[ currentCard ];
608    }
609
610    // método para embaralhar o baralho
611    void shuffle() {
612        Card temp;
613
614        for ( int i = 0; i < 52; i++ ) {
615            int index = (int)( Math.random() * 52 );
616
617            temp = deck[ i ];
618            deck[ i ] = deck[ index ];
619            deck[ index ] = temp;
620        }
621    }
622
623    // método para limpar a tela
624    void clear() {
625        displayField.setText( "" );
626        statusLabel.setText( "" );
627    }
628
629    // método para sair do programa
630    void exit() {
631        System.exit( 0 );
632    }
633
634    // método para gerar uma carta aleatória
635    Card dealCard() {
636        if ( currentCard >= 51 )
637            shuffle();
638
639        currentCard++;
640
641        return deck[ currentCard ];
642    }
643
644    // método para embaralhar o baralho
645    void shuffle() {
646        Card temp;
647
648        for ( int i = 0; i < 52; i++ ) {
649            int index = (int)( Math.random() * 52 );
650
651            temp = deck[ i ];
652            deck[ i ] = deck[ index ];
653            deck[ index ] = temp;
654        }
655    }
656
657    // método para limpar a tela
658    void clear() {
659        displayField.setText( "" );
660        statusLabel.setText( "" );
661    }
662
663    // método para sair do programa
664    void exit() {
665        System.exit( 0 );
666    }
667
668    // método para gerar uma carta aleatória
669    Card dealCard() {
670        if ( currentCard >= 51 )
671            shuffle();
672
673        currentCard++;
674
675        return deck[ currentCard ];
676    }
677
678    // método para embaralhar o baralho
679    void shuffle() {
680        Card temp;
681
682        for ( int i = 0; i < 52; i++ ) {
683            int index = (int)( Math.random() * 52 );
684
685            temp = deck[ i ];
686            deck[ i ] = deck[ index ];
687            deck[ index ] = temp;
688        }
689    }
690
691    // método para limpar a tela
692    void clear() {
693        displayField.setText( "" );
694        statusLabel.setText( "" );
695    }
696
697    // método para sair do programa
698    void exit() {
699        System.exit( 0 );
700    }
701
702    // método para gerar uma carta aleatória
703    Card dealCard() {
704        if ( currentCard >= 51 )
705            shuffle();
706
707        currentCard++;
708
709        return deck[ currentCard ];
710    }
711
712    // método para embaralhar o baralho
713    void shuffle() {
714        Card temp;
715
716        for ( int i = 0; i < 52; i++ ) {
717            int index = (int)( Math.random() * 52 );
718
719            temp = deck[ i ];
720            deck[ i ] = deck[ index ];
721            deck[ index ] = temp;
722        }
723    }
724
725    // método para limpar a tela
726    void clear() {
727        displayField.setText( "" );
728        statusLabel.setText( "" );
729    }
730
731    // método para sair do programa
732    void exit() {
733        System.exit( 0 );
734    }
735
736    // método para gerar uma carta aleatória
737    Card dealCard() {
738        if ( currentCard >= 51 )
739            shuffle();
740
741        currentCard++;
742
743        return deck[ currentCard ];
744    }
745
746    // método para embaralhar o baralho
747    void shuffle() {
748        Card temp;
749
750        for ( int i = 0; i < 52; i++ ) {
751            int index = (int)( Math.random() * 52 );
752
753            temp = deck[ i ];
754            deck[ i ] = deck[ index ];
755            deck[ index ] = temp;
756        }
757    }
758
759    // método para limpar a tela
760    void clear() {
761        displayField.setText( "" );
762        statusLabel.setText( "" );
763    }
764
765    // método para sair do programa
766    void exit() {
767        System.exit( 0 );
768    }
769
770    // método para gerar uma carta aleatória
771    Card dealCard() {
772        if ( currentCard >= 51 )
773            shuffle();
774
775        currentCard++;
776
777        return deck[ currentCard ];
778    }
779
780    // método para embaralhar o baralho
781    void shuffle() {
782        Card temp;
783
784        for ( int i = 0; i < 52; i++ ) {
785            int index = (int)( Math.random() * 52 );
786
787            temp = deck[ i ];
788            deck[ i ] = deck[ index ];
789            deck[ index ] = temp;
790        }
791    }
792
793    // método para limpar a tela
794    void clear() {
795        displayField.setText( "" );
796        statusLabel.setText( "" );
797    }
798
799    // método para sair do programa
800    void exit() {
801        System.exit( 0 );
802    }
803
804    // método para gerar uma carta aleatória
805    Card dealCard() {
806        if ( currentCard >= 51 )
807            shuffle();
808
809        currentCard++;
810
811        return deck[ currentCard ];
812    }
813
814    // método para embaralhar o baralho
815    void shuffle() {
816        Card temp;
817
818        for ( int i = 0; i < 52; i++ ) {
819            int index = (int)( Math.random() * 52 );
820
821            temp = deck[ i ];
822            deck[ i ] = deck[ index ];
823            deck[ index ] = temp;
824        }
825    }
826
827    // método para limpar a tela
828    void clear() {
829        displayField.setText( "" );
830        statusLabel.setText( "" );
831    }
832
833    // método para sair do programa
834    void exit() {
835        System.exit( 0 );
836    }
837
838    // método para gerar uma carta aleatória
839    Card dealCard() {
840        if ( currentCard >= 51 )
841            shuffle();
842
843        currentCard++;
844
845        return deck[ currentCard ];
846    }
847
848    // método para embaralhar o baralho
849    void shuffle() {
850        Card temp;
851
852        for ( int i = 0; i < 52; i++ ) {
853            int index = (int)( Math.random() * 52 );
854
855            temp = deck[ i ];
856            deck[ i ] = deck[ index ];
857            deck[ index ] = temp;
858        }
859    }
860
861    // método para limpar a tela
862    void clear() {
863        displayField.setText( "" );
864        statusLabel.setText( "" );
865    }
866
867    // método para sair do programa
868    void exit() {
869        System.exit( 0 );
870    }
871
872    // método para gerar uma carta aleatória
873    Card dealCard() {
874        if ( currentCard >= 51 )
875            shuffle();
876
877        currentCard++;
878
879        return deck[ currentCard ];
880    }
881
882    // método para embaralhar o baralho
883    void shuffle() {
884        Card temp;
885
886        for ( int i = 0; i < 52; i++ ) {
887            int index = (int)( Math.random() * 52 );
888
889            temp = deck[ i ];
890            deck[ i ] = deck[ index ];
891            deck[ index ] = temp;
892        }
893    }
894
895    // método para limpar a tela
896    void clear() {
897        displayField.setText( "" );
898        statusLabel.setText( "" );
899    }
900
901    // método para sair do programa
902    void exit() {
903        System.exit( 0 );
904    }
905
906    // método para gerar uma carta aleatória
907    Card dealCard() {
908        if ( currentCard >= 51 )
909            shuffle();
910
911        currentCard++;
912
913        return deck[ currentCard ];
914    }
915
916    // método para embaralhar o baralho
917    void shuffle() {
918        Card temp;
919
920        for ( int i = 0; i < 52; i++ ) {
921            int index = (int)( Math.random() * 52 );
922
923            temp = deck[ i ];
924            deck[ i ] = deck[ index ];
925            deck[ index ] = temp;
926        }
927    }
928
929    // método para limpar a tela
930    void clear() {
931        displayField.setText( "" );
932        statusLabel.setText( "" );
933    }
934
935    // método para sair do programa
936    void exit() {
937        System.exit( 0 );
938    }
939
940    // método para gerar uma carta aleatória
941    Card dealCard() {
942        if ( currentCard >= 51 )
943            shuffle();
944
945        currentCard++;
946
947        return deck[ currentCard ];
948    }
949
950    // método para embaralhar o baralho
951    void shuffle() {
952        Card temp;
953
954        for ( int i = 0; i < 52; i++ ) {
955            int index = (int)( Math.random() * 52 );
956
957            temp = deck[ i ];
958            deck[ i ] = deck[ index ];
959            deck[ index ] = temp;
960        }
961    }
962
963    // método para limpar a tela
964    void clear() {
965        displayField.setText( "" );
966        statusLabel.setText( "" );
967    }
968
969    // método para sair do programa
970    void exit() {
971        System.exit( 0 );
972    }
973
974    // método para gerar uma carta aleatória
975    Card dealCard() {
976        if ( currentCard >= 51 )
977            shuffle();
978
979        currentCard++;
980
981        return deck[ currentCard ];
982    }
983
984    // método para embaralhar o baralho
985    void shuffle() {
986        Card temp;
987
988        for ( int i = 0; i < 52; i++ ) {
989            int index = (int)( Math.random() * 52 );
990
991            temp = deck[ i ];
992            deck[ i ] = deck[ index ];
993            deck[ index ] = temp;
994        }
995    }
996
997    // método para limpar a tela
998    void clear() {
999        displayField.setText( "" );
1000       statusLabel.setText( "" );
1001   }
1002 }
```

Fig. 10.21 Programa que distribui cartas (parte 2 de 4).

```

64      } // fim da classe interna anônima
65
66  ); // fim da chamada para addActionListener
67
68  container.add( dealButton );
69
70  shuffleButton = new JButton( "Shuffle cards" );
71  shuffleButton.addActionListener(
72
73      // classe interna anônima
74      new ActionListener() {
75
76          // embaralha as cartas
77          public void actionPerformed( ActionEvent actionEvent )
78          {
79              displayField.setText( "SHUFFLING ..." );
80              shuffle();
81              displayField.setText( "DECK IS SHUFFLED" );
82          }
83
84      } // fim da classe interna anônima
85
86  ); // fim da chamada para addActionListener
87
88  container.add( shuffleButton );
89
90  displayField = new JTextField( 20 );
91  displayField.setEditable( false );
92  container.add( displayField );
93
94  statusLabel = new JLabel();
95  container.add( statusLabel );
96
97  setSize( 275, 120 ); // configura o tamanho da janela
98  show(); // mostra a janela
99 }
100
101 // embaralha as cartas com algoritmo de uma só passagem
102 public void shuffle()
103 {
104     currentCard = -1;
105
106     // para cada carta, escolhe aleatoriamente outra carta e as troca
107     for ( int first = 0; first < deck.length; first++ ) {
108         int second = ( int ) ( Math.random() * 52 );
109         Card temp = deck[ first ];
110         deck[ first ] = deck[ second ];
111         deck[ second ] = temp;
112     }
113
114     dealButton.setEnabled( true );
115 }
116
117 // distribui uma carta
118 public Card dealCard()
119 {
120     if ( ++currentCard < deck.length )
121         return deck[ currentCard ];
122     else {
123         dealButton.setEnabled( false );

```

Fig. 10.21 Programa que distribui cartas (parte 3 de 4).

```

124         return null;
125     }
126 }
127
128 // executa o aplicativo
129 public static void main( String args[] )
130 {
131     DeckOfCards app = new DeckOfCards();
132
133     app.addWindowListener(
134
135         // classe interna anônima
136         new WindowAdapter() {
137
138             // termina o aplicativo quando o usuário fecha a janela
139             public void windowClosing( WindowEvent windowEvent )
140             {
141                 System.exit( 0 );
142             }
143
144         } // fim da classe interna anônima
145
146     ); // fim da chamada para addWindowListener
147
148 } // fim do método main
149
150 } // fim da classe DeckOfCards
151
152 // classe que representa uma carta
153 class Card {
154     private String face;
155     private String suit;
156
157     // construtor que inicializa uma carta
158     public Card( String cardFace, String cardSuit )
159     {
160         face = cardFace;
161         suit = cardSuit;
162     }
163
164     // retorna representação de Card como String
165     public String toString()
166     {
167         return face + " of " + suit;
168     }
169
170 } // fim da classe Card

```

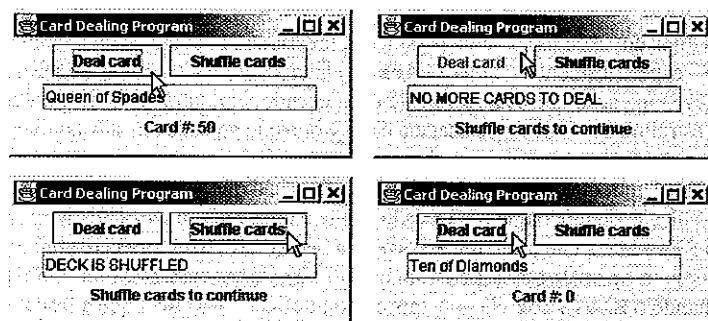


Fig. 10.21 Programa que distribui cartas (parte 4 de 4).

A classe **Card** (linhas 153 a 170) contém duas variáveis de instância **String** – **face** e **suit** – que são utilizadas para armazenar referências ao nome da face (ou valor) e o nome do naipe de uma carta (**Card**) específica. O construtor para a classe recebe dois **Strings** que ele utiliza para inicializar **face** e **suit**. O método **toString** é fornecido para criar um **String** que consiste na **face** da carta, o **string "of"** e o **suit** da carta.

A classe **DeckOfCards** (linhas 11 a 150) consiste em um **array deck** de 52 **Cards**, um inteiro **currentCard** para representar a carta mais recentemente distribuída no **array deck** (-1 se nenhuma carta ainda foi distribuída) e os componentes GUI utilizados para manipular o baralho de cartas. O método construtor instancia o aplicativo do **array deck** (linha 29) e utiliza a estrutura **for** nas linhas 33 a 35 para preencher o **array deck** com **Cards**. Observe que cada **Card** é instanciada e é inicializada com dois **Strings** – um do **array faces** (**Strings** "Ace" a "King") e um do **array suits** ("Hearts", "Diamonds", "Clubs" e "Spades"). O cálculo **count % 13** sempre resulta em um valor de 0 a 12 (os 13 subscritos do **array faces**) e o cálculo **count / 13** sempre resulta em um valor de 0 a 3 (os quatro subscritos no **array suits**). Quando o **array deck** é inicializado, ele contém as cartas com as faces de Ás a Rei ordenadas por naipe.

Quando o usuário clica no botão **Deal card**, o método **actionPerformed** (linhas 48 a 62) invoca o método **dealCard** (definido nas linhas 118 a 126) para obter a próxima carta no **array**. Se o **deck** não estiver vazio, uma referência de objeto **Card** é devolvida; caso contrário, devolve-se **null**. Se a referência não for **null**, as linhas 53 e 54 exibem o **Card** na **JTextField displayField** e o número da carta no **JLabel statusLabel**.

Se a referência retornada por **dealCard** for **null**, o **String** "NO MORE CARDS TO DEAL" é exibido em **JTextField** e o **String** "Shuffle cards to continue" é exibido em **JLabel**.

Quando o usuário clica no botão **Shuffle cards**, o método **actionPerformed** (linhas 77 a 82) invoca o método **shuffle** (definido nas linhas 102 a 115) para embaralhar as cartas. O método percorre as 52 cartas (subscritos de **array** 0 a 51) com um laço. Para cada carta, um número entre 0 e 51 é selecionado aleatoriamente. Em seguida, o objeto **Card** atual e o objeto **Card** aleatoriamente selecionado são trocados no **array**. Um total de apenas 52 trocas é feito em uma única passagem pelo **array** inteiro e o **array** dos objetos **Card** é embaralhado! Quando o embaralhamento está completo, o **String** "DECK IS SHUFFLED" é exibido no **JTextField**.

Repare no uso do método **setEnabled** nas linhas 114 e 123 para ativar e desativar o **dealButton**. Pode-se utilizar o método **setEnabled** em muitos componentes GUI. Quando é chamado com um argumento **false**, o componente GUI pelo qual ele é chamado é desativado e o usuário não pode interagir com ele. Para reativar o botão, o método **setEnabled** é chamado com um argumento **true**.

10.22 (Estudo de caso opcional) Pensando em objetos: tratamento de eventos

Usualmente, os objetos não executam suas operações espontaneamente. Em vez disso, uma operação específica normalmente é invocada quando um objeto remetente (um *objeto cliente*) envia uma mensagem para um objeto destinatário (um *objeto servidor*) pedindo que o objeto destinatário execute aquela operação específica. Nas seções anteriores, mencionamos que os objetos interagem enviando e recebendo mensagens. Começamos a modelar o comportamento de nosso sistema de elevador usando diagramas de mapa de estados e de atividades na Seção 5.11 e diagramas de colaborações na Seção 7.10. Nessa seção, discutimos como interagir os objetos do sistema do elevador.

Eventos

Na Fig. 7.19, apresentamos um exemplo de uma pessoa que pressiona um botão enviando uma mensagem **pressButton** para o botão – especificamente, o objeto **Person** chamou o método **pressButton** do objeto **Button**. Esta mensagem descreve uma ação que está realmente acontecendo; em outras palavras, a **Person** pressiona um **Button**. Em geral, a estrutura do nome da mensagem é um verbo que precede um substantivo – por exemplo, o nome da mensagem **pressButton** consiste no verbo "press" seguido pelo substantivo "button".

O *evento* é uma mensagem que notifica um objeto de uma ação que já aconteceu. Por exemplo, nessa seção, modificamos nossa simulação de modo que o **Elevator** envie um evento **elevatorArrived** para a **Door** do **Elevator** quando o **Elevator** chega em um **Floor**. Na Seção 7.10, o **Elevator** abre essa **Door** diretamente, enviando uma mensagem **openDoor**. Esperar um evento **elevatorArrived** permite que a **Door** determine as ações a tomar quando o **Elevator** chegou, tais como notificar a **Person** que a **Door** se abriu. Isto reforça o princípio de encapsulamento de OOD e modela o mundo real de maneira mais aproximada. Na realidade, a porta – e não o elevador – "notifica" uma pessoa da abertura de uma porta.

Observe que a estrutura de nomeação de eventos é o inverso do primeiro tipo de estrutura de nomeação de mensagens. Por convenção, o nome do evento consiste no substantivo que precede o verbo. Por exemplo, o nome do evento `elevatorArrived` consiste no substantivo “*elevator*” precedendo o verbo “*arrived*”.

Em nossa simulação, criamos uma superclasse chamada `ElevatorModelEvent` (Fig. 10.22) que representa um evento em nosso modelo. `ElevatorModelEvent` contém uma referência `Location` (linha 11) que representa o lugar onde o evento foi gerado e uma referência `Object` (linha 14) para a origem do evento. Em nossa simulação, os objetos usam instâncias de `ElevatorModelEvent` para enviar eventos para outros objetos. Quando um objeto recebe um evento, aquele objeto pode usar o método `getLocation` (linhas 31 a 34) e o método `getSource` (linhas 43 a 46) para determinar o lugar e a origem do evento.

```

1 // ElevatorModelEvent.java
2 // Pacote básico de eventos que guarda o objeto Location
3 package com.deitel.jhttp4.elevator.event;
4
5 // Pacotes Deitel
6 import com.deitel.jhttp4.elevator.model.*;
7
8 public class ElevatorModelEvent {
9
10    // Location que gerou o ElevatorModelEvent
11    private Location location;
12
13    // origem do ElevatorModelEvent gerado
14    private Object source;
15
16    // construtor de ElevatorModelEvent configura a Location
17    public ElevatorModelEvent( Object source,
18        Location location )
19    {
20        setSource( source );
21        setLocation( location );
22    }
23
24    // configura a Location do ElevatorModelEvent
25    public void setLocation( Location eventLocation )
26    {
27        location = eventLocation;
28    }
29
30    // obtém a Location do ElevatorModelEvent
31    public Location getLocation()
32    {
33        return location;
34    }
35
36    // configura a origem do ElevatorModelEvent
37    private void setSource( Object eventSource )
38    {
39        source = eventSource;
40    }
41
42    // obtém a origem do ElevatorModelEvent
43    public Object getSource()
44    {
45        return source;
46    }
47 }
```

Fig. 10.22 A classe `ElevatorModelEvent` é a superclasse para todas as outras classes de eventos em nosso modelo.

Por exemplo, a **Door** pode enviar o **ElevatorModelEvent** para a **Person** quando estiver abrindo ou fechando e o **Elevator** pode enviar um **ElevatorModelEvent** que informa uma pessoa sobre a partida ou a chegada do elevador. Usar objetos diferentes, enviando o mesmo tipo de evento, para descrever ações distintas, pode gerar confusão. Para eliminar a ambigüidade durante nossa discussão de quais eventos são enviados pelos objetos, criamos diversas subclasses **ElevatorModelEvent** na Fig. 10.23, de modo que teremos mais tranquilidade para associar cada evento com seu remetente. De acordo com a Fig. 10.23, as classes **BellEvent**, **PersonMoveEvent**, **LightEvent**, **ButtonEvent**, **ElevatorMoveEvent** e **DoorEvent** são subclasses da classe **ElevatorModelEvent**. Usando estas subclasses de evento, a **Door** envia um evento diferente (um **DoorEvent**) do enviado pelo **Button** (que envia um **ButtonEvent**). A Fig. 10.24 mostra o disparo de ações dos eventos das subclasses. Note que todas as ações na Fig. 10.24 aparecem na forma de “substantivo” + “verbo”.

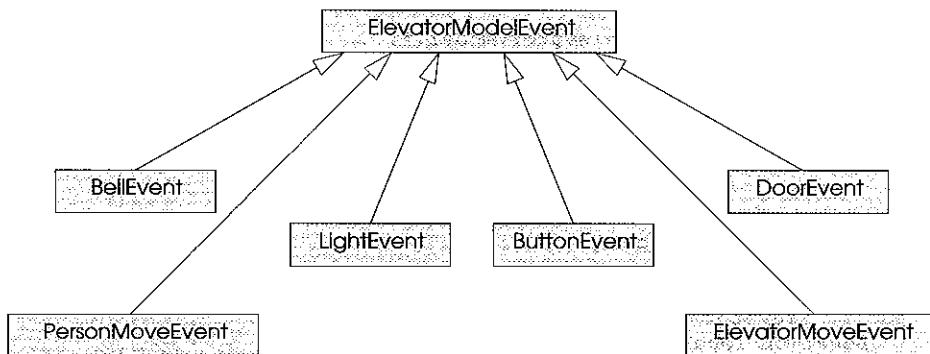


Fig. 10.23 Diagrama de classes que modela a generalização entre **ElevatorModelEvent** e suas subclasses.

Evento	Enviado quando (ação que dispara)	Enviado por objeto da classe
BellEvent	a Bell soou	Bell
ButtonEvent	o Button foi pressionado o Button foi desligado	Button Button
DoorEvent	a Door se abriu a Door se fechou	Door Door
LightEvent	a Light se acendeu a Light se apagou	Light
PersonMoveEvent	uma Person foi criada a Person chegou no Elevator a Person entrou no Elevator a Person saiu do Elevator a Person pressionou um Button a Person saiu da simulação	Person
ElevatorMoveEvent	o Elevator chegou no Floor o Elevator partiu do Floor	Elevator

Fig. 10.24 Disparando ações dos eventos de subclasses de **ElevatorModelEvent**.

Tratamento de eventos

O conceito de tratamento de eventos em Java é similar ao conceito de uma *colaboração*, descrito na Seção 7.10. O tratamento de eventos consiste em um objeto de uma classe enviar uma mensagem particular (que Java chama de um *evento*) para os objetos de outras classes que *esperam aquele tipo de mensagem*¹. A diferença é que os objetos que recebem a mensagem precisam se *registrar* para receber a mensagem; portanto, o tratamento de eventos descreve como um objeto envia um evento para outros objetos que estão “esperando” aquele tipo de evento – aqueles objetos se chamam *ouvintes de eventos* (*event listeners*). Para enviar um evento, o objeto remetente invoca um método particular do objeto destinatário, passando o objeto evento desejado com um parâmetro. Em nossa simulação, este objeto evento pertence a uma classe que estende `ElevatorModelEvent`.

Apresentamos um diagrama de colaborações na Fig. 7.20 que mostra interações de dois objetos `Person` – `waitingPassenger` e `ridingPassenger` – quando eles entram no `Elevator` e saem dele. A Fig. 10.25

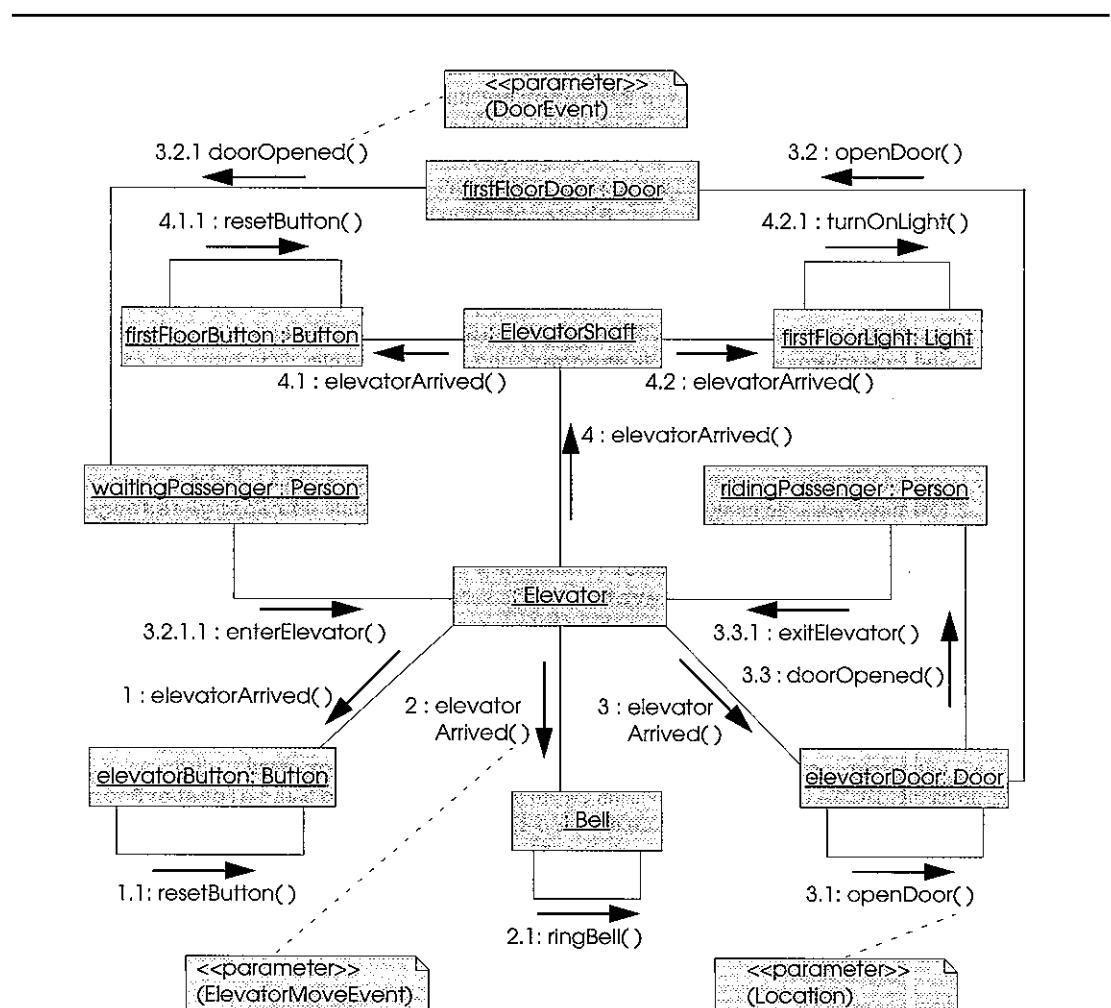


Fig. 10.25 Diagrama de colaborações modificado para passageiros que entram no `Elevator` e saem dele no primeiro Floor.

¹ Tecnicamente, o objeto envia uma notificação de um evento – ou alguma ação de disparo – para outro objeto. Entretanto, o jargão de Java se refere ao envio desta notificação como “envio de um evento”.

mostra um diagrama modificado, que incorpora o tratamento de eventos. Existem três diferenças entre os diagramas. Primeiro, colocamos *anotações* – comentários explicativos sobre alguns dos desenhos no diagrama. A UML representa as anotações como retângulos com os cantos superiores direitos “dobrados”. As anotações na UML são semelhantes a *comentários* em Java. Uma linha tracejada associa uma anotação com qualquer componente da UML (objeto, classe, seta, etc.). Neste diagrama, a notação <>parameter>> especifica que a anotação contém os parâmetros de uma dada mensagem: todos os eventos `doorOpened` passam um objeto `DoorEvent` como parâmetro; todos os eventos `elevatorArrived` passam um objeto `ElevatorMoveEvent`; todas as mensagens `openDoor` passam um objeto `Location`.

A segunda diferença entre os diagramas é que as interações da Fig. 10.25 ocorrem no primeiro `Floor`. Isto permite dar nomes a todos os objetos `Button` e `Door` (`firstFloorDoor` e `firstFloorButton`) para eliminar a ambiguidade, porque as classes `Button` e `Door` têm três objetos em nossa simulação. As interações que ocorrem no segundo `Floor` são idênticas àquelas que ocorrem no primeiro `Floor`.

A diferença mais significativa entre a Fig. 10.25 e a Fig. 7.20 é que o `Elevator` informa aos objetos (através de um evento) sobre uma ação que já aconteceu – o `Elevator` já chegou. Os objetos que recebem o evento então executam alguma ação em resposta ao tipo de mensagem que eles recebem.

De acordo com as mensagens 1, 2, 3 e 4, o `Elevator` executa somente uma ação – ele envia eventos `elevatorArrived` para os objetos interessados em receber aqueles eventos. Especificamente, o objeto `Elevator` envia um `ElevatorMoveEvent` usando o método `elevatorArrived` do objeto receptor. A Fig. 10.25 começa com o `Elevator` enviando um evento `elevatorArrived` para o `elevatorButton`. O `elevatorButton` então desliga a si mesmo (mensagem 1.1). O `Elevator` então envia um evento `elevatorArrived` para a `Bell` (mensagem 2) e a `Bell` invoca seu método `ringBell` adequadamente (isto é, o objeto `Bell` envia para si mesmo uma mensagem `ringBell` na mensagem 2.1).

O `Elevator` envia uma mensagem `elevatorArrived` para a `elevatorDoor` (mensagem 3). A `elevatorDoor` abre a porta invocando seu método `openDoor` (mensagem 3.1). Neste ponto, a `elevatorDoor` está aberta mas ainda não informou o `ridingPassenger` sobre a abertura. Antes de informar ao `ridingPassenger`, a `elevatorDoor` abre a `firstFloorDoor` enviando uma mensagem `openDoor` para a `firstFloorDoor` (mensagem 3.2) – isto garante que o `ridingPassenger` não irá sair antes que a `firstFloorDoor` abra. A `firstFloorDoor` informa ao `waitingPassenger` que a `firstFloorDoor` se abriu (mensagem 3.2.1) e o `waitingPassenger` entra no `Elevator` (mensagem 3.2.1.1). Todas as mensagens aninhadas em 3.2 foram passadas, de modo que a `elevatorDoor` pode informar ao `ridingPassenger` que a `elevatorDoor` se abriu invocando o método `doorOpened` do `ridingPassenger` (mensagem 3.3). O `ridingPassenger` responde saindo do `Elevator` (mensagem 3.3.1)².

Por último, o `Elevator` informa ao `ElevatorShaft` sobre a chegada (mensagem 4). O `ElevatorShaft` informa ao `firstFloorButton` sobre a chegada (mensagem 4.1) e o `firstFloorButton` se desliga (mensagem 4.1.1). O `ElevatorShaft` informa à `firstFloorLight` sobre a chegada (mensagem 4.2) e a `firstFloorLight` se acende (mensagem 4.2.1).

“Ouvintes de eventos”

Demonstramos o tratamento de eventos entre o `Elevator` e o objeto `elevatorDoor` usando o diagrama de colaborações modificado da Fig. 10.25 – o `Elevator` envia um evento `elevatorArrived` para a `elevatorDoor` (mensagem 3). Primeiro, precisamos determinar o objeto evento que o `Elevator` vai passar para a `elevatorDoor`. De acordo com a anotação no canto inferior esquerdo da Fig. 10.25, o `Elevator` passa um objeto `ElevatorMoveEvent` (Fig. 10.26) quando o `Elevator` invoca um método `elevatorArrived`. O diagrama de generalizações da Fig. 10.23 indica que `ElevatorMoveEvent` é uma subclasse de `ElevatorModelEvent`, de modo que `ElevatorMoveEvent` herda as referências `Object` e `Location` de `ElevatorModelEvent`³.

² O problema de o `waitingPassenger` entrar no `Elevator` (mensagem 3.2.1.1) antes de o `ridingPassenger` ter saído (mensagem 3.3.1) permanece em nosso diagrama de colaborações. Na Seção 15.12 de “Pensando em objetos”, mostramos como resolver este problema com o uso de *multithreading*, sincronização e classes ativas.

³ Em nossa simulação, todas as classes de evento têm esta estrutura – isto é, a estrutura da classe `ElevatorMoveEvent` é idêntica à estrutura da classe `DoorEvent`, `ButtonEvent`, etc. Quando você tiver terminado de ler o material nessa seção, recomendamos que você examine a implementação dos eventos no Apêndice G para adquirir uma compreensão melhor da estrutura dos eventos de nosso sistema – as Figuras G.1 a G.7 apresentam o código para os eventos e as Figs. G.8 a G.14 apresentam o código para os “ouvintes de eventos”.

```

1 // ElevatorMoveEvent.java
2 // Indica em que Floor o Elevator chegou ou de qual ele partiu
3 package com.deitel.jhttp4.elevator.event;
4
5 // Pacote Deitel
6 import com.deitel.jhttp4.elevator.model.*;
7
8 public class ElevatorMoveEvent extends ElevatorModelEvent {
9
10    // construtor de ElevatorMoveEvent
11    public ElevatorMoveEvent( Object source, Location location )
12    {
13        super( source, location );
14    }
15 }

```

Fig. 10.26 A classe `ElevatorMoveEvent`, subclasse de `ElevatorModelEvent`, é enviada quando o `Elevator` chegou no `Floor` ou dele partiu.

A `elevatorDoor` precisa implementar uma interface que “espera” um `ElevatorMoveEvent` – isto torna a `elevatorDoor` um ouvinte de evento. A interface `ElevatorMoveListener` (Fig. 10.27) oferece os métodos `elevatorDeparted` (linha 8) e `elevatorArrived` (linha 11), que permitem ao `Elevator` avisar o `ElevatorMoveListener` quando o `Elevator` chegou ou partiu. A interface que oferece os métodos para o ouvinte de evento, como `ElevatorMoveListener`, é chamada de *interface de ouvinte de evento*.

Os métodos `elevatorArrived` e `elevatorDeparted` recebem um objeto `ElevatorMoveEvent` (Fig. 10.26) como argumento. Portanto, quando o `Elevator` “envia um evento `elevatorArrived`” para outro objeto, o `Elevator` passa um objeto `ElevatorMoveEvent` como argumento para o método `elevatorArrived` do objeto receptor. Implementamos a classe `Door` – a classe da qual a `elevatorDoor` é uma instância – no Apêndice H, depois de continuarmos a refinar nosso projeto e aprender mais recursos de Java.

```

1 // ElevatorMoveListener.java
2 // Métodos invocados quando Elevator partiu ou chegou
3 package com.deitel.jhttp4.elevator.event;
4
5 public interface ElevatorMoveListener {
6
7    // invocado quando Elevator partiu
8    public void elevatorDeparted( ElevatorMoveEvent moveEvent );
9
10   // invocado quando Elevator chegou
11   public void elevatorArrived( ElevatorMoveEvent moveEvent );
12 }

```

Fig. 10.27 Interface `ElevatorMoveListener` fornece os métodos necessários para escutar eventos de partida e chegada do `Elevator`.

Diagrama de classes revisitado

A Fig. 10.28 modifica as associações no diagrama de classes da Fig. 9.19 para incluir o tratamento de eventos. Observe que, assim como o diagrama de colaborações da Fig. 10.25, a Fig. 10.28 indica que um objeto informa, ou *signaliza*, a outro objeto que algum evento ocorreu. Se um objeto que recebe o evento invoca um método `private`, o diagrama de classes representa esta invocação de método como uma *auto-associação* – isto é, a classe contém uma associação consigo mesma. As classes `Button`, `Door`, `Light` e `Bell` contêm auto-associações; observe que a associação não inclui uma seta indicando a direção da associação, porque a associação da classe é com ela mesma. Por último, o diagrama inclui uma associação entre a classe `Door` e a classe `Person` (a `Door` informa a uma `Person` que ela abriu), porque estabelecemos o relacionamento entre todos os objetos `Door` e um objeto `Person`.

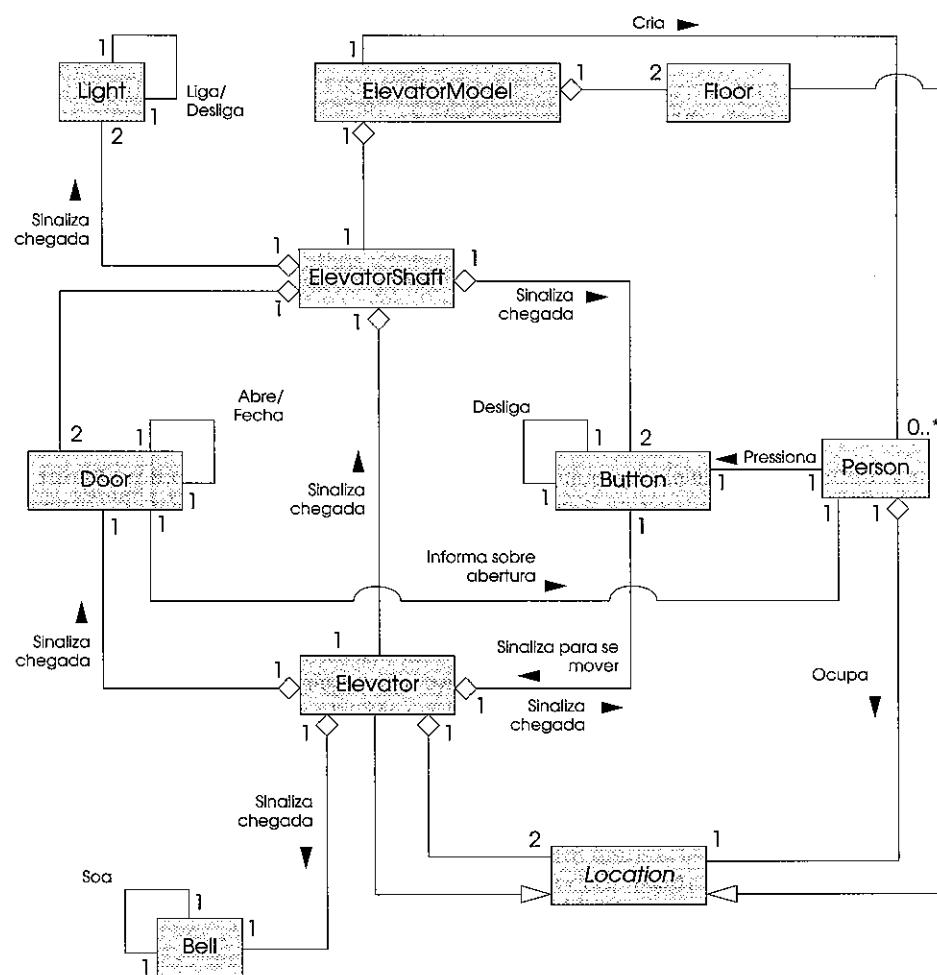


Fig. 10.28 Diagrama de classes de nosso simulador (incluindo o tratamento de eventos).

Resumo

- O valor de uma constante do tipo caractere é seu valor inteiro no conjunto de caracteres Unicode. Os *strings* podem incluir letras, dígitos e caracteres especiais como +, -, *, / e \$. O *string* em Java é um objeto da classe **String**. Os literais *string* ou constantes de *string* são freqüentemente chamados de *objetos anônimos String* e são escritos entre aspas duplas em um programa.
- A classe **String** fornece nove construtores.
- O método **length** de **String** retorna o número de caracteres em um **String**.
- O método **charAt** de **String** retorna o caractere em uma posição específica.
- Utiliza-se o método **equals** para testar dois objetos quanto à igualdade (isto é, se o conteúdo dos dois objetos for idêntico). O método devolve **true** se os objetos forem iguais e **false** caso contrário. O método **equals** utiliza uma *comparação lexicográfica* para **Strings**.
- Quando os valores de tipos primitivos de dados são comparados com **==**, o resultado é **true** se ambos os valores forem idênticos. Quando as referências são comparadas com **==**, o resultado é **true** se ambas as referências se referirem ao *mesmo objeto na memória*.

- Java trata todos os **Strings** anônimos com o mesmo conteúdo como um objeto anônimo **String**.
- O método **equalsIgnoreCase** de **String** realiza uma comparação entre **Strings** que não diferencia maiúsculas e minúsculas.
- O método **compareTo** de **String** devolve 0 se os **Strings** que ele está comparando forem iguais, um número negativo se o **String** que invoca **compareTo** for menor que o **String** que é passado como argumento e um número positivo se o **String** que invoca **compareTo** for maior que o **String** que é passado como argumento. O método **compareTo** utiliza uma comparação lexicográfica.
- O método **regionMatches** de **String** compara as partes de dois **Strings** quanto à igualdade.
- O método **startsWith** de **String** determina se um **String** inicia com os caracteres especificados como argumento. O método **endsWith** de **String** determina se um **String** termina com os caracteres especificados como argumento.
- O método **hashCode** faz um cálculo de código de *hash* que permite que um objeto **String** seja armazenado em uma tabela de *hash*. Esse método é herdado de **Object** e sobreescrito por **String**.
- O método **indexOf** de **String** localiza a primeira ocorrência de um caractere ou de um *substring* em um **String**. O método **lastIndexOf** localiza a última ocorrência de um caractere ou de um *substring* em um **String**.
- O método **substring** de **String** copia e devolve parte de um objeto **String** existente.
- O método **concat** de **String** concatena dois objetos **String** e devolve um novo objeto **String** que contém os caracteres de ambos os **Strings** originais.
- O método **replace** de **String** devolve um novo objeto **String** que substitui cada ocorrência em um **String** do seu primeiro argumento de caractere pelo seu segundo argumento de caractere.
- O método **toUpperCase** de **String** devolve um novo **String** com letras maiúsculas nas posições em que o **String** original tinha letras minúsculas. O método **toLowerCase** devolve um novo **String** com letras minúsculas nas posições em que o **String** original tinha letras maiúsculas.
- O método **trim** de **String** devolve um novo objeto **String** no qual todos os caracteres de espaçamento (como espaços em branco, nova linha e tabulações) foram removidos do início ou do final de um **String**.
- O método **toCharArray** de **String** devolve um novo *array* de caracteres que contém uma cópia dos caracteres em um **String**.
- O método **valueOf** da classe **String** devolve seu argumento convertido em um *string*.
- A primeira vez que o método **intern** de **String** é invocado sobre um **String**, ele devolve uma referência a esse objeto **String**. Invocações subsequentes de **intern** sobre diferentes objetos **String** que têm o mesmo conteúdo que o **String** original resultam em várias referências ao objeto **String** original.
- A classe **StringBuffer** fornece três construtores que permitem que os **StringBuffers** sejam inicializados sem caracteres e tenham uma capacidade inicial de 16 caracteres, sem caracteres e uma capacidade inicial especificada no argumento inteiro, ou com uma cópia dos caracteres do argumento **String** e uma capacidade inicial que é o número de caracteres no argumento **String** mais 16.
- O método **length** de **StringBuffer** devolve o número de caracteres atualmente armazenados em um **StringBuffer**. O método **capacity** devolve o número de caracteres que podem ser armazenados em um **StringBuffer** sem alocar mais memória.
- O método **ensureCapacity** assegura que um **StringBuffer** tenha uma capacidade mínima. O método **setLength** aumenta ou diminui o comprimento de um **StringBuffer**.
- O método **charAt** de **StringBuffer** devolve o caractere no índice especificado. O método **setCharAt** configura o caractere na posição especificada. O método **getChars** devolve um *array* de caracteres que contém uma cópia dos caracteres no **StringBuffer**.
- A classe **StringBuffer** fornece métodos **append** sobrecarregados para adicionar tipos de dados primitivos, *arrays* de caracteres e valores **String** e **Object** ao final de um **StringBuffer**.
- **StringBuffers** e os métodos **append** são utilizados pelo compilador Java na implementação dos operadores + e += para concatenar **Strings**.
- A classe **StringBuffer** fornece os métodos **insert** sobrecarregados para inserir valores de tipos primitivos de dados, *arrays* de caracteres, **String** e **Object** em qualquer posição de um **StringBuffer**.
- A classe **Character** fornece um construtor que recebe um argumento do tipo caractere.
- O método **isDefined** de **Character** determina se um caractere é definido no conjunto de caracteres Unicode. Se ele for, o método devolve **true**; caso contrário, **false**.
- O método **isDigit** de **Character** determina se um caractere é um dígito definido Unicode. Se for, o método devolve **true**; caso contrário, **false**.
- O método **isJavaIdentifierStart** de **Character** determina se um caractere é um caractere que pode ser utilizado como primeiro caractere de um identificador em Java [isto é, uma letra, um sublinhado (_) ou um sinal de cifrão (\$)]. Se puder, o método devolve **true**; caso contrário, **false**.

- O método `isJavaIdentifierPart` de `Character` determina se um caractere é um caractere que pode ser utilizado em um identificador em Java [isto é, um dígito, uma letra, um sublinhado (_) ou um sinal de cifrão (\$)]. Se puder, o método devolve `true`; caso contrário, `false`. O método `isLetter` determina se um caractere é uma letra. Se for, o método devolve `true`; caso contrário, `false`. O método `isLetterOrDigit` determina se um caractere é uma letra ou um dígito. Se for, o método devolve `true`; caso contrário, `false`.
- O método `isLowerCase` de `Character` determina se um caractere é uma letra minúscula. Se for, o método devolve `true`; caso contrário, `false`. O método `isUpperCase` de `Character` determina se um caractere é uma letra maiúscula. Se for, o método devolve `true`; caso contrário, `false`.
- O método `toUpperCase` de `Character` converte um caractere em seu equivalente em letras maiúsculas. O método `toLowerCase` converte um caractere em seu equivalente em letras minúsculas.
- O método `digit` de `Character` converte seu argumento de caractere em um inteiro no sistema numérico especificado por seu argumento inteiro `radix`. O método `forDigit` converte seu argumento inteiro `digit` em um caractere no sistema de numeração especificado por seu argumento inteiro `radix`.
- O método `charValue` de `Character` devolve o `char` armazenado em um objeto `Character`. O método `toString` devolve uma representação `String` de um `Character`.
- O método `hashCode` de `Character` faz o cálculo de um código de `hash` sobre um `Character`.
- O método construtor `default` do `StringTokenizer` cria um `StringTokenizer` para seu argumento `String` que utilizará o `string` delimitador `default " \n\t\r "`, que consiste em um espaço, uma nova linha, uma tabulação e um retorno de carro para separação em `tokens`.
- O método `countTokens` de `StringTokenizer` devolve o número de `tokens` no `String` a ser separado em `tokens`.
- O método `hasMoreTokens` de `StringTokenizer` determina se há mais `tokens` no `String` a ser separado em `tokens`.
- O método `nextToken` de `StringTokenizer` devolve um `String` com o próximo `token`.

Terminologia

acrescentando strings a outros strings
array de strings
caractere constante
caractere de impressão
caracteres de espaço em branco
classe Character
classe String
classe StringBuffer
classe StringTokenizer
código de caractere
comparando strings
comprimento de um string
concatenação
concatenação de strings
conjunto de caracteres
copiando strings
delimitador
dígitos hexadecimais
literal
literal de string
método append da classe StringBuffer
método capacity da classe StringBuffer
método charAt da classe String
método charAt da classe StringBuffer
método charValue da classe Character
método compareTo da classe String
método concat de classe String
método countTokens (StringTokenizer)
método digit da classe Character
método endsWith da classe String
método equals da classe String

método equalsIgnoreCase da classe String
método forDigit da classe Character
método getChars da classe String
método getChars da classe StringBuffer
método hashCode da classe Character
método hashCode da classe String
método hasMoreTokens
método indexOf da classe String
método insert da classe StringBuffer
método intern da classe String
método isDefined da classe Character
método isDigit da classe Character
método isJavaIdentifierPart
método isJavaIdentifierStart
método isLetter da classe Character
método isLetterOrDigit da classe Character
método isLowerCase da classe Character
método isUpperCase da classe Character
método lastIndexOf da classe String
método length da classe String
método length da classe StringBuffer
método nextToken de StringTokenizer
método regionMatches da classe String
método replace da classe String
método setCharAt da classe StringBuffer
método startsWith da classe String
método substring da classe String
método toCharArray da classe String
método toLowerCase da classe Character
método toLowerCase da classe String
método toString da classe Character

<i>método <code>toString</code> da classe <code>String</code></i>	<i>string</i>
<i>método <code>toString</code> da classe <code>StringBuffer</code></i>	<i>string constante</i>
<i>método <code>toUpperCase</code> da classe <code>Character</code></i>	<i>string de pesquisa</i>
<i>método <code>toUpperCase</code> da classe <code>String</code></i>	<i>StringIndexOutOfBoundsException</i>
<i>método <code>trim</code> da classe <code>String</code></i>	<i>strings de “tokenização”</i>
<i>método <code>valueOf</code> da classe <code>String</code></i>	<i>tabela de hash</i>
<i>processamento de string</i>	<i>token</i>
<i>processamento de texto</i>	<i>Unicode</i>
<i>representação numérica do código de um caractere</i>	

Exercícios de auto-revisão

- 10.1** Determine se cada uma das seguintes frases é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- Quando os objetos `String` são comparados com `==`, o resultado é `true` se os `Strings` contiverem os mesmos valores.
 - Um `String` pode ser modificado depois de criado.
- 10.2** Para cada uma das seguintes frases, escreva uma única instrução que realiza a tarefa indicada.
- Comparar o *string* em `s1` com o *string* em `s2` quanto à igualdade de conteúdo.
 - Acrescentar o *string* `s2` ao *string* `s1` utilizando `+=`.
 - Determinar o comprimento do *string* em `s1`.

Respostas aos exercícios de auto-revisão

- 10.1** a) Falso. Os objetos `String` que são comparados como operador `==` são realmente comparados para determinar se eles são o mesmo objeto na memória.
 b) Falso. Os objetos `String` são constantes e não podem ser modificados depois de criados. Os objetos `StringBuffer` podem ser modificados depois de criados.
- 10.2** a) `s1.equals(s2)`
 b) `s1 += s2;`
 c) `s1.length()`

Exercícios

Os Exercícios 10.3 a 10.6 são razoavelmente desafiadores. Se você resolver esses problemas, deverá ser capaz de implementar facilmente os jogos de cartas mais populares.

- 10.3** Modifique o programa da Fig. 10.21 de modo que o método de distribuição de cartas distribua uma mão de pôquer de cinco cartas. Depois, escreva os seguintes métodos adicionais:
- Determinar se a mão contém um par.
 - Determinar se a mão contém dois pares.
 - Determinar se a mão contém uma trinca (por exemplo, três valetes).
 - Determinar se a mão contém quadra (por exemplo, quatro ases).
 - Determinar se a mão contém um *flush* (isto é, todas as cinco cartas do mesmo naipe).
 - Determinar se a mão contém um *straight* (isto é, cinco cartas de valores consecutivos).
 - Determinar se a mão contém um *full house* (isto é, duas cartas de um valor e três cartas de outro valor).
- 10.4** Utilize os métodos desenvolvidos no Exercício 10.3 para escrever um programa que distribui duas mãos de pôquer de cinco cartas, avalia cada mão e determina qual é a melhor.
- 10.5** Modifique o programa desenvolvido no Exercício 10.4 de modo que você possa simular o distribuidor de cartas. A mão de cinco cartas do distribuidor de cartas é distribuída com a face voltada para baixo, de maneira que o jogador não possa vê-la. O programa deve avaliar a mão do distribuidor de cartas e, com base na qualidade da mão, o distribuidor de cartas deve distribuir mais uma, duas ou três cartas para substituir o número correspondente de cartas desnecessárias na mão original. O programa deve reavaliar a mão do distribuidor de cartas. (*Atenção:* esse é um problema difícil!)
- 10.6** Modifique o programa desenvolvido no Exercício 10.5, de modo que você possa tratar a mão do distribuidor de cartas automaticamente, mas permitindo ao jogador escolher quais cartas ele quer substituir. O programa deve avaliar ambas as mãos e determinar quem ganha. Agora utilize esse novo programa para disputar 20 rodadas contra o computador.

Quem ganha mais rodadas, você ou o computador? Faça um de seus amigos disputar 20 rodadas contra o computador. Quem ganha mais? Com base nos resultados dessas rodadas, faça modificações apropriadas para refinar o programa de pôquer (também é um problema difícil). Dispute mais 20 rodadas. O programa modificado joga uma rodada melhor?

10.7 Escreva um aplicativo que utiliza o método `compareTo` de `String` para comparar dois `strings` fornecidos como entrada pelo usuário. Crie uma saída informando se primeiro `string` é menor que, igual a ou maior que o segundo.

10.8 Escreva um aplicativo que utiliza o método `regionMatches` de `String` para comparar dois `strings` fornecidos como entrada pelo usuário. O programa deve inserir o número de caracteres que será comparado e o índice inicial da comparação. O programa deve declarar se o primeiro `string` é menor que, igual a ou maior que o segundo `string`. Ignore a distinção entre maiúsculas e minúsculas dos caracteres ao realizar a comparação.

10.9 Escreva um programa que usa a geração de números aleatórios para criar frases. O programa deve usar quatro `arrays` de ponteiros do tipo `char`, chamados `artigo`, `substantivo`, `verbo` e `preposicao`. O programa deve criar uma sentença selecionando uma palavra ao acaso de cada `array`, na seguinte ordem: `artigo`, `substantivo`, `verbo`, `preposicao`, `artigo` e `substantivo`. À medida que cada palavra é escolhida, ela deve ser concatenada com as palavras precedentes em um `array` grande o bastante para conter toda a frase. As palavras devem ser separadas por espaços. Quando a frase final for mostrada na saída, ela deve começar com uma letra maiúscula e terminar com um ponto final. O programa deve gerar 20 dessas frases.

Os `arrays` devem ser preenchidos como segue: o `array artigo` deve conter os artigos "o", "a", "um", "uma", "algum" e "qualquer"; o `array substantivo` deve conter os substantivos "menino", "menina", "cachorro", "cidade" e "carro"; o `array verbo` deve conter os verbos "dirigiu", "pulou", "correu", "caminhou" e "saltou"; o `array preposicao` deve conter as preposições "para", "de", "acima de", "debaixo de" e "sobre".

Depois de o programa precedente estar escrito e funcionando, modifique-o para produzir uma pequena história consistindo em várias destas frases (que tal a possibilidade de um escritor aleatório de trabalhos de conclusão?!).

10.10 (*Limericks*) Um *limerick* é um poema humorístico de cinco versos em que a primeira e a segunda linha rimam com a quinta, e a terceira linha rima com a quarta. Utilizando técnicas semelhantes àquelas desenvolvidas no Exercício 10.9, escreva um programa Java que produz *limericks* aleatoriamente. Refinar esse programa para produzir bons *limericks* é um problema desafiador, mas o resultado vale o esforço!

10.11 (*Pig Latin*) Escreva um aplicativo que codifica frases da língua inglesa em *Pig Latin*. O *Pig Latin* é uma forma de linguagem codificada freqüentemente utilizada por diversão. Existem muitas variações nos métodos utilizados para formar frases em *Pig Latin*. Para simplificar, utilize o seguinte algoritmo:

Para formar uma frase em *Pig Latin* a partir de uma frase em inglês, separe a frase em palavras com um objeto da classe `StringTokenizer`. Para traduzir cada palavra em inglês para uma palavra do *Pig Latin*, coloque a primeira letra da palavra em inglês no final da palavra e adicione as letras "ay". Assim a palavra "jump" torna-se "umpjay", a palavra "the" torna-se "hetay" e a palavra "computer" torna-se "omputercay". Os espaços entre as palavras permanecem iguais. Suponha o seguinte: a frase em inglês consiste em palavras separadas por espaços, não há nenhuma marcação de pontuação e todas as palavras têm duas ou mais letras. O método `printLatinWord` deve exibir cada palavra. Cada `token` retornado do `nextToken` é passado para o método `printLatinWord` para imprimir o texto em *Pig Latin*. Permita que o usuário insira a frase. Continue exibindo todas as frases convertidas em uma área de texto.

10.12 Escreva um aplicativo que insere um número de telefone como um `string` na forma (555) 555-5555. O programa deve utilizar um objeto da classe `StringTokenizer` para extrair o código de área como um `token`, os três primeiros dígitos do número de telefone como um segundo `token` e os últimos quatro dígitos do número de telefone como um terceiro `token`. Os sete dígitos do número de telefone devem ser concatenados em um `string`. O programa deve converter o `string` do código de área em `int` (lembre-se de `parseInt!`) e converter o `string` do número de telefone em `long`. O código de área e o número de telefone devem ser impressos. Lembre-se de que você que terá de alterar os caracteres delimitadores durante o processo de separação em `tokens`.

10.13 Escreva um aplicativo que insere uma linha de texto, separa a linha em `tokens` com um objeto da classe `StringTokenizer` e envia os `tokens` para a saída na ordem inversa.

10.14 Utilize os métodos de comparação de `string` discutidos e as técnicas para classificar `arrays` desenvolvidas no Capítulo 7 para escrever um programa que ordena alfabeticamente uma lista de `strings`. Permita que o usuário insira os `strings` em um campo de texto. Exiba os resultados em uma área de texto.

10.15 Escreva um aplicativo que recebe entradas de texto e envia o texto para saída em letras minúsculas e em letras maiúsculas.

10.16 Escreva um aplicativo que lê várias linhas de texto e um caractere de pesquisa e utiliza o método `indexOf` de `String` para determinar o número de ocorrências do caractere no texto.

10.17 Escreva um aplicativo baseado no programa do Exercício 10.16 que lê várias linhas de texto e utiliza o método `indexOf` de `String` para determinar o número total de ocorrências de cada letra do alfabeto no texto. As letras minúsculas e maiúsculas devem ser contadas juntas. Armazene os totais para cada letra em um *array* e imprima os valores em formato de tabela depois que os totais foram determinados.

10.18 Escreva um aplicativo que lê uma série de *strings* e gera como saída apenas aqueles *strings* que iniciam com a letra “B”. Os resultados devem sair em uma área de texto.

10.19 Escreva um aplicativo que lê uma série de *strings* e imprime somente aqueles *strings* que terminam com as letras “ED”. Os resultados devem sair em uma área de texto.

10.20 Escreva um aplicativo que lê um código inteiro para um caractere e exibe o caractere correspondente. Modifique esse programa de modo que ele gere todos os códigos de três dígitos possíveis no intervalo 000 a 255 e tente imprimir os caracteres correspondentes. Exiba os resultados em uma área de texto.

10.21 Escreva suas próprias versões dos métodos de `String` para pesquisar *strings*.

10.22 Escreva um programa que lê uma palavra de cinco letras fornecida pelo usuário e produz todas as possíveis palavras de três letras que podem ser derivadas das letras da palavra de cinco letras. Por exemplo, em inglês, as palavras de três letras produzidas a partir da palavra “bathe” incluem as palavras comumente utilizadas “ate”, “bat”, “bet”, “tab”, “hat”, “the” e “tea.”

Seção especial: exercícios de manipulação avançada de strings

Os exercícios precedentes são voltados para o texto e projetados para testar o entendimento do leitor de conceitos fundamentais de manipulação de *strings*. Esta seção inclui uma coleção de exercícios de manipulação de *strings* avançados e intermediários. O leitor deve achar esses problemas desafiadores, mas divertidos. Os problemas variam consideravelmente em dificuldade. Alguns tomam uma hora ou duas para escrever e implementar o programa. Outros são úteis para exercícios de laboratório que talvez exijam duas ou três semanas de estudo e implementação. Alguns são projetos desafiadores de conclusão de curso.

10.23 (*Análise de texto*) A disponibilidade de computadores com capacidade de manipulação de *strings* resultou em algumas abordagens bastante interessantes para analisar textos de grandes autores. Deu-se muita atenção à polêmica de que William Shakespeare não teria existido de fato. Alguns especialistas acreditam que há evidências substanciais para indicar que Christopher Marlowe ou outros autores foram quem realmente escreveu as obras-primas atribuídas a Shakespeare. Os pesquisadores estão utilizando computadores para encontrar semelhanças na escrita desses dois autores. Esse exercício examina três métodos para analisar textos com um computador.

- Escreva um aplicativo que lê várias linhas de texto do teclado e imprime uma tabela que indica o número de ocorrências de cada letra do alfabeto no texto. Por exemplo, a frase

To be, or not to be: that is the question:

Contém um “a”, dois “b”, nenhum “c”, etc.

- Escreva um aplicativo que lê várias linhas de texto e imprime uma tabela que indica o número de palavras de uma letra, palavras de duas letras, palavras de três letras, etc. que aparecem no texto. Por exemplo, a Fig. 10.29 mostra a contagem para a frase

whether 'tis nobler in the mind to suffer

Comprimento da palavra	Ocorrências
1	0
2	2
3	1
4	2 (incluindo 'tis)
5	0
6	2
7	1

Fig. 10.29 Contagem para o *string* "Whether 'tis nobler in the mind to suffer".

- c) Escreva um aplicativo que lê várias linhas de texto e imprime uma tabela que indica o número de ocorrências de cada palavra diferente no texto. A primeira versão de seu programa deve incluir as palavras na tabela na mesma ordem em que elas aparecem no texto. Por exemplo, as linhas

```
To be, or not to be: that is the question:  
Whether 'tis nobler in the mind to suffer
```

Contém a palavra "to" três vezes, a palavra "be" duas vezes, a palavra "or" uma vez, etc. Depois, você deve tentar uma impressão mais interessante (e útil) em que as palavras são classificadas alfabeticamente.

10.24 (*Imprimindo datas em vários formatos*) As datas são impressas em vários formatos comuns. Dois dos formatos mais comuns são

25/04/1955 e 25 de abril de 1955

Escreva um aplicativo que lê uma data no primeiro formato e imprime essa data no segundo formato.

10.25 (*Proteção de cheque*) Os computadores freqüentemente empregam sistemas de verificação de escrita em aplicativos de folha de pagamento e contas a pagar. Circulam muitas histórias estranhas relacionadas com cheques de pagamento de salário semanal impressos (por equívoco) com quantias acima de US\$ 1 milhão. Essas quantidades incorretas são impressas por sistemas computadorizados de preenchimento de cheque por causa de erro humano e/ou falha de máquina. Os projetistas de sistemas colocam controles em seus sistemas para evitar a emissão desses cheques errados.

Outro problema sério é a alteração intencional de um valor do cheque por alguém que pretende receber um cheque de forma fraudulenta. Para evitar que uma quantia em dinheiro seja alterada, a maioria dos sistemas computadorizados de preenchimento de cheque emprega uma técnica chamada *proteção de cheque*.

Os cheques impressos por computador contém um número fixo de espaços em que o computador pode imprimir uma quantia. Suponha que um cheque de pagamento contenha oito espaços em branco em que o computador deve imprimir o valor de um cheque de pagamento semanal. Se o valor for grande, todos os oito espaços serão preenchidos, por exemplo:

```
1,230.60 (valor do cheque)  
-----  
12345678 (posição dos números)
```

Por outro lado, se o valor for menor que US\$ 1000, vários dos espaços seriam comumente deixados em branco. Por exemplo,

```
99.87  
-----  
12345678
```

contém três espaços em branco. Se um cheque é impresso com espaços em branco, é mais fácil para alguém alterar o valor do cheque. Para evitar que um cheque seja alterado, muitos sistemas de preenchimento de cheque inserem *asteriscos no início* para proteger o valor, como segue:

```
***99.87  
-----  
12345678
```

Escreva um aplicativo que lê uma quantia em dinheiro que será impressa em um cheque e depois imprime o valor em formato de cheque protegido com asteriscos se necessário. Suponha que nove espaços estão disponíveis para imprimir o valor.

10.26 (*Valor de um cheque por extenso*) Continuando a discussão do exercício anterior, reiteramos a importância de se projetar sistemas de preenchimento de cheque para evitar a alteração do valor do cheque. Um método comum de segurança exige que o valor do cheque seja escrito em números e "por extenso" também. Mesmo se alguém for capaz de alterar o valor numérico do cheque, é extremamente difícil alterar o valor por extenso.

Muitos sistemas computadorizados de preenchimento de cheque não imprimem o valor do cheque por extenso. Talvez a razão principal dessa omissão é o fato que a maioria das linguagens de alto nível utilizadas em aplicativos comerciais não contém recursos adequados para manipulação de *strings*. Outra razão é que a lógica para o valor por extenso é pouco compreendida.

Escreva um aplicativo que lê o valor numérico do cheque e escreve as palavras equivalentes ao valor. Por exemplo, o valor 112,43 deve ser escrito assim

Cento e doze reais e quarenta e três centavos

10.27 (Código Morse) Talvez o mais famoso de todos os esquemas de codificação seja o código Morse, desenvolvido por Samuel Morse em 1832 para ser utilizado com o sistema de telegrafo. O código Morse associa uma série de pontos e traços para cada letra do alfabeto, para cada dígito e para alguns caracteres especiais (como ponto, vírgula, dois-pontos e ponto-e-vírgula). Em sistemas orientados para áudio, o ponto representa um som curto e o traço representa um som longo. Outras representações de pontos e traços são utilizadas com sistemas baseados em sinais luminosos e sistemas baseados em sinais de bandeira.

A separação entre palavras é indicada por um espaço, ou, simplesmente, a ausência de um ponto ou traço. Em um sistema baseado em áudio, o espaço é indicado por um período breve de tempo durante o qual não se transmite nenhum som. A versão internacional do código Morse aparece na Fig. 10.30.

Escreva um aplicativo que lê uma frase em inglês e codifica a frase em código Morse. Além disso, escreva um programa que lê uma frase em código Morse e a converte na frase em inglês equivalente. Utilize um espaço em branco entre cada letra codificada em Morse e três espaços em branco entre cada palavra codificada em Morse.

Caractere	Código	Caractere	Código
A	.-	T	-
B	-...	U	.-.
C	-.-.	V	...-
D	-..	W	.--
E	.	X	-...-
F	...-.	Y	-.--
G	--.	Z	---.
H		
I	..	Dígitos	
J	.---	1	.---
K	-.-	2	..---
L	-.-.	3	...--
M	--	4-
N	-.	5
O	---	6	-.....
P	.-.-.	7	-....
Q	--.-	8	----.
R	-.-	9	-----
S	...	0	-----

Fig. 10.30 As letras do alfabeto segundo o código Morse internacional.

10.28 (Um programa de conversão métrica) Escreva um aplicativo que ajudará o usuário com conversões métricas. O programa deve permitir que o usuário especifique os nomes das unidades como *strings* (isto é, centímetros, litros, gramas, etc. para o sistema métrico e polegadas, quartos, libras etc. para o sistema inglês) e deve responder a perguntas simples como

"Quantas polegadas há em 2 metros?"
"Quantos litros há em 10 quartos?"

O programa deve reconhecer conversões inválidas. Por exemplo, a pergunta

"Quantos pés há em 5 quilos?"

não é uma pergunta significativa porque "pés" é uma unidade de comprimento enquanto "quilos" é uma unidade de massa.

Seção especial: projetos desafiadores de manipulação de strings

10.29 (*Projeto: um corretor ortográfico*) Muitos pacotes populares de *software* processador de texto têm verificadores ortográficos incorporados.

Neste projeto, pede-se que você desenvolva seu próprio utilitário de verificação ortográfica. Fazemos sugestões para ajudá-lo a começar. Depois, você deve considerar a adição de mais recursos. Utilize um dicionário computadorizado (se você tiver acesso a um) como fonte de palavras.

Por que digitamos tantas palavras com ortografia incorreta? Em alguns casos, isso ocorre porque simplesmente não conhecemos a ortografia correta, então fazemos nossa “melhor suposição”. Em alguns casos, é porque transponemos duas letras (por exemplo, “pardão” em vez de “padrão”). De vez em quando, digitamos duas vezes uma letra accidentalmente (por exemplo, “útil” em vez de “útil”). Às vezes digitamos uma tecla próxima em vez daquela pretendida (por exemplo, “amiversário” em vez de “aniversário”). E assim por diante.

Projete e implemente um aplicativo de verificador ortográfico em Java. O programa deve manter um *array wordList* de *strings*. Permita que o usuário insira esses *strings*. [Nota: no Capítulo 17 apresentamos processamento de arquivos. Uma vez que você tiver esse recurso, você pode obter as palavras para o verificador ortográfico de um dicionário computadorizado armazenado em um arquivo.]

O programa deve solicitar que o usuário insira uma palavra. Ele então deve pesquisar essa palavra no *array wordList*. Se a palavra estiver presente no *array*, o programa deve imprimir “**A palavra está escrita corretamente**”.

Se a palavra não estiver presente no *array*, o programa deve imprimir “**A palavra não está escrita corretamente**”. Depois, o programa deve tentar localizar outras palavras em *wordList* que talvez sejam a palavra que usuário pretendeu digitar. Por exemplo, você pode tentar todas as transposições simples de letras adjacentes possíveis para descobrir que a palavra “padrão” é uma correspondência direta com uma palavra em *wordList*. Naturalmente, isso implica que seu programa verificará todas as outras transposições simples, como “apdrão”, “adprão”, “adrpão”, “adrápô”, e “adrãop”. Quando você localizar uma nova palavra que corresponde a uma palavra em *wordList*, imprima essa palavra em uma mensagem “**Você quer dizer “padrão”?**”.

Implemente outros testes, como substituir cada letra dupla por uma única letra e algum outro teste que você pode desenvolver para aprimorar o valor de seu verificador ortográfico.

10.30 (*Projeto: um gerador de palavras cruzadas*) A maioria das pessoas já brincou com palavras cruzadas, mas poucos tentaram gerar um jogo de palavras cruzadas. Gerar um jogo de palavras cruzadas é sugerido aqui como um projeto de manipulação de *strings* que requer bastante sofisticação e esforço.

Há muitas questões que o programador deve resolver para fazer funcionar até mesmo o mais simples programa gerador de palavras cruzadas. Por exemplo, como se representa a grade das palavras cruzadas dentro do computador? Deve-se utilizar uma série de *strings* ou utilizar *arrays* bidimensionais?

O programador precisa de uma fonte de palavras (isto é, um dicionário computadorizado) que possa ser consultado diretamente pelo programa. De que forma essas palavras devem ser armazenadas para facilitar as complexas manipulações requeridas pelo programa?

O leitor realmente ambicioso vai querer gerar a parte de “dicas” do quebra-cabeça, em que breves dicas para as palavras “horizontais” e as palavras “verticais” são impressas para o jogador de palavras cruzadas. Simplesmente imprimir uma versão do jogo em branco, não é um problema simples.

Imagens gráficas e Java2D

Objetivos

- Entender contextos gráficos e objetos gráficos.
- Entender e ser capaz de manipular cores.
- Entender e ser capaz de manipular fontes.
- Utilizar métodos **Graphics** para desenhar linhas, retângulos, retângulos com cantos arredondados, retângulos tridimensionais, elipses, arcos e polígonos.
- Ser capaz de utilizar métodos da classe **Graphics2D** da API Java2D para desenhar linhas, retângulos, retângulos com cantos arredondados, elipses, arcos e caminhos genéricos.
- Ser capaz de especificar as características **Paint** e **Stroke** de formas exibidas com **Graphics2D**.

Uma imagem vale dez mil palavras.

Provérbio chinês

Trate a natureza em termos de cilindros, esferas, cones, tudo em perspectiva.

Paul Cezanne

Nada se torna real até ser experimentado – mesmo um provérbio não significa nada para você até sua vida ilustrá-lo.

John Keats

Uma imagem mostra num relance o que precisaria de dezenas de páginas de um livro para ser exposto.

Ivan Sergeyevich



Sumário do capítulo

- 11.1 **Introdução**
- 11.2 **Contextos gráficos e objetos gráficos**
- 11.3 **Controle de cor**
- 11.4 **Controle de fontes**
- 11.5 **Desenhando linhas, retângulos e elipses**
- 11.6 **Desenhando arcos**
- 11.7 **Desenhando polígonos e polilinhas**
- 11.8 **A API Java2D**
- 11.9 **Formas de Java2D**
- 11.10 (Estudo de caso opcional) Pensando em objetos: projetando interfaces com a UML**

[Resumo](#) • [Terminologia](#) • [Exercícios de auto-revisão](#) • [Respostas aos exercícios de auto-revisão](#) •

[Exercícios](#)

11.1 Introdução

Neste capítulo, oferecemos uma visão geral de vários recursos de Java para desenhar formas bidimensionais, controlar cores e controlar fontes. Um dos atrativos iniciais de Java era seu suporte a imagens gráficas que permitia aprimorar visualmente *applets* e aplicativos. Java agora contém muitos recursos mais sofisticados de desenho como parte da *API Java2D*. Este capítulo inicia com uma introdução a vários recursos originais de desenho de Java. Em seguida, apresentamos vários dos novos e mais poderosos recursos do Java2D, como controlar o estilo das linhas utilizadas para desenhar formas e controlar como as formas são preenchidas com cores e padrões.

A Fig. 11.1 mostra uma parte da hierarquia de classes Java que inclui diversas classes gráficas básicas e várias classes e interfaces da API Java2D abordadas neste capítulo. A classe **Color** contém métodos e constantes para manipular cores. A classe **Font** contém métodos e constantes para manipular fontes. A classe **FontMetrics** contém métodos para obter informações sobre fontes. A classe **Polygon** contém métodos para criar polígonos. A classe **Graphics** contém métodos para desenhar *strings*, linhas, retângulos e outras formas. A metade inferior da figura lista várias classes e interfaces da API Java2D. A classe **BasicStroke** ajuda a especificar as características no desenho de linhas. As classes **GradientPaint** e **TexturePaint** ajudam a especificar as características para preencher formas com cores ou padrões. As classes **GeneralPath**, **Arc2D**, **Ellipse2D**, **Line2D**, **Rectangle2D** e **RoundRectangle2D** definem uma variedade de formas em Java2D.

Para começar a desenhar em Java, devemos primeiro entender o sistema de coordenadas de Java (Fig. 11.2), que é um esquema para identificar cada ponto possível na tela. Por *default*, o canto superior esquerdo de um componente GUI (como um *applet* ou uma janela) tem as coordenadas (0, 0). Um par de coordenadas é composto por uma *coordenada x* (a *coordenada horizontal*) e uma *coordenada y* (a *coordenada vertical*). A coordenada x é a distância horizontal para a direita, a partir do canto superior esquerdo. A coordenada y é a distância vertical para baixo, a partir do canto superior esquerdo. O *eixo X* descreve cada coordenada horizontal e o *eixo Y* descreve cada coordenada vertical.

Observação de engenharia de software 11.1



A coordenada superior esquerda (0, 0) de uma janela localiza-se, na verdade, atrás da barra de título da janela. Por essa razão, as coordenadas de desenho devem ser ajustadas para desenhar dentro das bordas da janela. A classe **Container** (uma superclasse de todas as janelas em Java) tem o método **getInsets** que devolve um objeto **Insets** (pacote `java.awt`) para esse propósito. O objeto **Insets** tem quatro membros **public** – **top**, **bottom**, **left** e **right** – que representam o número de pixels de cada borda da janela até a área de desenho efetiva da janela.

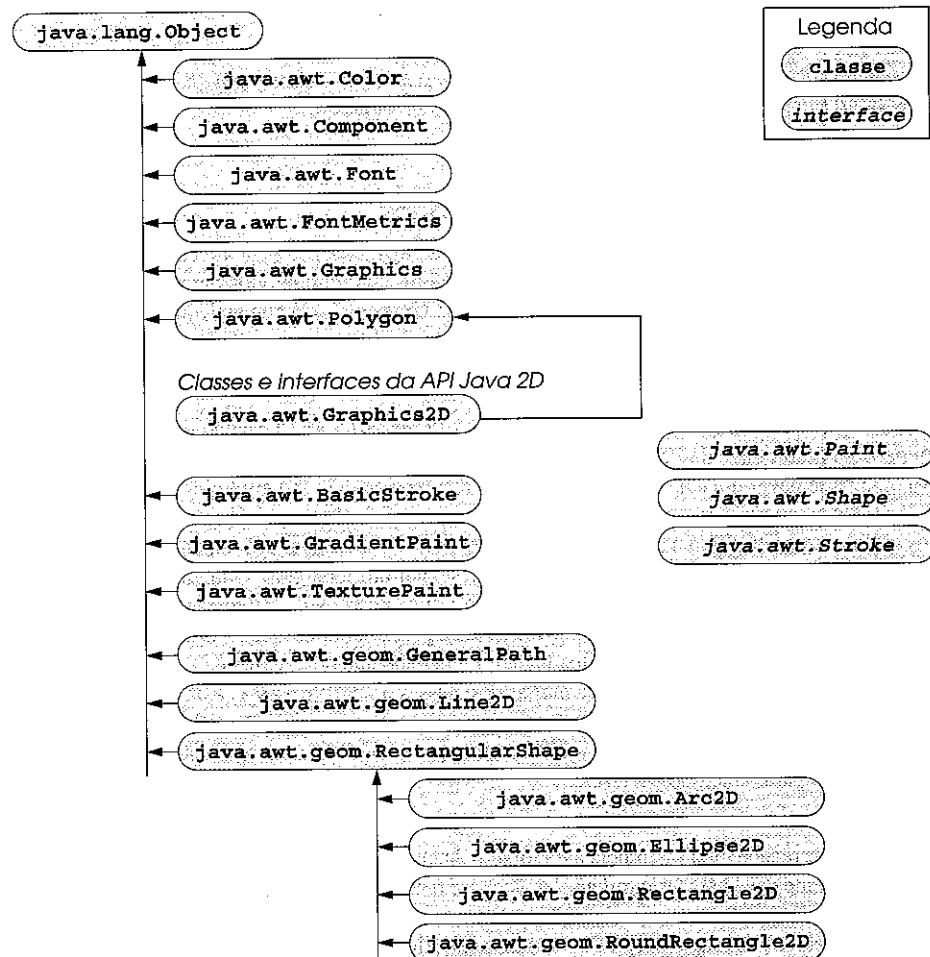


Fig. 11.1 Algumas classes e interfaces utilizadas neste capítulo a partir dos recursos gráficos originais de Java e da API Java2D.

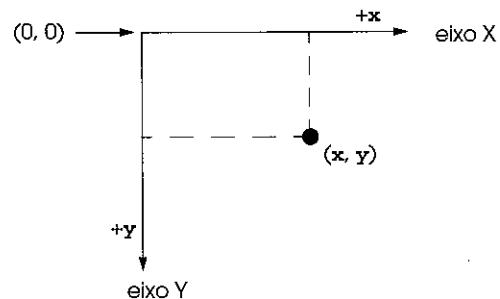


Fig. 11.2 Sistema de coordenadas Java. As unidades são medidas em *pixels*.

O texto e as formas são exibidos na tela especificando-se coordenadas. As unidades de coordenadas são medidas em *pixels*. O *pixel* é a menor unidade de resolução do monitor de vídeo.



Dica de portabilidade 11.1

Monitores diferentes têm resoluções diferentes (isto é, a densidade de pixels varia). Isso pode fazer com que as imagens gráficas pareçam ter tamanhos diferentes em monitores diferentes.

11.2 Contextos gráficos e objetos gráficos

Um *contexto gráfico* Java permite desenhar na tela. O objeto **Graphics** gerencia um contexto gráfico que controla como as informações são desenhadas. Os objetos **Graphics** contêm métodos para desenho, manipulação de fontes, manipulação de cor, entre outros. Todos os *applets* que vimos no texto que realizam desenho na tela utilizam o objeto **Graphics g** (o argumento para o método **paint** do *applet*) para gerenciar o contexto gráfico do *applet*. Neste capítulo, demonstramos o desenho em aplicativos. Entretanto, todas as técnicas mostradas aqui podem ser utilizadas em *applets*.

A classe **Graphics** é uma classe **abstract** (isto é, os objetos **Graphics** não podem ser instanciados). Isso contribui para a portabilidade de Java. Como o desenho é realizado de maneira diferente em cada plataforma que suporta Java, não pode haver uma classe que implemente recursos de desenho em todos os sistemas. Por exemplo, os recursos gráficos que permitem a um PC que roda o Microsoft Windows desenhar um retângulo são diferentes dos recursos gráficos que permitem a uma estação de trabalho UNIX desenhar um retângulo – e ambos são diferentes dos recursos gráficos que permitem a um Macintosh desenhar um retângulo. Quando Java é implementado em cada plataforma, cria-se uma classe derivada de **Graphics** que, na verdade, implementa todos os recursos de desenho. Essa implementação é ocultada de nós pela classe **Graphics**, que fornece a interface que permite escrever programas que utilizam imagens gráficas de uma maneira independente de plataforma.

A classe **Component** é a superclasse para muitas das classes no pacote **java.awt** (discutimos a classe **Component** no Capítulo 12). O método **paint** de **Component** recebe um objeto **Graphics** como argumento. Esse objeto é passado para o método **paint** pelo sistema quando é necessária uma operação de pintura para um **Component**. O cabeçalho para o método **paint** é

```
public void paint( Graphics g )
```

O objeto **Graphics g** recebe uma referência a um objeto da classe derivada de **Graphics** do sistema. O cabeçalho do método precedente deve parecer familiar para você – é o mesmo que utilizamos em nossas classes de *applets*. Na verdade, a classe **Component** é uma classe base indireta da classe **JApplet** – a superclasse de cada *applet* neste livro. Muitos recursos da classe **JApplet** são herdados da classe **Component**. O método **paint**, definido na classe **Component**, nada faz por *default* – ele deve ser sobreescrito pelo programador.

Raramente o método **paint** é chamado diretamente pelo programador, porque desenhar gráficos é um *processo baseado em eventos*. Quando um *applet* é executado, o método **paint** é automaticamente chamado (depois de chamados os métodos **init** e **start** de **JApplet**). Para **paint** ser chamado novamente, um *evento* deve ocorrer (como cobrir e descobrir o *applet* na tela). De maneira semelhante, quando qualquer **Component** é exibido, o método **paint** deste **Component** é chamado.

Se o programador precisa chamar **paint**, faz-se uma chamada ao método **repaint** da classe **Component**. O método **repaint**, logo que possível, faz uma chamada ao método **update** da classe **Component**, para limpar o fundo do **Component** de qualquer desenho anterior, assim **update** chama **paint** diretamente. O método **repaint** é freqüentemente chamado pelo programador para forçar uma operação **paint**. O método **repaint** não deve ser sobreescrito porque realiza algumas tarefas dependentes do sistema. O método **update** é raramente chamado de forma direta e algumas vezes é sobreescrito. Sobreescrivar o método **update** é útil para suavizar animações (isto é, reduzir a “cintilação”) como discutiremos no Capítulo 18. Os cabeçalhos para **repaint** e **update** são

```
public void repaint()
public void update( Graphics g )
```

O método **update** recebe um objeto **Graphics** como argumento, que é fornecido automaticamente pelo sistema quando **update** é chamado.

Neste capítulo, focalizamos o método `paint`. No próximo capítulo, concentrar-nos-emos mais na natureza baseada em eventos dos recursos gráficos e discutiremos os métodos `repaint` e `update` em mais detalhes. Também discutiremos a classe `JComponent` – uma superclasse de muitos componentes GUI do pacote `javax.swing`. As subclasses de `JComponent` em geral pintam a partir de seus métodos `paintComponent`.

11.3 Controle de cor

As cores melhoram a aparência de um programa e ajudam a transmitir um significado. Por exemplo, o semáforo de trânsito tem três luzes de cores diferentes – vermelho significa parar, amarelo significa atenção e verde significa andar.

A classe `Color` define métodos e constantes para manipular cores em um programa Java. As constantes de cor predefinidas estão resumidas na Fig. 11.3 e vários métodos e construtores de cor estão resumidos na Fig. 11.4. Observe que dois dos métodos na Fig. 11.4 são métodos `Graphics` que são específicos para cores.

Cada cor é criada a partir de um componente vermelho, um verde e um azul. Juntos, esses componentes são chamados de *valores RGB*. Todos os três componentes RGB podem ser inteiros no intervalo 0 a 255 ou todas as três partes de RGB podem ser valores em ponto flutuante no intervalo 0.0 a 1.0. A primeira parte do RGB define a quantidade de vermelho, a segunda define a quantidade de verde e a terceira define a quantidade de azul. Quanto maior for o valor de RGB, maior será a quantidade dessa cor em particular. Java permite que o programador escolha entre 256 x 256 x 256 (ou aproximadamente 16,7 milhões de) cores. Entretanto, nem todos os computadores são capazes de exibir todas essas cores. Se esse for o seu caso, a tela do computador exibirá a cor mais próxima possível.

Constante Color	Cor	Valor RGB
<code>public final static Color orange</code>	laranja	255, 200, 0
<code>public final static Color pink</code>	cor-de-rosa	255, 175, 175
<code>public final static Color cyan</code>	ciano	0, 255, 255
<code>public final static Color magenta</code>	magenta	255, 0, 255
<code>public final static Color yellow</code>	amarelo	255, 255, 0
<code>public final static Color black</code>	preto	0, 0, 0
<code>public final static Color white</code>	branco	255, 255, 255
<code>public final static Color gray</code>	cinza	128, 128, 128
<code>public final static Color lightGray</code>	cinza-claro	192, 192, 192
<code>public final static Color darkGray</code>	cinza-escuro	64, 64, 64
<code>public final static Color red</code>	vermelho	255, 0, 0
<code>public final static Color green</code>	verde	0, 255, 0
<code>public final static Color blue</code>	azul	0, 0, 255

Fig. 11.3 Constantes `static` da classe `Color` e valores de RGB.

Método	Descrição
<code>public Color(int r, int g, int b)</code>	Cria uma cor com base no conteúdo de vermelho, verde e azul, expressos como inteiros entre 0 e 255.
<code>public Color(float r, float g, float b)</code>	Cria uma cor com base no conteúdo de vermelho, verde e azul, expressos como valores de ponto flutuante de 0.0 a 1.0.

Fig. 11.4 Métodos `Color` e métodos `Graphics` relacionados com cores (parte 1 de 2).

Método	Descrição
<code>public int getRed() // classe color</code>	Devolve um valor entre 0 e 255 que representa conteúdo de vermelho.
<code>public int getGreen() // classecolor</code>	Devolve um valor entre 0 e 255 que representa o conteúdo de verde.
<code>public int getBlue() // classe color</code>	Devolve um valor entre 0 e 255 que representa o conteúdo de azul.
<code>public Color getColor() // classe Graphic</code>	Devolve um objeto <code>Color</code> que representa a cor atual para o contexto gráfico.
<code>public void setColor(Color c) // classe Graphic</code>	Configura a cor atual para desenho com o contexto gráfico.

Fig. 11.4 Métodos `Color` e métodos `Graphics` relacionados com cores (parte 2 de 2).*Erro comum de programação 11.1*

Escrever o nome de qualquer constante `staticColor` com uma letra inicial maiúscula é um erro de sintaxe.

Dois construtores `Color` são mostrados na Figura 11.4 – um que recebe três argumentos `int` e um que recebe três argumentos `float`, com cada argumento especificando a quantidade de vermelho, verde e azul, respectivamente. Os valores `int` devem estar entre 0 e 255 e os valores `float` devem estar entre 0.0 e 1.0. O novo objeto `Color` terá as quantidades especificadas de vermelho, verde e azul. Os métodos `Color` `getRed`, `getGreen` e `getBlue` devolvem valores inteiros entre 0 a 255 que representam a quantidade de vermelho, verde e azul, respectivamente. O método `Graphics` `getColor` devolve um objeto `Color` que representa a cor de desenho atual. O método `Graphics` `setColor` configura a cor de desenho atual.

O aplicativo da Fig. 11.5 demonstra vários métodos da Fig. 11.4 que representam retângulos preenchidos e *strings* em várias cores diferentes.

```

1 // Fig. 11.5: ShowColors.java
2 // Demonstrando Colors.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class ShowColors extends JFrame {
12
13     // construtor configura a barra de título e as dimensões da janela
14     public ShowColors()
15     {
16         super( "Using colors" );
17
18         setSize( 400, 130 );
19         setVisible( true );
20     }
21 }
```

Fig. 11.5 Demonstrando como configurar e obter uma `Color` (parte 1 de 2).

```

22     // desenha retângulos e Strings em cores diferentes
23     public void paint( Graphics g )
24     {
25         // chama o método paint da superclasse
26         super.paint( g );
27
28         // configura nova cor de desenho usando inteiros
29         g.setColor( new Color( 255, 0, 0 ) );
30         g.fillRect( 25, 25, 100, 20 );
31         g.drawString( "Current RGB: " + g.getColor(), 130, 40 );
32
33         // configura nova cor de desenho usando valores em ponto flutuante
34         g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
35         g.fillRect( 25, 50, 100, 20 );
36         g.drawString( "Current RGB: " + g.getColor(), 130, 65 );
37
38         // configura nova cor de desenho usando objetos static Color
39         g.setColor( Color.blue );
40         g.fillRect( 25, 75, 100, 20 );
41         g.drawString( "Current RGB: " + g.getColor(), 130, 90 );
42
43         // exibe valores RGB individuais
44         Color color = Color.magenta;
45         g.setColor( color );
46         g.fillRect( 25, 100, 100, 20 );
47         g.drawString( "RGB values: " + color.getRed() + ", " +
48             color.getGreen() + ", " + color.getBlue(), 130, 115 );
49     }
50
51     // executa aplicativo
52     public static void main( String args[] )
53     {
54         ShowColors application = new ShowColors();
55
56         application.setDefaultCloseOperation(
57             JFrame.EXIT_ON_CLOSE );
58     }
59
60 } // fim da classe ShowColors

```

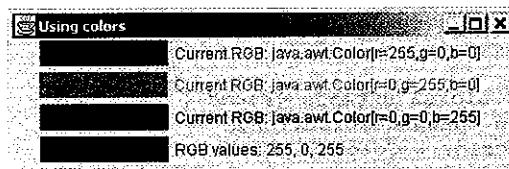


Fig. 11.5 Demonstrando como configurar e obter uma **Color** (parte 2 de 2).

Quando o aplicativo inicia a execução, o método `paint` da classe `ShowColors` (linhas 23 a 49) é chamado para pintar a janela. A linha 29 utiliza o método `setColor` de `Graphics` para configurar a cor atual de desenho. O método `setColor` recebe um objeto `Color`. A expressão `new Color(255, 0, 0)` cria um novo objeto `Color` que representa vermelho (valor vermelho 255 e 0 para os valores azul e verde). A linha 30 utiliza o método `fillRect` de `Graphics` para desenhar um retângulo preenchido com a cor atual. O método `fillRect` recebe os mesmos parâmetros que o método `drawRect` (discutido no Capítulo 3). A linha 31 utiliza o método `drawString` de `Graphics` para desenhar um `String` com a cor atual. A expressão `g.getColor()` recupera a cor atual do objeto `Graphics`. A `Color` retornada é concatenada com o string "Current RGB: " resultando em uma chamada implícita ao método `toString` da classe `Color`. Observe que a representação como `String` do objeto `Color` contém o nome da classe e do pacote (`java.awt.Color`) e os valores de vermelho, verde e azul.

As linhas 34 a 36 e as linhas 39 a 41 realizam as mesmas tarefas novamente. A linha 34 utiliza o construtor **Color** com três argumentos **float** para criar a cor verde (**0.0f** para vermelho, **1.0f** para verde e **0.0f** para azul). Observe a sintaxe das constantes. A letra **f** acrescentada a uma constante de ponto flutuante indica que a constante deve ser tratada como tipo **float**. Normalmente, as constantes de ponto flutuante são tratadas como tipo **double**.

A linha 39 configura a cor atual de desenho como uma das constantes **Color** (**Color.blue**) predefinidas. Observe que não é necessário o operador **new** para criar a constante. Como as constantes **Color** são **static**, elas são definidas quando a classe **Color** é carregada na memória durante a execução.

A instrução nas linhas 47 e 48 demonstra os métodos de **Color** **getRed**, **getGreen** e **getBlue** para o objeto predefinido **Color.magenta**.

Observe as linhas 56 e 57 de **main**. O método **setDefaultCloseOperation** de **JFrame** especifica a ação **default** executada quando o usuário clica no botão de fechamento da janela de um aplicativo. Neste caso, especificamos **JFrame.EXIT_ON_CLOSE** para indicar que o programa deve terminar quando o usuário clica no botão de fechar. Outras opções são **DO NOTHING_ON_CLOSE** (para ignorar o evento de fechar janela), **HIDE_ON_CLOSE** (para esconder a janela de modo que ela possa ser novamente exibida mais tarde) e **DISPOSE_ON_CLOSE** (para descartar a janela de forma que ela não possa ser novamente exibida mais tarde). Deste ponto em diante, implementaremos nosso próprio **WindowListener** somente se o programa deve realizar outras tarefas quando o usuário clica no botão de fechamento da janela. Caso contrário, usamos o método **setDefaultCloseOperation** para especificar que o programa deve terminar quando o usuário clica no botão de fechar.



Observação de engenharia de software 11.2

*Para alterar a cor, é necessário criar um novo objeto **Color** (ou utilizar uma das constantes **Color** predefinidas), uma vez que não há métodos **set** na classe **Color** para alterar as características da cor atual.*

Um dos mais novos recursos de Java é o componente GUI predefinido **JColorChooser** (pacote **javax.swing**) para selecionar cores. O aplicativo da Fig. 11.6 permite pressionar um botão para exibir um diálogo **JColorChooser**. Quando se seleciona uma cor e se pressiona no botão **OK** no diálogo, a cor de fundo ou segundo plano da janela do aplicativo muda de cor.

```

1 // Fig. 11.6: ShowColors2.java
2 // Demonstrando JColorChooser.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class ShowColors2 extends JFrame {
12     private JButton changeColorButton;
13     private Color color = Color.lightGray;
14     private Container container;
15
16     // configura a GUI
17     public ShowColors2()
18     {
19         super( "Using JColorChooser" );
20
21         container = getContentPane();

```

Fig. 11.6 Demonstrando o diálogo **JColorChooser** (parte 1 de 3).

```

22     container.setLayout( new FlowLayout() );
23
24     // configura changeColorButton e registra o tratador de eventos
25     changeColorButton = new JButton( "Change Color" );
26
27     changeColorButton.addActionListener(
28
29         // classe interna anônima
30         new ActionListener() {
31
32             // exibe o JColorChooser quando o usuário clica no botão
33             public void actionPerformed( ActionEvent event )
34             {
35                 color = JColorChooser.showDialog(
36                     ShowColors2.this, "Choose a color", color );
37
38                 // se nenhuma cor é devolvida, configura com a cor default
39                 if ( color == null )
40                     color = Color.lightGray;
41
42                 // muda a cor de fundo do painel de conteúdo
43                 container.setBackground( color );
44             }
45
46         } // fim da classe interna anônima
47
48     ); // fim da chamada para addActionListener
49
50     container.add( changeColorButton );
51
52     setSize( 400, 130 );
53     setVisible( true );
54 }
55
56 // executa o aplicativo
57 public static void main( String args[] )
58 {
59     ShowColors2 application = new ShowColors2();
60
61     application.setDefaultCloseOperation(
62         JFrame.EXIT_ON_CLOSE );
63 }
64
65 } // fim da classe ShowColors2

```

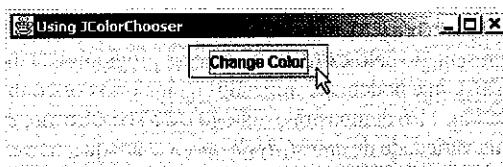


Fig. 11.6 Demonstrando o diálogo JColorChooser (parte 2 de 3).

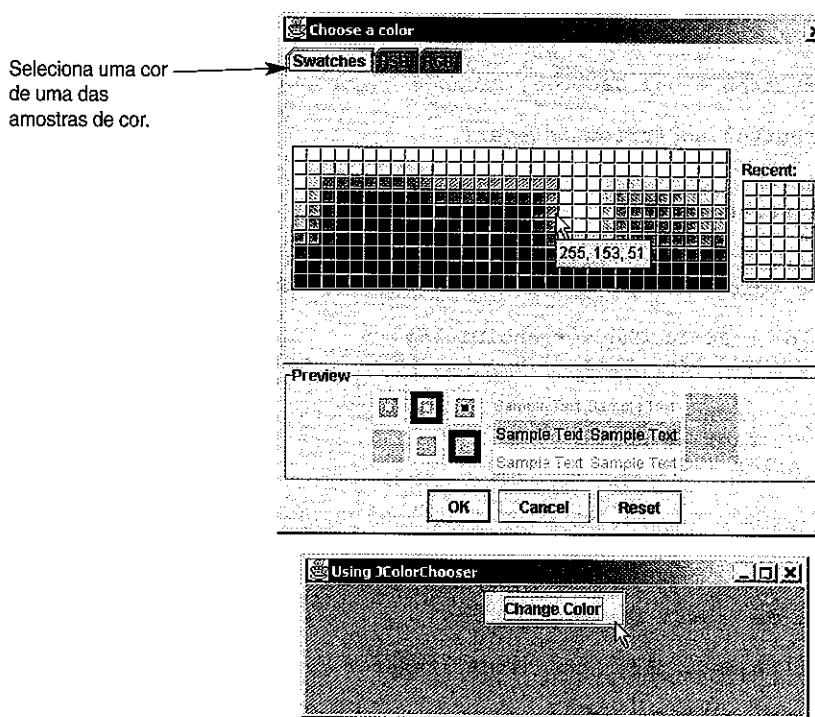


Fig. 11.6 Demonstrando o diálogo `JColorChooser` (parte 3 de 3).

As linhas 35 e 36 (do método `actionPerformed` para `changeColor`) utilizam o método `static showDialog` da classe `JColorChooser` para exibir o diálogo de seleção de cor. Esse método devolve o objeto `Color` selecionado (ou `null`, se o usuário pressionar `Cancel` ou fechar o diálogo sem pressionar `OK`).

O método `showDialog` recebe três argumentos – uma referência ao seu `Component` pai, um `String` para exibir na barra de título do diálogo e a `Color` inicial selecionada para o diálogo. O componente pai é a janela a partir da qual o diálogo é exibido. Enquanto o diálogo de seleção de cor estiver na tela, o usuário não pode interagir com o componente pai. Esse tipo de diálogo se chama *diálogo modal* e é discutido no Capítulo 13. Repare na sintaxe especial `ShowColors2.this` utilizada na instrução da linha 36. Ao utilizar uma classe interna, pode-se acessar a referência `this` do objeto da classe externa qualificando `this` com o nome da classe externa e o operador ponto `(..)`.

Depois que o usuário seleciona uma cor, as linhas 39 e 40 determinam se `color` é `null` e, se for, configuram `color` com o `default Color.lightGray`. A linha 43 utiliza o método `setBackground` para alterar a cor de fundo do painel de conteúdo (representado pelo `container` nesse programa). O método `setBackground` é um dos muitos métodos de `Component` que podem ser utilizados na maioria dos componentes GUI.

A segunda captura de tela da Fig. 11.6 demonstra o diálogo `JColorChooser default` que permite ao usuário selecionar uma cor a partir de uma variedade de *amostras de cor*. Repare que, na verdade, há três guias na parte superior do diálogo – **Swatches**, **HSB** e **RGB**. Elas representam três maneiras diferentes de selecionar uma cor. A guia **HSB** permite selecionar uma cor com base em um *tom*, uma *saturação* e um *brilho*. A guia **RGB** permite selecionar uma cor utilizando controles deslizantes para selecionar os componentes de vermelho, verde e azul da cor. As guias **HSB** e **RGB** são mostradas na Fig. 11.7.

11.4 Controle de fontes

Esta seção apresenta métodos e constantes para o controle de fontes. A maioria dos métodos de fontes e das constantes de fontes faz parte da classe da `Font`. Alguns métodos da classe `Font` e da classe `Graphics` estão resumidos na Fig. 11.8.

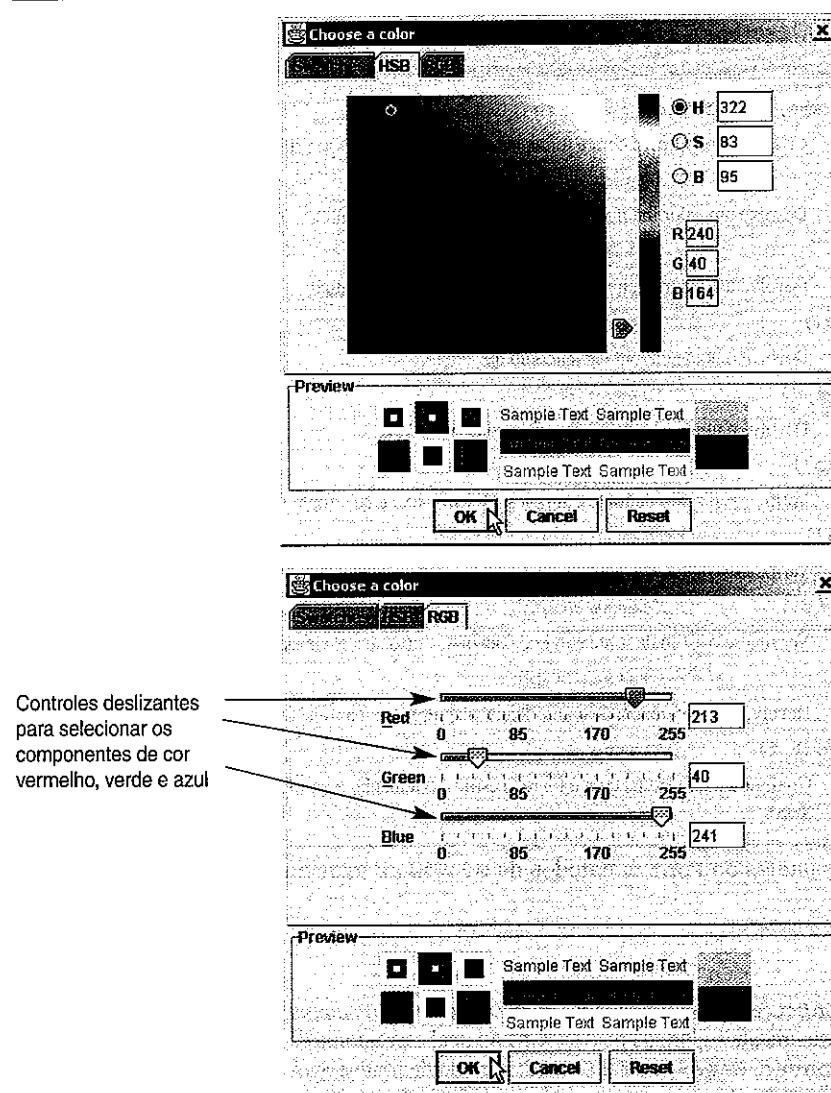


Fig. 11.7 As guias HSB e RGB do diálogo JColorChooser.

Método ou constante	Descrição
<code>public final static int PLAIN // classe Font</code>	Constante que representa um estilo de fonte simples.
<code>public final static int BOLD // classe Font</code>	Constante que representa um estilo de fonte em negrito.
<code>public final static int ITALIC // classe Font</code>	Constante que representa um estilo de fonte em itálico.

Fig. 11.8 Métodos e constantes de `Font` e métodos de `Graphics` relacionados com fontes (parte 1 de 2).

Método ou constante Descrição

```

public Font( String name, int style, int size )
            Cria um objeto Font com a fonte, o estilo e o tamanho especificados.

public int getStyle()          // Classe Font
            Devolve um valor inteiro que indica o estilo de fonte atual.

public int getSize()           // Classe Font
            Devolve um valor inteiro que indica o tamanho da fonte atual.

public String getName()        // Classe Font
            Devolve o nome da fonte atual como um string.

public String getFamily()      // Classe Font
            Devolve o nome da família de fontes como um string.

public boolean isPlain()       // Classe Font
            Verifica se o estilo da fonte é simples. Devolve true se a fonte for simples.

public boolean isBold()        // Classe Font
            Verifica se o estilo da fonte é negrito. Devolve true se a fonte estiver em negrito.

public boolean isItalic()     // Classe Font
            Verifica se o estilo da fonte é itálico. Devolve true se a fonte estiver em itálico.

public Font getFont()         // Classe Graphic
            Devolve uma referência de objeto Font que representa a fonte atual.

public void setFont(Font f)   // Classe Graphic
            Configura a fonte atual com a fonte, o estilo e o tamanho especificados pela referência ao objeto
            Font f.

```

Fig. 11.8 Métodos e constantes de **Font** e métodos de **Graphics** relacionados com fontes (parte 2 de 2).

O construtor da classe **Font** recebe três argumentos – *nome da fonte*, *estilo da fonte* e *tamanho da fonte*. O nome da fonte é qualquer fonte atualmente suportada pelo sistema no qual o programa está sendo executado, como as fontes *default* de Java **Monospaced**, **SansSerif** e **Serif**. O estilo de fonte é **Font.PLAIN**, **Font.ITALIC** ou **Font.BOLD** (constantes *static* da classe **FONT**). Os estilos de fontes podem ser utilizados em combinação (por exemplo, **FONT.ITALIC + FONT.BOLD**). O tamanho da fonte é medido em pontos. O *ponto* é 1/72 de uma polegada. O método **Graphics** **setFont** configura a fonte atual de desenho – a fonte em que o texto será exibido – para seu argumento **Font**.



Dica de portabilidade 11.2

O número de fontes varia significativamente de um sistema para outro. O J2SDK garante que as fontes **Serif**, **Monospaced**, **SansSerif**, **Dialog** e **DialogInput** estarão disponíveis.



Erro comum de programação 11.2

Especificar uma fonte que não está disponível em um sistema é um erro de lógica. Java a substituirá pela fonte default do sistema.

O programa da Fig. 11.9 exibe texto em quatro fontes diferentes com cada fonte em um tamanho diferente. O programa utiliza o construtor de **Font** para inicializar os objetos **Font** nas linhas 30, 35, 40 E 47 (cada um em uma chamada ao método **setFont** de **Graphics**, para alterar a fonte de desenho). Cada chamada ao construtor **Font** passa um nome de fonte (**Serif**, **Monospaced** ou **SansSerif**) como um **String**, um estilo de fonte

(Font.PLAIN, Font.ITALIC ou Font.BOLD) e um tamanho de fonte. Uma vez que o método **setFont** de **Graphics** é invocado, todo texto exibido após a chamada aparecerá na nova fonte até que ela seja alterada. Observe que essa linha 45 altera a cor de desenho para vermelho, de modo que o próximo *string* exibido aparece em vermelho.

```

1 // Fig. 11.9: Fonts.java
2 // Usando fontes
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class Fonts extends JFrame {
12
13     // configura a barra de título e as dimensões da janela
14     public Fonts()
15     {
16         super( "Using fonts" );
17
18         setSize( 420, 125 );
19         setVisible( true );
20     }
21
22     // exibe Strings em fontes e cores diferentes
23     public void paint( Graphics g )
24     {
25         // chama o método paint da superclasse
26         super.paint( g );
27
28         // configura a fonte atual para Serif (Times),
29         // negrito, 12 pontos e desenha um string
30         g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
31         g.drawString( "Serif 12 point bold.", 20, 50 );
32
33         // configura a fonte atual para Monospaced (Courier),
34         // itálico, 24 pontos e desenha um string
35         g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
36         g.drawString( "Monospaced 24 point italic.", 20, 70 );
37
38         // configura a fonte atual para SansSerif (Helvetica),
39         // normal, 14 pontos e desenha um string
40         g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
41         g.drawString( "SansSerif 14 point plain.", 20, 90 );
42
43         // configura a fonte atual para Serif (times),
44         // negrito/itálico, 18 pontos e desenha um string
45         g.setColor( Color.red );
46         g.setFont(
47             new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
48         g.drawString( g.getFont().getName() + " " +
49             g.getFont().getSize() +
50             " point bold italic.", 20, 110 );
51     }
52
53     // executa o aplicativo
54     public static void main( String args[] )
55     {

```

Fig. 11.9 Usando o método **setFont** de **Graphics** para mudar **Fonts** (parte 1 de 2).

```

56     Fonts application = new Fonts();
57
58     application.setDefaultCloseOperation(
59         JFrame.EXIT_ON_CLOSE );
60 }
61
62 } // fim da classe Fonts

```

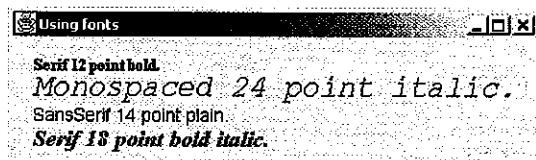


Fig. 11.9 Usando o método `setFont` de `Graphics` para mudar `Fonts` (parte 2 de 2).



Observação de engenharia de software 11.3

Para alterar a fonte, você deve criar um novo objeto `Font`; não existem métodos `set` na classe `Font` para alterar as características da fonte atual.

Freqüentemente é necessário obter informações sobre a fonte atual como o nome, o estilo e o tamanho da fonte. Vários métodos `Font` utilizados para obter informações sobre fontes são resumidos na Fig. 11.8. O método `getStyle` devolve um valor inteiro que representa o estilo atual. O valor inteiro devolvido é `Font.PLAIN`, `Font.ITALIC`, `Font.BOLD` ou qualquer combinação de `Font.PLAIN`, `Font.ITALIC` e `Font.BOLD`.

O método `getSize` devolve o tamanho da fonte em pontos. O método `getName` devolve o nome atual da fonte como um `String`. O método `getFamily` devolve o nome da família de fontes à qual a fonte atual pertence. O nome da família de fontes é específico da plataforma.



Dica de portabilidade 11.3

Java utiliza nomes padronizados de fontes e os mapeia para nomes de fontes específicas do sistema para fins de portabilidade. Isso é transparente para o programador.

Os métodos de `Font` também estão disponíveis para testar o estilo da fonte atual e são resumidos na Fig. 11.8. O método `isPlain` devolve `true` se o estilo atual de fonte for simples. O método `isBold` devolve `true` se o estilo atual de fonte for negrito. O método `isItalic` devolve `true` se o estilo atual da fonte for itálico.

Às vezes, é preciso saber informações precisas sobre as medidas da fonte – como a *altura*, a *descida* (quanto um caractere desce abaixo da linha de base), a *subida* (quanto um caractere se eleva acima da linha de base) e a *entrelinha* (a diferença entre a descida de uma linha de texto e a subida da linha de texto abaixo dela – isto é, o espaçamento entre linhas). A Fig. 11.10 ilustra algumas *medidas de fontes* comuns. Observe que a coordenada passada para `drawString` corresponde ao canto inferior esquerdo da linha de base da fonte.

A classe `FontMetrics` define vários métodos para obter medidas de fonte. Esses métodos e o método `getFontMetrics` de `Graphics` são resumidos na Fig. 11.11.

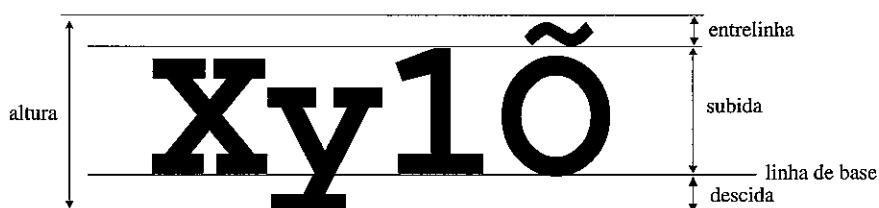


Fig. 11.10 Medidas de fontes.

Método	Descrição
<code>public int getAscent()</code>	// classe FontMetrics Devolve um valor que representa a subida de uma fonte, em pontos.
<code>public int getDescent()</code>	// classe FontMetrics Devolve um valor que representa a descida de uma fonte, em pontos.
<code>public int getLeading()</code>	// classe FontMetrics Devolve um valor que representa a entrelinha de uma fonte, em pontos.
<code>public int getHeight()</code>	// classe FontMetrics Devolve um valor que representa a altura de uma fonte, em pontos.
<code>public FontMetrics getFontMetrics()</code>	// classe Graphics Devolve o objeto <code>FontMetrics</code> para a <code>Font</code> de desenho atual.
<code>public FontMetrics getFontMetrics(Font f)</code>	// classe Graphics Devolve o objeto <code>FontMetrics</code> para o argumento <code>Font</code> especificado.

Fig. 11.11 Os métodos `FontMetrics` e `Graphics` para obter medidas de fontes.

O programa da Fig. 11.12 utiliza os métodos da Fig. 11.11 para obter informações de medida de fonte para duas fontes.

```

1 // Fig. 11.12: Metrics.java
2 // Demonstrando os métodos da classe FontMetrics e da
3 // classe Graphics úteis para obter métricas da fonte.
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class Metrics extends JFrame {
13
14     // configura o String da barra de título e as dimensões da janela
15     public Metrics()
16     {
17         super( "Demonstrating FontMetrics" );
18
19         setSize( 510, 210 );
20         setVisible( true );
21     }
22
23     // exibe métricas da fonte
24     public void paint( Graphics g )
25     {
26         // chama o método paint da superclasse
27         super.paint( g );
28
29         g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
30         FontMetrics metrics = g.getFontMetrics();
31         g.drawString( "Current font: " + g.getFont(), 10, 40 );
32         g.drawString( "Ascent: " + metrics.getAscent(), 10, 55 );

```

Fig. 11.12 Obtendo informações sobre métricas da fonte (parte 1 de 2).

```

33     g.drawString( "Descent: " + metrics.getDescent(), 10, 70 );
34     g.drawString( "Height: " + metrics.getHeight(), 10, 85 );
35     g.drawString( "Leading: " + metrics.getLeading(), 10, 100 );
36
37     Font font = new Font( "Serif", Font.ITALIC, 14 );
38     metrics = g.getFontMetrics( font );
39     g.setFont( font );
40     g.drawString( "Current font: " + font, 10, 130 );
41     g.drawString( "Ascent: " + metrics.getAscent(), 10, 145 );
42     g.drawString( "Descent: " + metrics.getDescent(), 10, 160 );
43     g.drawString( "Height: " + metrics.getHeight(), 10, 175 );
44     g.drawString( "Leading: " + metrics.getLeading(), 10, 190 );
45 }
46
47 // executa o aplicativo
48 public static void main( String args[] )
49 {
50     Metrics application = new Metrics();
51
52     application.setDefaultCloseOperation(
53         JFrame.EXIT_ON_CLOSE );
54 }
55
56 } // fim da classe Metrics

```

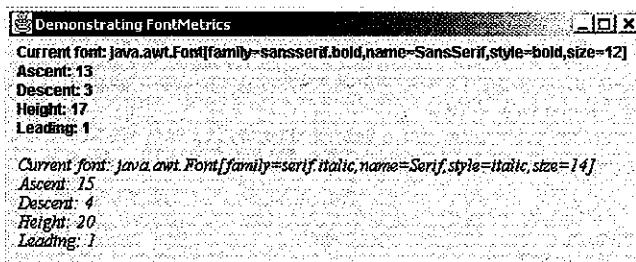


Fig. 11.12 Obtendo informações sobre métricas da fonte (parte 2 de 2).

A linha 29 cria e configura a fonte atual de desenho como uma fonte **SansSerif**, de 12 pontos, negrito. A linha 30 utiliza o método **getFontMetrics** de **Graphics** para obter o objeto **FontMetrics** para a fonte atual. A linha 31 utiliza uma chamada implícita para o método **toString** da classe **Font** para enviar para a saída a representação de *string* da fonte. As linhas 32 a 35 utilizam os métodos de **FontMetrics** para obter a subida, a descida, a altura e a entrelinha da fonte.

A linha 37 cria uma nova fonte **Serif**, de 14 pontos, em itálico. A linha 38 utiliza uma segunda versão do método **getFontMetrics** de **Graphics**, que recebe um argumento **Font** e devolve um objeto **FontMetrics** correspondente. As linhas 41 a 44 obtêm a descida, a subida, a altura e a entrelinha da fonte. Observe que as medidas de fonte são ligeiramente diferentes para as duas fontes.

11.5 Desenhando linhas, retângulos e elipses

Esta seção apresenta diversos métodos **Graphics** para desenhar linhas, retângulos e elipses. Os métodos e seus parâmetros são resumidos na Fig. 11.13. Para cada método de desenho que exige um parâmetro **width** e **height**, **width** e **height** devem ser valores não-negativos. Caso contrário, a forma não será exibida.

Método	Descrição
<code>public void drawLine(int x1, int y1, int x2, int y2)</code>	Desenha uma linha entre o ponto (x1, y1) e o ponto (x2, y2).
<code>public void drawRect(int x, int y, int width, int height)</code>	Desenha um retângulo com a <code>width</code> e a <code>height</code> especificadas. O canto superior esquerdo do retângulo tem as coordenadas (x, y).
<code>public void fillRect(int x, int y, int width, int height)</code>	Desenha um retângulo sólido com a <code>width</code> e a <code>height</code> especificadas. O canto superior esquerdo do retângulo tem as coordenadas (x, y).
<code>public void clearRect(int x, int y, int width, int height)</code>	Desenha um retângulo sólido com a <code>width</code> e a <code>height</code> especificadas, na cor de fundo atual. O canto superior esquerdo do retângulo tem as coordenadas (x, y).
<code>public void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)</code>	Desenha um retângulo com cantos arredondados na cor atual com a <code>width</code> e a <code>height</code> especificadas. A <code>arcWidth</code> e a <code>arcHeight</code> determinam o arredondamento dos cantos (veja Fig. 11.15).
<code>public void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)</code>	Desenha um retângulo sólido com cantos arredondados na cor atual com a <code>width</code> e a <code>height</code> especificadas. A <code>arcWidth</code> e a <code>arcHeight</code> determinam o arredondamento dos cantos (veja Fig. 11.15).
<code>public void draw3DRect(int x, int y, int width, int height, boolean b)</code>	Desenha um retângulo tridimensional na cor atual com a <code>width</code> e a <code>height</code> especificadas. O canto superior esquerdo do retângulo tem as coordenadas (x, y). O retângulo aparece em alto relevo quando <code>b</code> é <code>true</code> e em baixo relevo quando <code>b</code> é <code>false</code> .
<code>public void fill3DRect(int x, int y, int width, int height, boolean b)</code>	Desenha um retângulo tridimensional preenchido na cor atual com a <code>width</code> e a <code>height</code> especificadas. O canto superior esquerdo do retângulo tem as coordenadas (x, y). O retângulo aparece em alto relevo quando <code>b</code> é <code>true</code> e em baixo relevo quando <code>b</code> é <code>false</code> .
<code>public void drawOval(int x, int y, int width, int height)</code>	Desenha uma elipse na cor atual com a <code>width</code> e a <code>height</code> especificadas. O canto superior esquerdo do retângulo delimitador está nas coordenadas (x, y). A elipse toca todos os quatro lados do retângulo delimitador, no centro de cada lado (veja a Fig. 11.16).
<code>public void fillOval(int x, int y, int width, int height)</code>	Desenha uma elipse preenchida na cor atual com a <code>width</code> e a <code>height</code> especificadas. O canto superior esquerdo do retângulo delimitador está nas coordenadas (x, y). A elipse toca todos os quatro lados do retângulo delimitador no centro de cada lado (veja a Fig. 11.16).

Fig. 11.13 Métodos de `Graphics` que desenham linhas, retângulos e elipses.

O aplicativo da Fig. 11.14 demonstra o desenho de diversas linhas, retângulos, retângulos em 3D, retângulos arredondados e elipses.

```
1 // Fig. 11.14: LinesRectsOvals.java
2 // Desenhando linhas, retângulos e elipses
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class LinesRectsOvals extends JFrame {
12
13     // configura o String da barra de título e as dimensões da janela
14     public LinesRectsOvals()
15     {
16         super( "Drawing lines, rectangles and ovals" );
17
18         setSize( 400, 165 );
19         setVisible( true );
20     }
21
22     // exibe várias linhas, retângulos e elipses
23     public void paint( Graphics g )
24     {
25         // chama o método paint da superclasse
26         super.paint( g );
27
28         g.setColor( Color.red );
29         g.drawLine( 5, 30, 350, 30 );
30
31         g.setColor( Color.blue );
32         g.drawRect( 5, 40, 90, 55 );
33         g.fillRect( 100, 40, 90, 55 );
34
35         g.setColor( Color.cyan );
36         g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
37         g.drawRoundRect( 290, 40, 90, 55, 20, 20 );
38
39         g.setColor( Color.yellow );
40         g.draw3DRect( 5, 100, 90, 55, true );
41         g.fill3DRect( 100, 100, 90, 55, false );
42
43         g.setColor( Color.magenta );
44         g.drawOval( 195, 100, 90, 55 );
45         g.fillOval( 290, 100, 90, 55 );
46     }
47
48     // executa o aplicativo
49     public static void main( String args[] )
50     {
51         LinesRectsOvals application = new LinesRectsOvals();
52
53         application.setDefaultCloseOperation(
54             JFrame.EXIT_ON_CLOSE );
55     }
56
57 } // fim da classe LinesRectsOvals
```

Fig. 11.14 Demonstrando o método `drawLine` da classe `Graphics` (parte 1 de 2).

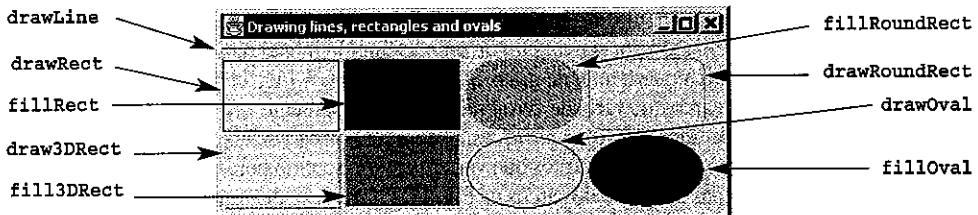


Fig. 11.14 Demonstrando o método `drawLine` da classe `Graphics` (parte 2 de 2).

Os métodos `fillRoundRect` (linha 36) e `drawRoundRect` (linha 37) desenham retângulos com cantos arredondados. Seus dois primeiros argumentos especificam as coordenadas do canto superior esquerdo do *retângulo delimitador* – a área em que o retângulo arredondado será desenhado. Observe que as coordenadas do canto superior esquerdo não são a borda do retângulo arredondado, mas as coordenadas em que a borda estaria se o retângulo tivesse cantos quadrados. O terceiro e quarto argumentos especificam a `width` e a `height` do retângulo. Seus últimos dois argumentos – `arcWidth` e `arcHeight` – determinam os diâmetros horizontal e vertical dos arcos utilizados para representar os cantos.

Os métodos `draw3DRect` (linha 40) e `fill3DRect` (linha 41) recebem os mesmos argumentos. Os dois primeiros argumentos especificam o canto superior esquerdo do retângulo. Os próximos dois argumentos especificam a `width` e a `height` do retângulo, respectivamente. O último argumento determina se o retângulo está em *baixo-relevo* (`true`) ou *alto-relevo* (`false`). O efeito tridimensional de `draw3DRect` aparece como duas bordas do retângulo na cor original e duas bordas em uma cor ligeiramente mais escura. O efeito tridimensional de `fill3DRect` aparece como duas bordas do retângulo na cor de desenho original e o preenchimento e as outras duas bordas em uma cor ligeiramente mais escura. Retângulos em alto-relevo têm a cor de desenho original nas bordas superior e esquerda. Retângulos em baixo-relevo têm a cor de desenho original nas bordas inferior e direita. O efeito tridimensional é difícil de ver em algumas cores.

A Fig. 11.15 indica a largura do arco, a altura do arco, a altura e a largura de um retângulo arredondado. Utilizar o mesmo valor para `arcWidth` e `arcHeight` produz um quarto de círculo em cada canto. Quando `width`, `height`, `arcWidth` e `arcHeight` têm os mesmos valores, o resultado é um círculo. Se os valores para `width` e `height` forem os mesmos e os valores de `arcWidth` e `arcHeight` forem 0, o resultado é um quadrado.

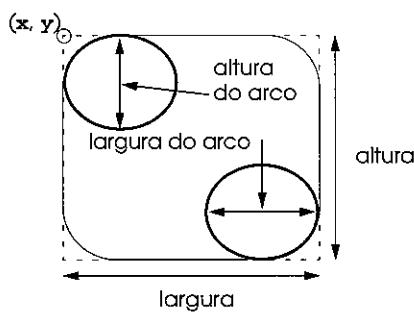


Fig. 11.15 A largura e a altura do arco para retângulos arredondados.

Os métodos `drawOval` e `fillOval` aceitam os mesmos quatro argumentos. Os dois primeiros argumentos especificam as coordenadas do canto superior esquerdo do retângulo delimitador que contém a elipse. Os últimos dois argumentos especificam a `width` e a `height` do retângulo delimitador, respectivamente. A Fig. 11.16 mostra uma elipse delimitada por um retângulo. Observe que a elipse toca o centro de todos os quatro lados do retângulo delimitador (o retângulo delimitador não é exibido na tela).

11.6 Desenhandos arcos

O arco faz parte de uma elipse. Os ângulos de arcos são medidos em graus. Os arcos *varrem* a partir de um *ângulo inicial* ou número de graus especificado por seu *ângulo de arco*. O ângulo inicial indica em graus onde o arco começa. O ângulo do arco especifica o número total de graus que o arco varre. A Fig. 11.17 ilustra dois arcos. O conjunto esquerdo de eixos mostra um arco que varre de zero grau a aproximadamente 110 graus. Os arcos que varrem em uma direção anti-horária são medidos em *graus positivos*. O conjunto direito de eixos mostra um arco que varre de zero grau a aproximadamente -110 graus. Os arcos que varrem em um sentido horário são medidos em *graus negativos*. Repare nas caixas tracejadas em torno dos arcos na Fig. 11.17. Ao desenhar um arco, especificamos um retângulo delimitador para uma elipse. O arco varrerá ao longo de parte da elipse. Os métodos *drawArc* e *fillArc* de *Graphics* para desenhar arcos são resumidos na Fig. 11.18.

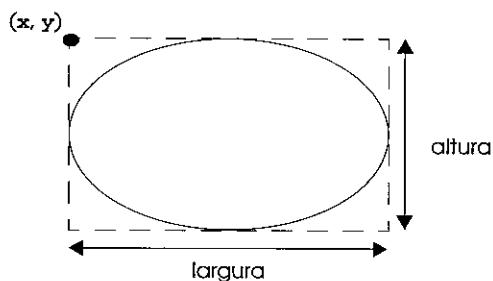


Fig. 11.16 Elipse delimitada por um retângulo.

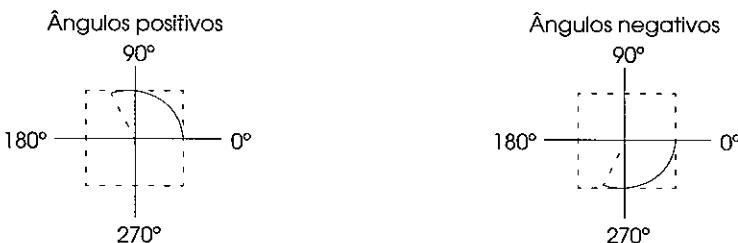


Fig. 11.17 Ângulos de arcos positivo e negativo.

Método	Descrição
<code>public void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)</code>	Desenha um arco em relação às coordenadas do canto superior esquerdo do retângulo delimitador <code>(x, y)</code> com <code>a width</code> e <code>a height</code> especificadas. O segmento de arco é desenhado começando-se em <code>startAngle</code> e varrendo-se <code>arcAngle</code> graus.
<code>public void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)</code>	Desenha um arco sólido (isto é, um setor) em relação às coordenadas do canto superior esquerdo do retângulo delimitador <code>(x, y)</code> com <code>a width</code> e <code>a height</code> especificadas. O segmento de arco é desenhado começando-se em <code>startAngle</code> e varrendo-se <code>arcAngle</code> graus.

Fig. 11.18 Métodos de *Graphics* para desenhar arcos.

O programa da Fig. 11.19 demonstra os métodos para desenhar arcos da Fig. 11.18. O programa desenha seis arcos (três não-preenchidos e três preenchidos). Para ilustrar o retângulo delimitador que ajuda a determinar onde o arco aparece, os primeiros três arcos são exibidos dentro de um retângulo amarelo que tem os mesmos argumentos `x, y, width e height` que os arcos.

```

1 // Fig. 11.19: DrawArcs.java
2 // Desenhando arcos
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class DrawArcs extends JFrame {
12
13     // configura o String da barra de título e as dimensões da janela
14     public DrawArcs()
15     {
16         super( "Drawing Arcs" );
17
18         setSize( 300, 170 );
19         setVisible( true );
20     }
21
22     // desenha retângulos e arcos
23     public void paint( Graphics g )
24     {
25         // chama o método paint da superclasse
26         super.paint( g );
27
28         // começa em 0 e varre 360 graus
29         g.setColor( Color.yellow );
30         g.drawRect( 15, 35, 80, 80 );
31         g.setColor( Color.black );
32         g.drawArc( 15, 35, 80, 80, 0, 360 );
33
34         // começa em 0 e varre 110 graus
35         g.setColor( Color.yellow );
36         g.drawRect( 100, 35, 80, 80 );
37         g.setColor( Color.black );
38         g.drawArc( 100, 35, 80, 80, 0, 110 );
39
40         // começa em 0 e varre -270 graus
41         g.setColor( Color.yellow );
42         g.drawRect( 185, 35, 80, 80 );
43         g.setColor( Color.black );
44         g.drawArc( 185, 35, 80, 80, 0, -270 );
45
46         // começa em 0 e varre 360 graus
47         g.fillArc( 15, 120, 80, 40, 0, 360 );
48
49         // começa em 270 e varre -90 graus
50         g.fillArc( 100, 120, 80, 40, 270, -90 );
51
52         // começa em 0 e varre -270 graus
53         g.fillArc( 185, 120, 80, 40, 0, -270 );

```

Fig. 11.19 Demonstrando `drawArc` e `fillArc` (parte 1 de 2).

```

54 }
55
56 // executa o aplicativo
57 public static void main( String args[] )
58 {
59     DrawArcs application = new DrawArcs();
60
61     application.setDefaultCloseOperation(
62         JFrame.EXIT_ON_CLOSE );
63 }
64
65 } // fim da classe DrawArcs

```

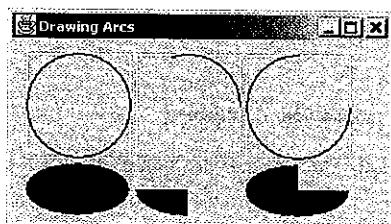


Fig. 11.19 Demonstrando `drawArc` e `fillArc` (parte 2 de 2).

11.7 Desenhando polígonos e polilinhas

Os *polígonos* são formas de múltiplos lados. As *polilinhas* são uma série de pontos conectados. Os métodos de `Graphics` para desenhar polígonos e polilinhas são discutidos na Fig. 11.20. Observe que alguns métodos requerem um objeto `Polygon` (pacote `java.awt`). Os construtores da classe `Polygon` também são descritos na Fig. 11.20.

Método	Descrição
<code>public void drawPolygon(int xPoints[], int yPoints[], int points)</code>	Desenha um polígono. A coordenada x de cada ponto é especificada no array <code>xPoints</code> e a coordenada y de cada ponto é especificada no array <code>yPoints</code> . O último argumento especifica o número de <code>points</code> . Esse método desenha um polígono fechado – mesmo se o último ponto for diferente do primeiro ponto.
<code>public void drawPolyline(int xPoints[], int yPoints[], int points)</code>	Desenha uma série de linhas conectadas. A coordenada x de cada ponto é especificada no array <code>xPoints</code> e a coordenada y de cada ponto é especificada no array <code>yPoints</code> . O último argumento especifica o número de <code>points</code> . Se o último ponto for diferente do primeiro ponto, a polilinha não é fechada.
<code>public void drawPolygon(Polygon p)</code>	Desenha o polígono fechado especificado.
<code>public void fillPolygon(int xPoints[], int yPoints[], int points)</code>	Desenha um polígono sólido. A coordenada x de cada ponto é especificada no array <code>xPoints</code> e a coordenada y de cada ponto é especificada no array <code>yPoints</code> . O último argumento especifica o número de <code>points</code> . Esse método desenha um polígono fechado – mesmo se o último ponto for diferente do primeiro ponto.

Fig. 11.20 Métodos de `Graphics` para desenhar polígonos e a classe de construtores `Polygon` (parte 1 de 2).

Método	Descrição
<code>public void fillPolygon(Polygon p)</code>	Desenha o polígono sólido especificado. O polígono é fechado.
<code>public Polygon()</code>	Constrói um novo objeto polígono. O polígono não contém nenhum ponto.
<code>public Polygon(int xValues[], int yValues[], int numberofPoints)</code>	// classe Polygon Constrói um novo objeto polígono. O polígono tem <code>numberofPoints</code> lados, com cada ponto consistindo em uma coordenada <code>x</code> de <code>xValues</code> e uma coordenada <code>y</code> de <code>yValues</code> .

Fig. 11.20 Métodos de `Graphics` para desenhar polígonos e a classe de construtores `Polygon` (parte 2 de 2).

O programa da Fig. 11.21 desenha polígonos e polilinhas utilizando os métodos e construtores da Fig. 11.20.

```

1 // Fig. 11.21: DrawPolygons.java
2 // Desenhando polígonos
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class DrawPolygons extends JFrame {
12
13     // configura o String da barra de título e as dimensões da janela
14     public DrawPolygons()
15     {
16         super( "Drawing Polygons" );
17
18         setSize( 275, 230 );
19         setVisible( true );
20     }
21
22     // desenha polígonos e polilinhas
23     public void paint( Graphics g )
24     {
25         // chama o método paint da superclasse
26         super.paint( g );
27
28         int xValues[] = { 20, 40, 50, 30, 20, 15 };
29         int yValues[] = { 50, 50, 60, 80, 80, 60 };
30         Polygon polygon1 = new Polygon( xValues, yValues, 6 );
31
32         g.drawPolygon( polygon1 );
33
34         int xValues2[] = { 70, 90, 100, 80, 70, 65, 60 };
35         int yValues2[] = { 100, 100, 110, 110, 130, 110, 90 };
36
37         g.drawPolyline( xValues2, yValues2, 7 );
38

```

Fig. 11.21 Demonstrando `drawPolygon` e `fillPolygon` (parte 1 de 2).

```

39     int xValues3[] = { 120, 140, 150, 190 };
40     int yValues3[] = { 40, 70, 80, 60 };
41
42     g.fillPolygon( xValues3, yValues3, 4 );
43
44     Polygon polygon2 = new Polygon();
45     polygon2.addPoint( 165, 135 );
46     polygon2.addPoint( 175, 150 );
47     polygon2.addPoint( 270, 200 );
48     polygon2.addPoint( 200, 220 );
49     polygon2.addPoint( 130, 180 );
50
51     g.fillPolygon( polygon2 );
52 }
53
54 // executa o aplicativo
55 public static void main( String args[] )
56 {
57     DrawPolygons application = new DrawPolygons();
58
59     application.setDefaultCloseOperation(
60         JFrame.EXIT_ON_CLOSE );
61 }
62
63 } // fim da classe DrawPloygons

```

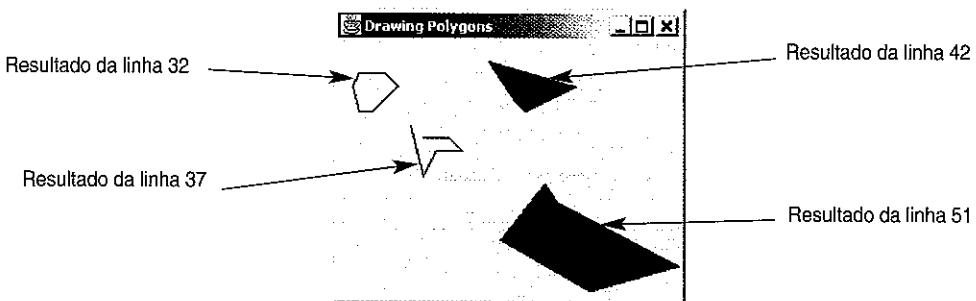


Fig. 11.21 Demonstrando `drawPolygon` e `fillPolygon` (parte 2 de 2).

As linhas 28 e 29 criam dois `arrays int` e os utilizam para especificar os pontos para o `Polygon polygon1`. A chamada do construtor `Polygon` na linha 30 recebe o `array xValues`, que contém a coordenada `x` de cada ponto, o `array yValues`, que contém a coordenada `y` de cada ponto e 6 (o número de pontos no polígono). A linha 32 exibe `polygon1` passando-o como argumento para o método `drawPolygon` de `Graphics`.

As linhas 34 e 35 criam dois `arrays int` e os utilizam para especificar os pontos para uma série de linhas conectadas. O `array xValues2` contém a coordenada `x` de cada ponto e o `array yValues2` contém a coordenada `y` de cada ponto. A linha 37 utiliza o método `drawPolyline` de `Graphics` para exibir uma série de linhas conectadas especificadas com os argumentos `xValues2, yValues2` e 7 (o número de pontos).

As linhas 39 e 40 criam dois `arrays int` e os utilizam para especificar os pontos de um polígono. O `array xValues3` contém a coordenada `x` de cada ponto e o `array yValues3` contém a coordenada `y` de cada ponto. A linha 42 exibe um polígono passando para o método `fillPolygon` de `Graphics` os dois `arrays` (`xValues3` e `yValues3`) e o número de pontos a desenhar (4).

Erro comum de programação 11.3



Dispara-se uma `ArrayIndexOutOfBoundsException` se o número de pontos especificado no terceiro argumento para o método `drawPolygon` ou o método `fillPolygon` for maior que o número de elementos nos arrays de coordenadas que definem o polígono a ser exibido.

A linha 44 cria o **Polygon polygon2** sem pontos. As linhas 45 a 49 utilizam o método **addPoint** de **Polygon** para adicionar pares de coordenadas *x* e *y* ao **Polygon**. A linha 51 exibe o **Polygon polygon2** passando-o para o método **fillPolygon** de **Graphics**.

11.8 A API Java2D

A nova *API Java2D* fornece recursos gráficos bidimensionais avançados que exigem manipulações gráficas complexas e detalhadas. A API inclui recursos para processamento de *line art*, texto e imagens nos pacotes **java.awt**, **java.awt.image**, **java.awt.color**, **java.awt.font**, **java.awt.geom**, **java.awt.print** e **java.awt.image.renderable**. Os recursos da API são muito amplos para serem trabalhados neste texto. Para uma visão geral dos recursos, veja a demo Java2D (demonstrada no Capítulo 3). Nesta seção, apresentamos uma visão geral dos vários recursos de Java2D.

O desenho com a API Java2D é realizado com uma instância da classe **Graphics2D** (pacote **java.awt**). A classe **Graphics2D** é uma subclasse da classe **Graphics**, portanto ela tem todos os recursos gráficos demonstrados anteriormente neste capítulo. De fato, o objeto real que utilizamos para desenhar em cada método **paint** é um objeto **Graphics2D** que é passado para o método **paint** e acessado pela referência **g** da superclasse **Graphics**. Para acessar os recursos de **Graphics2D**, devemos fazer a coerção (*downcast*) da referência **Graphics** passada a **paint** para uma referência **Graphics2D** com uma instrução como

```
Graphics2D g2d = ( Graphics2D ) g;
```

Os programas das próximas seções utilizam essa técnica.

11.9 Formas de Java2D

A seguir, apresentamos várias formas do pacote **java.awt.geom** de Java2D, incluindo **Ellipse2D.Double**, **Rectangle2D.Double**, **RoundRectangle2D.Double**, **Arc2D.Double** e **Line2D.Double**. Observe a sintaxe do nome de cada classe. Cada uma dessas classes representa uma forma com dimensões especificadas como valores em ponto flutuante de precisão dupla. Há uma versão separada de cada uma para trabalhar com valores de ponto flutuante de precisão simples (como **Ellipse2D.Float**). Em cada caso, **Double** é uma classe interna **static** da classe à esquerda do operador ponto (por exemplo, **Ellipse2D**). Para utilizar a classe interna **static**, simplesmente qualificamos seu nome com o nome da classe externa.

O programa da Fig. 11.22 demonstra várias formas e recursos de desenho de Java2D, como linhas espessas, formas preenchidas com padrões e linhas tracejadas. São apenas alguns dos muitos recursos oferecidos por Java2D.

```

1 // Fig. 11.22: Shapes.java
2 // Demonstrando algumas formas de Java2D
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.awt.geom.*;
8 import java.awt.image.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 public class Shapes extends JFrame {
14
15     // configura o String da barra de título e as dimensões da janela
16     public Shapes()
17     {
18         super( "Drawing 2D shapes" );
19

```

Fig. 11.22 Demonstrationando algumas formas de Java2D (parte 1 de 3).

```

20     setSize( 425, 160 );
21     setVisible( true );
22 }
23
24 // desenha formas com a Java2D API
25 public void paint( Graphics g )
26 {
27     // chama o método paint da superclasse
28     super.paint( g );
29
30     // cria 2D convertendo g para Graphics2D
31     Graphics2D g2d = ( Graphics2D ) g;
32
33     // desenha elipse 2D preenchida com um gradiente de azul até amarelo
34     g2d.setPaint( new GradientPaint( 5, 30, Color.blue, 35,
35         100, Color.yellow, true ) );
36     g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
37
38     // desenha retângulo 2D em vermelho
39     g2d.setPaint( Color.red );
40     g2d.setStroke( new BasicStroke( 10.0f ) );
41     g2d.draw( new Rectangle2D.Double( 80, 30, 65, 100 ) );
42
43     // desenha retângulo arredondado 2D com um fundo buferizado
44     BufferedImage buffImage = new BufferedImage(
45         10, 10, BufferedImage.TYPE_INT_RGB );
46
47     Graphics2D gg = buffImage.createGraphics();
48     gg.setColor( Color.yellow );    // desenha em amarelo
49     gg.fillRect( 0, 0, 10, 10 );    // desenha um retângulo preenchido
50     gg.setColor( Color.black );    // desenha em preto
51     gg.drawRect( 1, 1, 6, 6 );    // desenha um retângulo
52     gg.setColor( Color.blue );    // desenha em azul
53     gg.fillRect( 1, 1, 3, 3 );    // desenha um retângulo preenchido
54     gg.setColor( Color.red );    // desenha em vermelho
55     gg.fillRect( 4, 4, 3, 3 );    // desenha um retângulo preenchido
56
57     // pinta buffImage sobre o JFrame
58     g2d.setPaint( new TexturePaint(
59         buffImage, new Rectangle( 10, 10 ) ) );
60     g2d.fill( new RoundRectangle2D.Double(
61         155, 30, 75, 100, 50, 50 ) );
62
63     // desenha arco em forma de torta 2D em branco
64     g2d.setPaint( Color.white );
65     g2d.setStroke( new BasicStroke( 6.0f ) );
66     g2d.draw( new Arc2D.Double(
67         240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
68
69     // desenha linhas 2D em verde e amarelo
70     g2d.setPaint( Color.green );
71     g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
72
73     float dashes[] = { 10 };
74
75     g2d.setPaint( Color.yellow );
76     g2d.setStroke( new BasicStroke( 4, BasicStroke.CAP_ROUND,
77         BasicStroke.JOIN_ROUND, 10, dashes, 0 ) );
78     g2d.draw( new Line2D.Double( 320, 30, 395, 150 ) );
79 }
```

Fig. 11.22 Demonstrando algumas formas de Java2D (parte 2 de 3).

```

80
81     // executa o aplicativo
82     public static void main( String args[] )
83     {
84         Shapes application = new Shapes();
85
86         application.setDefaultCloseOperation(
87             JFrame.EXIT_ON_CLOSE );
88     }
89
90 } // fim da classe Shapes

```

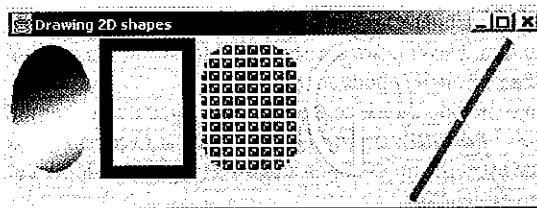


Fig. 11.22 Demonstrando algumas formas de Java2D (parte 3 de 3).

A linha 31 converte a referência `Graphics` recebida por `paint` para uma referência `Graphics2D` e a atribui a `g2d` para permitir acesso aos recursos de `Java2D`.

A primeira forma que desenhamos é uma elipse preenchida com cores que mudam gradualmente. As linhas 34 e 35 invocam o método `setPaint` de `Graphics2D` para configurar o objeto `Paint` que determina a cor a exibir na forma. O objeto `Paint` é um objeto de qualquer classe que implementa a interface `java.awt.Paint`. O objeto `Paint` pode ser algo tão simples como um dos objetos `Color` predefinidos apresentados na Seção 11.3 (a classe `Color` implementa `Paint`) ou o objeto `Paint` pode ser uma instância das classes da API Java2D `GradientPaint`, `SystemColor` ou `TexturePaint`. Nesse caso, utilizamos um objeto `GradientPaint`.

A classe `GradientPaint` ajuda a desenhar uma forma em cores que se alteram gradualmente – o que é chamado de *gradiente*. O construtor `GradientPaint` utilizado aqui exige sete argumentos. Os primeiros dois argumentos especificam a coordenada inicial para o gradiente. O terceiro argumento especifica a `Color` inicial para o gradiente. O quarto e o quinto argumentos especificam a coordenada final para o gradiente. O sexto argumento especifica a `Color` final para o gradiente. O último argumento especifica se o gradiente é cíclico (`true`) ou acíclico (`false`). As duas coordenadas determinam a direção do gradiente. Como a segunda coordenada (35, 100) está abaixo e à direita da primeira coordenada (5, 30), o gradiente segue para baixo e para a direita em um ângulo. Como esse gradiente é cíclico (`true`), a cor inicia com azul, gradualmente torna-se amarelo, depois gradualmente volta para azul. Se o gradiente for acíclico, a transição de cor seria da primeira cor especificada (por exemplo, azul) para a segunda cor (por exemplo, amarelo).

A linha 36 utiliza o método `fill` de `Graphics2D` para desenhar um objeto `Shape` preenchido. O objeto `Shape` é uma instância de qualquer classe que implementa a interface `Shape` (pacote `java.awt`) – nesse caso, uma instância da classe `Ellipse2D.Double`. O construtor `Ellipse2D.Double` recebe quatro argumentos que especificam o retângulo delimitador da elipse a ser exibido.

Em seguida, desenhamos um retângulo vermelho com uma borda grossa. A linha 39 utiliza `setPaint` para configurar o objeto `Paint` para `Color.red`. A linha 40 utiliza o método `setStroke` de `Graphics2D` para configurar as características da borda do retângulo (ou das linhas para qualquer outra forma). O método `setStroke` exige um objeto `Stroke` como argumento. O objeto `Stroke` é uma instância de qualquer classe que implementa a interface `Stroke` (pacote `java.awt`) – nesse caso, uma instância da classe `BasicStroke`. A classe `BasicStroke` fornece diversos construtores para especificar a largura da linha, como as linhas terminam (denominadas *terminações de linha*), como as linhas se juntam (denominadas *junções de linha*) e os atributos de traço da linha (se for uma linha tracejada). O construtor aqui especifica que a linha deve ter 10 *pixels* de largura.

A linha 41 utiliza o método `draw` de `Graphics2D` para desenhar um objeto `Shape` – nesse caso, uma instância da classe `Rectangle2D.Double`. O construtor `Rectangle2D.Double` recebe quatro argumentos que especificam a coordenada superior esquerda *x*, a coordenada superior esquerda *y*, a largura e a altura do retângulo.

Em seguida, desenhamos um retângulo arredondado preenchido com um padrão criado em um objeto **BufferedImage** (pacote `java.awt.image`). As linhas 44 e 45 criam o objeto **BufferedImage**. Pode-se utilizar a classe **BufferedImage** para produzir imagens coloridas e em escala de tons de cinza. Essa **BufferedImage** em particular tem 10 pixels de altura e 10 pixels de largura. O terceiro argumento do construtor **BufferedImage.TYPE_INT_RGB** indica que a imagem é armazenada em cores utilizando o esquema de cores RGB.

Para criar o padrão de preenchimento do retângulo arredondado, devemos primeiro desenhar na **BufferedImage**. A linha 47 cria um objeto **Graphics2D** que pode ser utilizado para desenhar na **BufferedImage**. As linhas 48 a 55 utilizam os métodos **setColor**, **fillRect** e **drawRect** (discutidos anteriormente neste capítulo) para criar o padrão.

As linhas 58 e 59 configuram o objeto **Paint** como um novo objeto **TexturePaint** (pacote `java.awt`). O objeto **TexturePaint** utiliza a imagem armazenada em sua **BufferedImage** associada como a textura de preenchimento para uma forma preenchida. O segundo argumento especifica a área **Rectangle** da **BufferedImage** que será replicada na textura. Nesse caso, **Rectangle** tem o mesmo tamanho que a **BufferedImage**. Mas uma parte menor da **BufferedImage** pode ser utilizada.

As linhas 60 e 61 utilizam o método **fill** de **Graphics2D** para desenhar um objeto **Shape** preenchido – nesse caso, uma instância da classe **RoundRectangle2D.Double**. O construtor **RoundRectangle2D.Double** recebe seis argumentos que especificam as dimensões do retângulo e a largura e a altura do arco utilizadas para determinar o arredondamento dos cantos.

Em seguida, desenhamos um arco em forma de torta com uma linha branca espessa. A linha 64 configura o objeto **Paint** como **Color.white**. A linha 65 configura o objeto **Stroke** de um novo **BasicStroke** como uma linha de 6 pixels de largura. As linhas 66 e 67 utilizam o método **draw** de **Graphics2D** para desenhar um objeto **Shape** – nesse caso, um **Arc2D.Double**. Os primeiros quatro argumentos do construtor **Arc2D.Double** especificam a coordenada superior esquerda *x*, a coordenada superior esquerda *y*, a largura e a altura do retângulo delimitador para o arco. O quinto argumento especifica o ângulo inicial. O sexto argumento especifica o ângulo do arco. O último argumento especifica a maneira como o arco é fechado. A constante **Arc2D.PIE** indica que o arco é fechado desenhando-se duas linhas. Uma linha do ponto inicial do arco ao centro do retângulo delimitador e uma linha do centro do retângulo delimitador ao ponto terminal. A classe **Arc2D** fornece duas outras constantes **static** para especificar como o arco é fechado. A constante **Arc2D.CHORD** desenha uma linha do ponto inicial ao ponto final. A constante **Arc2D.OPEN** especifica que o arco não é fechado.

Por fim, desenhamos duas linhas que utilizam objetos **Line2D** – um sólido e um tracejado. A linha 70 configura o objeto **Paint** como **Color.green**. A linha 71 utiliza o método **draw** de **Graphics2D** para desenhar um objeto **Shape** – nesse caso, uma instância da classe **Line2D.Double**. Os argumentos do construtor **Line2D.Double** especificam as coordenadas iniciais e coordenadas finais da linha.

A linha 73 define um *array* de um elemento **float** que contém o valor 10. Esse *array* será utilizado para descrever os traços na linha tracejada. Nesse caso, cada traço terá 10 pixels de comprimento. Para criar traços de comprimentos diferentes em um padrão, simplesmente forneça os comprimentos de cada traço como elemento no *array*. A linha 75 configura o objeto **Paint** como **Color.yellow**. As linhas 76 e 77 configuram o objeto **Stroke** com um novo **BasicStroke**. A linha terá 4 pixels de largura e terá terminações arredondadas (**BasicStroke.CAP_ROUND**). Se as linhas se juntam (como em um vértice de retângulo), as linhas de junção serão arredondadas (**BasicStroke.JOIN_ROUND**). O argumento **dashes** especifica os comprimentos dos traços da linha. O último argumento indica o subscrito inicial do *array* **dashes** para o primeiro traço do padrão. A linha 78 então desenha uma linha com o **Stroke** atual.

A seguir apresentamos um caminho geral – uma forma construída com linhas retas e curvas complexas. Um caminho geral é representado com um objeto da classe **GeneralPath** (pacote `java.awt.geom`). O programa da Fig. 11.23 demonstra o desenho de um caminho geral na forma de uma estrela de cinco pontas.

```

1 // Fig. 11.23: Shapes2.java
2 // Demonstrando um caminho geral
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.awt.geom.*;

```

Fig. 11.23 Demonstrando algumas formas de Java2D (parte 1 de 3).

```

8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class Shapes2 extends JFrame {
12
13     // configura o String da barra de título, a cor de fundo
14     // e as dimensões da janela
15     public Shapes2()
16     {
17         super( "Drawing 2D Shapes" );
18
19         getContentPane().setBackground( Color.yellow );
20         setSize( 400, 400 );
21         setVisible( true );
22     }
23
24
25     // desenha caminhos gerais
26     public void paint( Graphics g )
27     {
28         // chama o método paint da superclasse
29         super.paint( g );
30
31         int xPoints[] =
32             { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
33         int yPoints[] =
34             { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };
35
36         Graphics2D g2d = ( Graphics2D ) g;
37
38         // cria uma estrela a partir de uma série de pontos
39         GeneralPath star = new GeneralPath();
40
41         // configura a coordenada inicial do caminho geral
42         star.moveTo( xPoints[ 0 ], yPoints[ 0 ] );
43
44         // cria a estrela - isto não desenha a estrela
45         for ( int count = 1; count < xPoints.length; count++ )
46             star.lineTo( xPoints[ count ], yPoints[ count ] );
47
48         // fecha a forma
49         star.closePath();
50
51         // faz translação da origem para (200, 200)
52         g2d.translate( 200, 200 );
53
54         // faz rotação em torno da origem e desenha estrelas em cores aleatórias
55         for ( int count = 1; count <= 20; count++ ) {
56
57             // faz rotação do sistema de coordenadas
58             g2d.rotate( Math.PI / 10.0 );
59
60             // configura cor de desenho aleatória
61             g2d.setColor( new Color(
62                 ( int )( Math.random() * 256 ),
63                 ( int )( Math.random() * 256 ),
64                 ( int )( Math.random() * 256 ) ) );
65
66             // desenha estrela cheia
67             g2d.fill( star );

```

Fig. 11.23 Demonstrando algumas formas de Java2D (parte 2 de 3).

```

68     }
69
70 } // fim do método paint
71
72 // executa o aplicativo
73 public static void main( String args[] )
74 {
75     Shapes2 application = new Shapes2();
76
77     application.setDefaultCloseOperation(
78         JFrame.EXIT_ON_CLOSE );
79 }
80
81 } // fim da classe Shapes2

```



Fig. 11.23 Demonstrando algumas formas de Java2D (parte 3 de 3).

As linhas 31 a 34 definem dois *arrays int* que representam as coordenadas *x* e *y* dos pontos da estrela. A linha 39 define o objeto **GeneralPath star**.

A linha 42 utiliza o método *moveTo* de **GeneralPath** para especificar o primeiro ponto na **star**. A estrutura **for** nas linhas 45 e 46 utiliza o método *lineTo* de **GeneralPath** para desenhar uma linha até o próximo ponto na **star**. Cada nova chamada a *lineTo* desenha uma linha do ponto anterior ao ponto atual. A linha 49 utiliza o método *closePath* de **GeneralPath** para desenhar uma linha do último ponto ao ponto especificado na última chamada a *moveTo*. Isso completa o caminho geral.

A linha 52 utiliza o método *translate* de **Graphics2D** para mover a origem do desenho para a posição (200, 200). Todas as operações de desenho agora utilizam a posição (200, 200) como (0, 0).

A estrutura **for** nas linhas 55 a 68 desenha a **star** 20 vezes rotacionando-a em torno do novo ponto de origem. A linha 58 utiliza o método *rotate* de **Graphics2D** para rotacionar a próxima forma exibida. O argumento especifica o ângulo de rotação em radianos (com $360^\circ = 2\pi$ radianos). A linha 67 utiliza o método *fill* de **Graphics2D** para desenhar uma versão preenchida da **star**.

11.10 (Estudo de caso opcional) Pensando em objetos: projetando interfaces com a UML

Na Seção 10.22, incorporamos o tratamento de eventos à nossa simulação, modificando o diagrama de colaborações que trata dos passageiros que entram no elevador e saem dele. Incluímos tanto o tratamento de eventos quanto a herança naquele diagrama. O **Elevator** informa à sua **Door** da chegada do **Elevator**. Esta **Door** abre a **Door**

do **Floor** de chegada obtendo seu *handle* através de um objeto **Location** (que foi incluído no evento de chegada) e até dois objetos **Person** podem entrar no **Elevator** e sair dele depois que as duas **Doors** se abrem. Também discutimos interfaces *listener*. Nessa seção, representamos nossa interface *listener* com a UML.

Realizações

A UML expressa o relacionamento entre uma classe e uma interface através de uma *realização*. A classe *realiza*, ou implementa, os comportamentos de uma interface. O diagrama de classes pode mostrar uma realização entre classes e interfaces. Como mencionado na Seção 3.8 de “Pensando em objetos”, a UML oferece duas notações para desenhar um diagrama de classes – o diagrama completo e o diagrama elidido (condensado). A Fig. 11.24 mostra o diagrama de classes completo que modela a realização entre a classe **Person** e a interface **DoorListener**. O diagrama é semelhante ao diagrama de generalizações, exceto pelo fato de que a seta que expressa o relacionamento é tracejada, e não linha cheia. Observe que o compartimento do meio na interface **DoorListener** está vazio, porque as interfaces não contêm variáveis – as interfaces podem conter constantes, mas a interface **DoorListener** não contém nenhuma constante. Por último, note a palavra “interface” colocada entre os símbolos de aspas francesas (« ») localizados no primeiro compartimento da interface **DoorListener**. Esta notação distingue a interface **DoorListener** como uma interface em nosso sistema. Itens colocados entre « e » são chamados de *estereótipos* na UML. Um estereótipo indica o papel – ou finalidade – de um elemento em um diagrama UML.

A Fig. 11.25 mostra a maneira alternativa de representar a realização da classe **Person** e da interface **DoorListener** na UML. A Fig. 11.25 é o diagrama elidido da Fig. 11.24. O círculo pequeno representa a interface e a linha cheia representa a realização. Ocultando suas operações, condensamos a interface, tornando-a mais fácil de ler; entretanto, ao fazer isso, sacrificamos as informações sobre seus comportamentos. Quando se constrói um diagrama elidido, é prática comum colocar as informações relativas a qualquer comportamento em um diagrama separado – por exemplo, colocamos a classe **DoorListener** completa no diagrama de classes da Fig. 11.28.

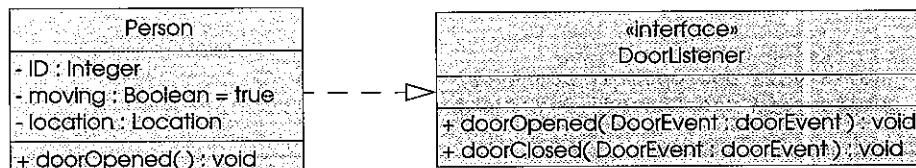


Fig. 11.24 Diagrama de classes que modela a classe **Person** que realiza a interface **DoorListener**.

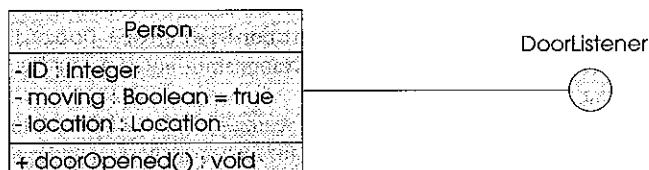


Fig. 11.25 Diagrama elidido de classes que modela a classe **Person** que realiza a interface **DoorListener**.

A engenharia progressiva (*forward engineering*) a partir da UML para o código Java implementado se beneficia de diagramas de realizações bem construídos. Quando declarar qualquer classe especifique a realização entre a classe e sua interface – aquela classe irá “implementar” a interface e sobreescriver os métodos da interface. Por exemplo, usamos a Fig. 11.24 para começar a construir **Person.java**:

```
public class Person implements DoorListener {
    // construtor
    public Person() {}

    // métodos de DoorListener
    public void doorOpened( DoorEvent doorEvent ) {}
    public void doorClosed( DoorEvent doorEvent ) {}
}
```

A Fig. 11.26 mostra a implementação completa em Java para a Fig. 11.24. As linhas 6 a 8 e as linhas 14 e 15 incluem os atributos e as operações de **Person**, respectivamente – neste caso, a operação **doorOpened** (linha 14) já estava incluída quando implementamos a interface **DoorListener**, de modo que incluímos somente os atributos de **Person**:

```
1 // Person.java
2 // Gerada a partir da Fig. 11.24
3 public class Person implements DoorListener {
4
5     // atributos
6     private int ID;
7     private boolean moving = true;
8     private Location location;
9
10    // construtor
11    public Person() {}
12
13    // métodos de DoorListener
14    public void doorOpened( DoorEvent doorEvent ) {}
15    public void doorClosed( DoorEvent doorEvent ) {}
16 }
```

Fig. 11.26 A classe **Person** é gerada a partir da Fig. 11.24.

Quando uma **Door** abre ou fecha, ela invoca somente aqueles métodos declarados na interface **DoorListener**, mas somente se a **Person** se registrou com aquela **Door** para receber **DoorEvents**. Finalmente, apresentamos o diagrama elidido de classes que modela as realizações em nosso modelo de elevador na Fig. 11.27 – o diagrama elidido não contém nenhum método de interface (o que torna o diagrama mais fácil de ler), de modo que apresentamos o diagrama de classes para as interfaces na Fig. 11.28, que mostra todos os métodos de interface. Consulte estes diagramas quando estiver estudando a implementação da simulação de elevador nos Apêndices G, H e I.

De acordo com a Fig. 11.27, as classes **Door**, **Light**, **Bell** e **Button** implementam a interface **ElevatorMoveListener**. A classe **Elevator** implementa as interfaces **ButtonListener**, **DoorListener** e **BellListener**. A classe **ElevatorModel** implementa a interface **PersonMoveListener**. A classe **ElevatorShaft** implementa as interfaces **LightListener**, **ButtonListener** e **DoorListener**. Por último, a classe **Person** implementa a interface **DoorListener**. Examinamos novamente a Fig. 11.27 no Apêndice H, quando começarmos a codificar nosso modelo.

Nessa seção mostramos como representar as interfaces e realizações com a UML. Também apresentamos diagramas de classes que mostram as interfaces *listener* e suas realizações para nossa simulação de elevador. Na Seção 12.16 de “Pensando em objetos”, modelamos como o usuário interage com nossa simulação.

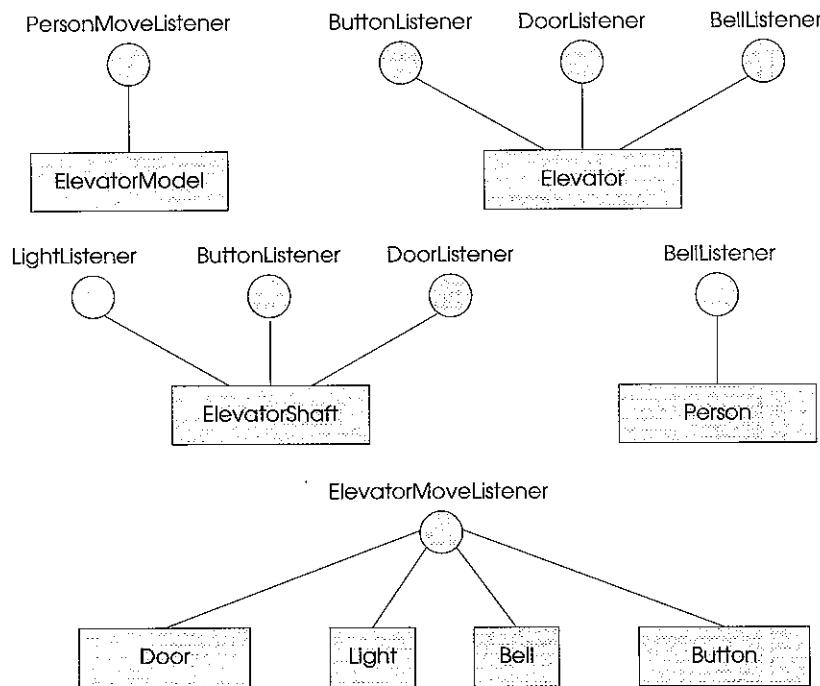


Fig. 11.27 Diagrama de classes que modela as realizações no modelo do elevador.

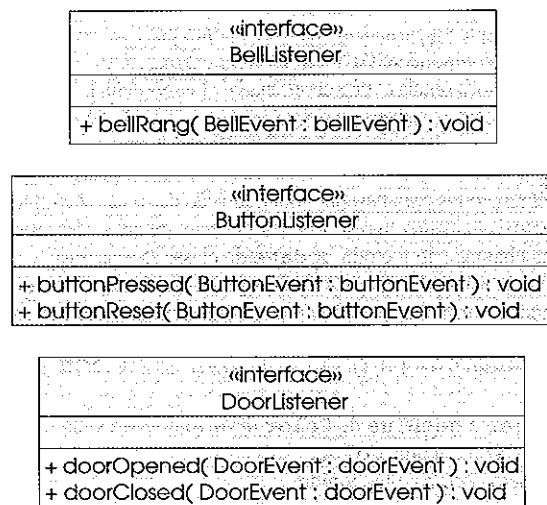


Fig. 11.28 Diagrama de classes para as interfaces *listener* (parte 1 de 2).

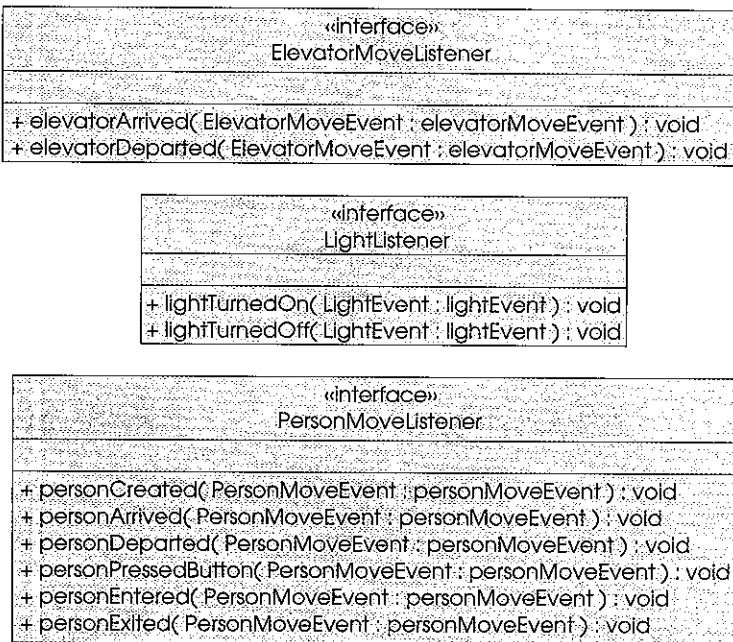


Fig. 11.28 Diagrama de classes para as interfaces *listener* (parte 2 de 2).

Resumo

- O sistema de coordenadas é um esquema para identificar cada ponto possível na tela.
- O canto superior esquerdo de um componente GUI tem as coordenadas $(0, 0)$. O par de coordenadas é composto por uma coordenada x (a coordenada horizontal) e uma coordenada y (a coordenada vertical).
- As unidades coordenadas são medidas em *pixels*. O *pixel* é a menor unidade de resolução do monitor.
- Um contexto gráfico Java permite desenhar na tela. Um objeto **Graphics** gerencia um contexto gráfico que controla como as informações são desenhadas.
- Os objetos **Graphics** contêm métodos para desenho, manipulação de fontes, manipulação de cores, etc.
- O método **paint** normalmente é chamado em resposta a um *evento*, tal como descobrir uma janela.
- O método **repaint** faz uma chamada ao método **update** da classe **Component** logo que possível para limpar o fundo do **Component** de qualquer desenho anterior, então **update** chama **paint** diretamente.
- A classe **Color** define métodos e constantes para manipular cores em um programa Java.
- Java utiliza cores RGB, em que os componentes de cor vermelho, verde e azul são inteiros no intervalo 0 a 255 ou valores em ponto flutuante no intervalo 0.0 a 1.0. Quanto maior for o valor de RGB, maior será a quantidade dessa cor em particular.
- Os métodos **getRed**, **getGreen** e **getBlue** de **Color** devolvem valores inteiros entre 0 e 255, representando a quantidade de vermelho, verde e azul em uma **Color**.
- A classe **Color** fornece 13 objetos **Color** predefinidos.
- O método **getColor** de **Graphics** devolve um objeto **Color** que representa a cor de desenho atual. O método **setColor** de **Graphics** configura a cor de desenho atual.
- Java fornece a classe **JColorChooser** para exibir um diálogo para selecionar cores.
- O método **static showDialog** da classe **JColorChooser** exibe um diálogo de seleção de cor. Esse método devolve o objeto **Color** selecionado (ou **null**, se nenhum for selecionado).

- O diálogo de padrão **JColorChooser** permite selecionar uma cor a partir de uma variedade de amostras de cor. A guia **HSB** permite selecionar uma cor baseada em tom, saturação e brilho. A guia **RGB** permite selecionar uma cor utilizando controles deslizantes para os componentes vermelho, verde e azul da cor.
- O método **setBackground** de **Component** (um dos muitos métodos de **Component** que podem ser utilizados para a maioria dos componentes GUI) altera a cor de fundo de um componente.
- O construtor da classe **Font** recebe três argumentos – o nome, o estilo e o tamanho da fonte. O nome da fonte é qualquer fonte atualmente suportada pelo sistema. O estilo da fonte é **Font.PLAIN**, **Font.ITALIC** ou **Font.BOLD**. O tamanho da fonte é medido em pontos.
- O método **setFont** de **Graphics** configura a fonte de desenho.
- A classe **FontMetrics** define vários métodos para obter medidas das fontes.
- O método **getFontMetrics** de **Graphics** sem argumentos obtém o objeto **FontMetrics** para a fonte atual. O método **getFontMetrics** de **Graphics** que recebe um argumento **Font** devolve um objeto **FontMetrics** correspondente.
- Os métodos **draw3DRect** e **fill3DRect** aceitam cinco argumentos que especificam o canto superior esquerdo do retângulo, a **width** e a **height** do retângulo e se o retângulo está em alto-relevo (**true**) ou baixo-relevo (**false**).
- Os métodos **drawRoundRect** e **fillRoundRect** desenham retângulos com cantos arredondados. Seus primeiros dois argumentos especificam o canto superior esquerdo, o terceiro e quarto argumentos especificam a **width** e a **height** e os últimos dois argumentos – **arcWidth** e **arcHeight** – determinam os diâmetros vertical e horizontal dos arcos utilizados para representar os cantos.
- Os métodos **drawOval** e **fillOval** aceitam os mesmos argumentos – a coordenada do canto superior esquerdo e a **width** e a **height** do retângulo delimitador que contém a elipse.
- O arco faz parte de uma elipse. Os arcos varrem a partir de um ângulo inicial o número de graus especificado por seu ângulo de arco. O ângulo inicial especifica onde o arco inicia e o ângulo de arco especifica o número total de graus que o arco varre. Os arcos que varrem no sentido anti-horário são medidos em graus positivos e os que varrem no sentido horário são medidos em graus negativos.
- Os métodos **drawArc** e **fillArc** aceitam os mesmos argumentos – a coordenada do canto superior esquerdo, a **width** e a **height** do retângulo delimitador que contém o arco e o **startAngle** e **arcAngle**, que definem a varredura do arco.
- Os polígonos são formas de múltiplos lados. As polilinhas são séries de pontos conectados.
- O construtor **Polygon** recebe um *array* que contém a coordenada *x* de cada ponto, um *array* que contém a coordenada *y* de cada ponto e o número de pontos no polígono.
- Uma versão do método **drawPolygon** de **Graphics** exibe um objeto **Polygon**. Outra versão recebe um *array* que contém a coordenada *x* de cada ponto, um *array* que contém a coordenada *y* de cada ponto e o número de pontos no polígono e exibe o polígono correspondente.
- O método **drawPolyline** de **Graphics** exibe uma série de linhas conectadas especificada por seus argumentos (um *array* que contém a coordenada *x* de cada ponto, um *array* que contém a coordenada *y* de cada ponto e o número de pontos).
- O método **addPoint** de **Polygon** adiciona pares de *coordenadas x* e *y* para um **Polygon**.
- A API Java2D fornece recursos gráficos bidimensionais avançados para processamento de *line art*, texto e imagens.
- Para acessar os recursos de **Graphics2D**, faça a conversão da referência **Graphics** passada a **paint** para uma referência **Graphics2D**, como em `(Graphics2D) g`.
- O método **setPaint** de **Graphics2D** configura o objeto **Paint** que determina a cor e a textura da forma a ser exibida. O objeto **Paint** é um objeto de qualquer classe que implementa a interface `java.awt.Paint`. O objeto **Paint** pode ser uma classe **Color** ou uma instância das classes **GradientPaint**, **SystemColor** ou **TexturePaint** da API Java2D.
- A classe **GradientPaint** desenha uma forma em uma cor que se altera gradualmente – o que é chamado de gradiente.
- O método **fill** de **Graphics2D** desenha um objeto **Shape** preenchido. O objeto **Shape** é uma instância de qualquer classe que implementa a interface **Shape**.
- O construtor **Ellipse2D.Double** recebe quatro argumentos que especificam o retângulo delimitador da elipse a ser exibida.
- O método **setStroke** de **Graphics2D** configura as características das linhas utilizadas para desenhar uma forma. O método **setStroke** exige um objeto **Stroke** como argumento. O objeto **Stroke** é uma instância de qualquer classe que implementa a interface **Stroke**, como um **BasicStroke**.
- O método **draw** de **Graphics2D** desenha um objeto **Shape**. O objeto **Shape** é uma instância de qualquer classe que implementa a interface **Shape**.

- O construtor `Rectangle2D.Double` recebe quatro argumentos que especificam a coordenada superior esquerda *x*, a coordenada superior esquerda *y*, a largura e a altura do retângulo.
- A classe `BufferedImage` pode ser utilizada para produzir imagens coloridas e em escala de tons de cinza.
- O objeto `TexturePaint` utiliza a imagem armazenada em sua `BufferedImage` associada como a textura de preenchimento para uma forma preenchida.
- O construtor `RoundRectangle2D.Double` recebe seis argumentos que especificam as dimensões do retângulo e a largura e a altura do arco utilizados para determinar o arredondamento dos cantos.
- Os primeiros quatro argumentos do construtor `Arc2D.Double` especificam a coordenada superior esquerda *x*, a coordenada superior esquerda *y*, a largura e a altura do retângulo delimitador para o arco. O quinto argumento especifica o ângulo inicial. O sexto argumento especifica o ângulo final. O último argumento especifica o tipo de arco (`Arc2D.PIE`, `Arc2D.CHORD` ou `Arc2D.OPEN`).
- Os argumentos do construtor `Line2D.Double` especificam as coordenadas inicial e final da linha.
- Um caminho geral é uma forma construída com linhas retas e curvas complexas representadas com um objeto da classe `GeneralPath` (pacote `java.awt.geom`).
- O método `moveTo` de `GeneralPath` especifica o primeiro ponto em um caminho geral. O método `lineTo` de `GeneralPath` desenha uma linha até o próximo ponto no caminho geral. Cada nova chamada a `lineTo` desenha uma linha do ponto anterior ao ponto atual. O método `closePath` de `GeneralPath` desenha uma linha a partir do último ponto até o ponto especificado na última chamada a `moveTo`.
- O método `translate` de `Graphics2D` move a origem do desenho para uma nova posição. Todas as operações de desenho agora utilizam essa posição como (0, 0).
- O método `rotate` de `Graphics2D` é usado para fazer uma rotação do próximo objeto a ser exibido. Seu argumento especifica o ângulo de rotação em radianos.

Terminologia

<i>altura do arco</i>	<i>fonte SansSerif</i>
<i>ângulo</i>	<i>fonte Serif</i>
<i>API Java2D</i>	<i>grau</i>
<i>arco limitado por um retângulo</i>	<i>graus negativos</i>
<i>arco varrido por um ângulo</i>	<i>graus positivos</i>
<i>classe Arc2D.Double</i>	<i>interface Paint</i>
<i>classe BufferedImage</i>	<i>interface Shape</i>
<i>classe Color</i>	<i>interface Stroke</i>
<i>classe Component</i>	<i>largura de arco</i>
<i>classe Ellipse2D.Double</i>	<i>linha de base</i>
<i>classe FontMetrics</i>	<i>medidas de fontes</i>
<i>classe GeneralPath</i>	<i>método closePath</i>
<i>classe GradientPaint</i>	<i>método draw</i>
<i>classe Graphics</i>	<i>método draw3DRect</i>
<i>classe Graphics2D</i>	<i>método drawArc</i>
<i>classe Rectangle2D.Double</i>	<i>método drawLine</i>
<i>classe RoundRectangle2D.Double</i>	<i>método drawOval</i>
<i>classe SystemColor</i>	<i>método drawPolygon</i>
<i>classe TexturePaint</i>	<i>método drawPolyline</i>
<i>componente vertical</i>	<i>método drawRect</i>
<i>contexto gráfico</i>	<i>método drawRoundRect</i>
<i>coordenada</i>	<i>método fill</i>
<i>coordenada x</i>	<i>método fill3DRect</i>
<i>coordenada y</i>	<i>método fillArc</i>
<i>cor de fundo</i>	<i>método fillOval</i>
<i>descida</i>	<i>método fillPolygon</i>
<i>desenhar um arco</i>	<i>método fillRect</i>
<i>eixo x</i>	<i>método fillRoundRect</i>
<i>eixo y</i>	<i>método getAscent</i>
<i>entrelinha</i>	<i>método getBlue</i>
<i>fonte</i>	<i>método getDescent</i>
<i>fonte Monospaced</i>	<i>método getFamily</i>

<i>método</i> <code>getFont</code>	<i>método</i> <code>setFont</code>
<i>método</i> <code>getFontList</code>	<i>método</i> <code>setPaint</code>
<i>método</i> <code>getFontMetrics</code>	<i>método</i> <code>setStroke</code>
<i>método</i> <code>getGreen</code>	<i>método</i> <code>translate</code>
<i>método</i> <code>getHeight</code>	<i>método</i> <code>update</code>
<i>método</i> <code>getLeading</code>	<i>nome de fontes</i>
<i>método</i> <code>getName</code>	<i>objeto de imagens gráficas</i>
<i>método</i> <code>getRed</code>	<i>pixel</i>
<i>método</i> <code>getSize</code>	<i>polígono</i>
<i>método</i> <code>getStyle</code>	<i>polígono fechado</i>
<i>método</i> <code>isBold</code>	<i>polígono preenchido</i>
<i>método</i> <code>isItalic</code>	<i>ponto</i>
<i>método</i> <code>isPlain</code>	<i>processo baseado em eventos</i>
<i>método</i> <code>lineTo</code>	<i>retângulo delimitador</i>
<i>método</i> <code>moveTo</code>	<i>sistema de coordenadas</i>
<i>método</i> <code>paint</code>	<i>subida</i>
<i>método</i> <code>repaint</code>	<i>valor RGB</i>
<i>método</i> <code>rotate</code>	
<i>método</i> <code>setColor</code>	

Exercícios de auto-revisão

- 11.1** Preencha as lacunas em cada uma das frases seguintes:
- Em Java2D, o método _____ da classe _____ configura as características de uma linha utilizada para desenhar uma forma.
 - A classe _____ ajuda a definir o preenchimento para uma forma, de tal modo que o preenchimento gradualmente muda de uma cor para outra.
 - O método _____ da classe **Graphics** desenha uma linha entre dois pontos.
 - RGB é um acrônimo para _____, _____ e _____.
 - Os tamanhos das fontes são medidos em unidades chamadas _____.
 - A classe _____ ajuda a definir o preenchimento para uma forma que utiliza um padrão desenhado em uma **BufferedImage**.
- 11.2** Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- Os dois primeiros argumentos do método **drawOval** de **Graphics** especificam a coordenada do centro da elipse.
 - No sistema de coordenadas Java, os valores *x* aumentam da esquerda para a direita.
 - O método **fillPolygon** desenha um polígono sólido na cor atual.
 - O método **drawArc** permite ângulos negativos.
 - O método **getSize** devolve o tamanho da fonte atual em centímetros.
 - A coordenada de *pixel* (0, 0) localiza-se exatamente no centro do monitor.
- 11.3** Encontre o(s) erro(s) em cada uma das instruções seguintes e explique como corrigir o(s) erro(s). Pressuponha que *g* seja um objeto **Graphics**.
- `g.setFont("SansSerif");`
 - `g.erase(x, y, w, h); // apaga o retângulo em (x, y)`
 - `Font f = new Font("Serif", Font.BOLDITALIC, 12);`
 - `g.setColor(Color.Yellow); // muda a cor para amarelo`

Respostas aos exercícios de auto-revisão

- 11.1** a) `setStroke`, **Graphics2D**. b) **GradientPaint**. c) **drawLine**. d) *Red*, *Green*, *Blue*. e) pontos. f) **TexturePaint**.
- 11.2** a) Falsa. Os dois primeiros argumentos especificam o canto superior esquerdo do retângulo delimitador.
 b) Verdadeira.
 c) Verdadeira.
 d) Verdadeira.
 e) Falsa. Os tamanhos das fontes são medidos em pontos.

- f) Falsa. A coordenada $(0,0)$ corresponde ao canto superior esquerdo de um componente GUI em que o desenho ocorre.
- 11.3**
- O método `setFont` recebe um objeto `Font` como argumento – não um `String`.
 - A classe `Graphics` não tem um método `erase`. Deve-se usar o método `clearRect`.
 - `Font.BOLDITALIC` não é um estilo válido de fonte. Para obter uma fonte em negrito e itálico, utilize `Font.BOLD + Font.ITALIC`.
 - `Yellow` deve iniciar com uma letra minúscula: `g.setColor(Color.yellow);`.

Exercícios

- 11.4** Preencha as lacunas em cada uma das frases seguintes:
- Utiliza-se a classe _____ da API Java2D para definir elipses.
 - Os métodos `draw` e `fill` da classe `Graphics2D` exigem um objeto do tipo _____ como argumento.
 - As três constantes que especificam o estilo de fonte são _____, _____ e _____.
 - O método `Graphics2D` _____ configura a cor de pintura para formas de Java2D.
- 11.5** Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- O método `drawPolygon` conecta automaticamente os pontos finais do polígono.
 - O método `drawLine` desenha uma linha entre dois pontos.
 - O método `fillArc` utiliza graus para especificar o ângulo.
 - No sistema de coordenadas Java, os valores y aumentam de cima para baixo.
 - A classe `Graphics` herda diretamente da classe `Object`.
 - A classe `Graphics` é uma classe `abstract`.
 - A classe `Font` herda diretamente da classe `Graphics`.
- 11.6** Escreva um programa que desenha uma série de oito círculos concêntricos. Os círculos devem ser separados por 10 pixels. Utilize o método `drawOval` da classe `Graphics`.
- 11.7** Escreva um programa que desenha uma série de oito círculos concêntricos. Os círculos devem ser separados por 10 pixels. Utilize o método `drawArc`.
- 11.8** Modifique sua solução do Exercício 11.6 para desenhar as elipses utilizando instâncias da classe `Ellipse2D.Double` e o método `draw` da classe `Graphics2D`.
- 11.9** Escreva um programa que desenha linhas de comprimentos aleatórios em cores aleatórias.
- 11.10** Modifique sua solução do Exercício 11.9 para desenhar linhas aleatórias, em cores aleatórias e espessuras de linha aleatórias. Utilize a classe `Line2D.Double` e o método `draw` da classe `Graphics2D` para desenhar as linhas.
- 11.11** Escreva um programa que exibe aleatoriamente triângulos gerados em cores diferentes. Cada triângulo deve ser preenchido com uma cor diferente. Utilize a classe `GeneralPath` e o método `fill` da classe `Graphics2D` para desenhar os triângulos.
- 11.12** Escreva um programa que desenha caracteres em tamanhos de fonte e cores diferentes aleatoriamente.
- 11.13** Escreva um programa que desenha uma grade 8 por 8. Utilize o método `drawLine`.
- 11.14** Modifique sua solução do Exercício 11.13 para desenhar a grade utilizando instâncias da classe `Line2D.Double` e o método `draw` da classe `Graphics2D`.
- 11.15** Escreva um programa que desenha uma grade 10 por 10. Utilize o método `drawRect`.
- 11.16** Modifique sua solução do Exercício 11.15 para desenhar a grade utilizando instâncias da classe `Rectangle2D.Double` e o método `draw` da classe `Graphics2D`.
- 11.17** Escreva um programa que desenha um tetraedro (uma pirâmide). Utilize a classe `GeneralPath` e o método `draw` da classe `Graphics2D`.
- 11.18** Escreva um programa que desenha um cubo. Utilize a classe `GeneralPath` e o método `draw` da classe `Graphics2D`.
- 11.19** No Exercício 3.9, você escreveu um *applet* que lê o raio de um círculo fornecido pelo usuário e exibe o diâmetro, a circunferência e a área do círculo. Modifique sua solução do Exercício 3.9 para ler um conjunto de coordenadas além

do raio. Depois desenhe o círculo e exiba o diâmetro, a circunferência e a área do círculo utilizando um objeto `Ellipse2D.Double` para representar o círculo e o método `draw` da classe `Graphics2D` para exibir o círculo.

11.20 Escreva um aplicativo que simula um protetor de tela. O aplicativo deve desenhar linhas aleatoriamente, utilizando o método `drawLine` da classe `Graphics`. Depois de desenhar 100 linhas, o aplicativo deve apagar seu próprio desenho e começar a desenhar as linhas novamente. Para permitir que o programa desenhe continuamente, coloque uma chamada a `repaint` como a última linha no método `paint`. Você encontrou algum problema com isso em seu sistema?

11.21 Eis uma pequena antecipação do que está por vir. O pacote `javax.swing` contém uma classe chamada `Timer` que é capaz de chamar o método `actionPerformed` da interface `ActionListener` a intervalos fixos de tempo (especificados em milissegundos). Modifique a solução do Exercício 11.20 para remover a chamada a `repaint` a partir do método `paint`. Defina sua classe de modo que ela implemente `ActionListener` (o método `actionPerformed` simplesmente deve chamar `repaint`). Defina uma variável de instância do tipo `Timer` chamada `timer` em sua classe. No construtor da sua classe, escreva as seguintes instruções:

```
timer = new Timer( 1000, this );
timer.start();
```

isso cria uma instância da classe `Timer` que chamará o método `actionPerformed` do objeto `this` a cada 1000 milissegundos (isto é, um segundo).

11.22 Modifique sua solução do Exercício 11.21 para permitir que o usuário digite o número de linhas aleatórias que devem ser desenhadas antes de o aplicativo apagar seu próprio desenho e começar a desenhar linhas novamente. Utilize um `JTextField` para obter o valor. O usuário deve ser capaz de digitar um novo número no `JTextField` a qualquer momento durante a execução do programa. Utilize uma definição de classe interna para realizar tratamento de eventos para o `JTextField`.

11.23 Modifique sua solução do Exercício 11.21 para escolher aleatoriamente formas diferentes a exibir (utiliza os métodos da classe `Graphics`).

11.24 Modifique sua solução do Exercício 11.23 para utilizar classes e recursos de desenho da API Java2D. Para formas como retângulos e elipses, desenhe-as com gradientes gerados aleatoriamente (utilize a classe `GradientPaint` para gerar o gradiente).

11.25 Escreva uma versão gráfica de sua solução do Exercício 6.37 – *Torres de Hanói*. Depois de estudar o Capítulo 16, você será capaz de implementar uma versão desses exercícios utilizando os recursos de imagem, animação e áudio de Java.

11.26 Modifique o programa de jogo de dados da Fig. 7.9 de modo que ele atualize as contagens para cada face do dado depois de cada lançamento. Converta o aplicativo em um aplicativo com janelas (isto é, uma subclasse de `JFrame`) e utilize o método `drawString` de `Graphics` para enviar os totais para a saída.

11.27 Modifique sua solução do Exercício 7.21 – *Gráficos de Tartaruga* – para adicionar uma interface gráfica com o usuário que utiliza `JTextFields` e `JButtons`. Além disso, desenhe linhas em vez de asteriscos (*). Quando o programa dos gráficos de tartaruga especificar um movimento, traduza o número de posições em um número de *pixels* na tela, multiplicando o número de posições por 10 (ou qualquer valor que você escolher). Implemente o desenho com recursos da API Java2D.

11.28 Produza uma versão gráfica do problema do *Passeio do Cavalo* (Exercícios 7.22, 7.23 e 7.26). À medida que cada movimento é feito, a célula apropriada do tabuleiro deve ser atualizada com o número adequado do movimento. Se o resultado do programa for um *passeio completo* ou um *passeio fechado*, o programa deve exibir uma mensagem apropriada. Se quiser, utilize a classe `Timer` (veja o Exercício 11.24) para ajudar a animar o Passeio do Cavalo. A cada segundo, deve ser feito o movimento seguinte.

11.29 Produza uma versão gráfica da simulação *A lebre e a tartaruga* (Exercício 7.41). Simule a montanha desenhando um arco que se estende da parte inferior esquerda da janela para a parte superior direita. A lebre e a tartaruga devem correr subindo a montanha. Implemente a saída gráfica de modo que a lebre e a tartaruga realmente sejam impressas no arco a cada movimento. [Nota: estenda o percurso da corrida de 70 para 300, a fim de permitir uma área gráfica maior.]

11.30 Produza uma versão gráfica do problema *Percorrendo um labirinto* (Exercícios 7.38 a 7.40). Utilize os labirintos que você produziu como guia para criar as versões gráficas. Enquanto o labirinto está sendo resolvido, um círculo pequeno deve ser exibido no labirinto para indicar a posição atual. Se quiser, utilize a classe `Timer` (veja o Exercício 11.24) para ajudar a animar o percurso do labirinto. A cada segundo, o movimento seguinte deve ser feito.

11.31 Produza uma versão gráfica da *Classificação com o Bucket Sort* (Exercício 7.28) que exiba cada valor que está sendo colocado no *depósito* apropriado e no fim seja copiado de volta para o *array* original.

11.32 Escreva um programa que utiliza o método `drawPolyline` para desenhar uma espiral.

11.33 Escreva um programa que lê quatro números e dispõe os números num gráfico em forma de torta. Utilize a classe `Arc2D.Double` e o método `fill` da classe `Graphics2D` para fazer o desenho. Desenhe cada pedaço da torta em uma cor diferente.

11.34 Escreva um *applet* que lê quatro números e dispõe os números num gráfico de barras. Utilize a classe `Rectangle2D.Double` e o método `fill` da classe `Graphics2D` para realizar o desenho. Desenhe cada barra em uma cor diferente.

12

Componentes da interface gráfica com o usuário: parte 1

Objetivos

- Entender os princípios de projeto das interfaces gráficas com o usuário.
- Ser capaz de construir interfaces gráficas com o usuário.
- Entender os pacotes que contêm componentes de GUI (interface gráfica com o usuário) e as classes e interfaces de tratamento de eventos.
- Ser capaz de criar e manipular botões, rótulos, listas, campos de texto e painéis.
- Entender eventos de mouse e eventos de teclado.
- Entender e ser capaz de utilizar gerenciadores de leiaute.

...os mais sábios profetas certificam-se primeiro do evento.
Horace Walpole

Você acha que posso ouvir essas coisas o dia todo?
Lewis Carroll

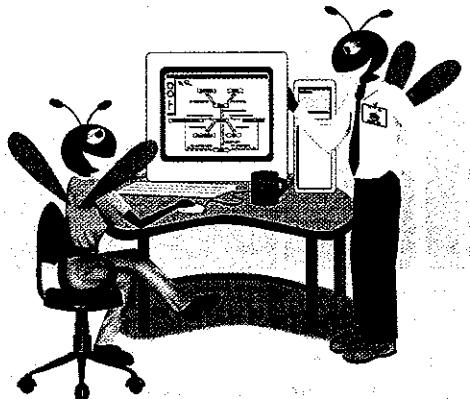
Profira a afirmativa; enfatize sua escolha manifestando ignorar todas as que você rejeitou.
Ralph Waldo Emerson

Você paga e faz a sua escolha.
Punch

Adivinhe se puder, escolha se ousar.
Pierre Corneille

Aqueles que aqui entrarem abandonem todas as suas esperanças!
Dante Alighieri

Sai, perseguido por um urso.
William Shakespeare



Súmario do capítulo

- 12.1 Introdução
- 12.2 Visão geral do Swing
- 12.3 JLabel
- 12.4 Modelo de tratamento de eventos
- 12.5 Criação de JPasswordField e JList
 - 12.5.1 Como funciona o tratamento de eventos
- 12.6 JButton
- 12.7 JCheckBox e JRadioButton
- 12.8 JComboBox
- 12.9 JList
- 12.10 Listas de seleção múltipla
- 12.11 Tratamento de eventos de mouse
- 12.12 Classes adaptadoras
- 12.13 Tratamento de eventos de teclado
- 12.14 Gerenciadores de layout
 - 12.14.1 FlowLayout
 - 12.14.2 BorderLayout
 - 12.14.3 GridLayout
- 12.15 Painéis
- 12.16 (Estudo de caso opcional) Pensando em objetos: casos de uso

*Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão
Links úteis*

12.1 Introdução

A interface gráfica com o usuário (GUI – *graphical user interface*) apresenta uma interface pictórica para um programa. A GUI fornece a um programa uma “aparência” e um “comportamento” diferenciados. Fornecendo a diferentes aplicativos um conjunto consistente de componentes intuitivos de interface com o usuário, as GUIs dão ao usuário um nível básico de familiaridade com um programa sem que ele jamais tenha usado o programa. Por outro lado, isto reduz o tempo exigido dos usuários para aprender a usar um programa e aumenta a sua habilidade de usar este programa de uma maneira produtiva.



Observação de aparência e comportamento 12.1

Interfaces com o usuário consistentes também permitem que os usuários aprendam mais rapidamente novos aplicativos.

Como exemplo de uma GUI, a Fig. 12.1 contém uma janela do Netscape Navigator com alguns de seus componentes GUI rotulados. Na janela, há uma *barra de menus* que contém os *menus* (**File**, **Edit**, **View**, etc.). Embaixo da barra de menus, há um conjunto de *botões* em que cada um tem uma tarefa definida no Netscape Navigator. Embaixo dos botões há um *campo de texto* em que o usuário pode digitar o nome do *site* da World Wide Web a visitar. Os menus, botões, campos de texto e rótulos fazem parte da GUI do Netscape Navigator. Eles permitem que você interaja com o programa Navigator. Neste capítulo e no próximo, demonstraremos muitos componentes GUI que permitem aos usuários interagir com os seus programas.

As GUIs são construídas a partir de *componentes GUI* (às vezes chamados de *controles* ou *widglets* – a notação abreviada para *window gadgets*). O componente GUI é um objeto com o qual o usuário interage através do mouse, do teclado ou outra forma de entrada, como o reconhecimento de voz. Vários componentes GUI comuns são listados na Fig. 12.2. Nas seções seguintes, discutiremos cada um desses componentes GUI em detalhes. No próximo capítulo, discutiremos os componentes GUI mais avançados.

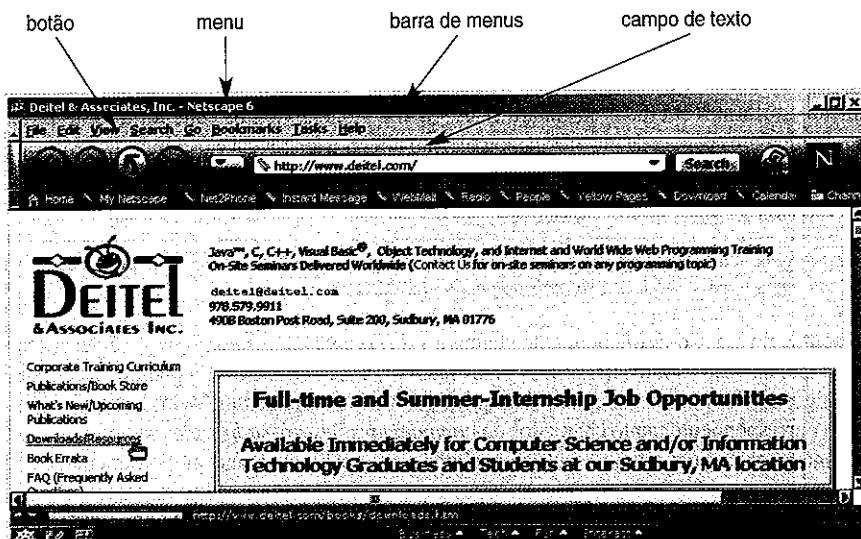


Fig. 12.1 Uma janela de exemplo do Netscape Navigator com componentes GUI.

Componente	Descrição
JLabel	Área em que podem ser exibidos texto não-editável ou ícones.
JTextField	Área em que o usuário insere dados pelo teclado. A área também pode exibir informações.
JButton	Área que aciona um evento quando você clica.
JCheckBox	Componente GUI que tem dois estados: selecionado ou não-selecionado.
JComboBox	Lista escamoteável de itens a partir da qual o usuário pode fazer uma seleção clicando em um item na lista ou digitando na caixa, se permitido.
JList	Área em que uma lista de itens é exibida, a partir da qual o usuário pode fazer uma seleção clicando uma vez em qualquer elemento na lista. Um clique duplo em um elemento na lista gera um evento de ação. Múltiplos elementos podem ser selecionados.
JPanel	Contêiner em que os componentes podem ser colocados.

Fig. 12.2 Alguns componentes GUI básicos.

12.2 Visão geral do Swing

As classes que criam os componentes GUI da Fig. 12.2 fazem parte dos *componentes GUI Swing* do pacote `javax.swing`. Esses componentes GUI se tornaram padrão em Java a partir da versão 1.2 da plataforma Java 2. A maior parte dos *componentes Swing* (como são comumente denominados) são escritos, manipulados e exibidos completamente em Java (sendo conhecidos como *componentes Java puros*).

Os componentes GUI originais do pacote *Abstract Windowing Toolkit* `java.awt` (também chamado de AWT) estão diretamente associados com os recursos da interface gráfica com o usuário da plataforma local. Quando um programa Java com uma GUI AWT é executado em diferentes plataformas Java, os componentes GUI do programa são exibidos com uma aparência diferente em cada plataforma. Considere um programa que exibe um objeto do tipo `Button` (pacote `java.awt`). Em um computador com o sistema operacional Microsoft Windows, o `Button` terá a mesma aparência e comportamento dos botões de outras aplicações Windows. De modo similar, em um computador com o sistema operacional Apple Macintosh, o `Button` terá a mesma aparência e comportamento que os

de outras aplicações Macintosh. Além das diferenças na aparência, algumas vezes a maneira como o usuário interage com os componentes GUI de uma plataforma particular difere de uma plataforma para outra.

Juntos, o aspecto e a maneira como o usuário interage com o programa são conhecidos como *aparência e comportamento* desse programa. Os componentes do Swing permitem que o programador especifique uma aparência e um comportamento uniformes em todas as plataformas. Além disso, o Swing permite fornecer uma aparência e um comportamento particulares para cada plataforma, ou mesmo alterar a aparência e o comportamento enquanto o programa está sendo executado.



Observação de aparência e comportamento 12.2

Como os componentes Swing são escritos em Java, eles oferecem um maior nível de portabilidade e flexibilidade que os componentes GUI Java originais do pacote `java.awt`.

Os componentes Swing são freqüentemente chamados de *componentes peso-leve* – são escritos inteiramente em Java de modo que não são “sobrecarregados” pelos recursos GUI complexos da plataforma em que são utilizados. Os componentes AWT (muitos dos quais são equivalentes aos componentes Swing) que são vinculados à plataforma local são correspondentemente chamados de *componentes peso-pesado* – eles se apóiam no *sistema de janelas* da plataforma local para determinar sua funcionalidade, sua aparência e seu comportamento. Cada componente peso-pesado tem um “peer” (do pacote `java.awt.peer`) que é responsável pelas interações entre o componente e a plataforma local que exibe e manipula o componente. Muitos componentes Swing ainda são componentes peso-pesado. Em particular, as subclasses de `java.awt.Window` (como a `JFrame` utilizada em vários capítulos anteriores) que exibem janelas na tela e subclasses de `java.applet.Applet` (como `JApplet`) ainda exigem interação direta com o sistema de janelas local. Como tal, os componentes GUI Swing peso-pesado são menos flexíveis que muitos dos componentes peso-leve que demonstraremos.



Dica de portabilidade 12.1

A aparência de uma GUI definida com componentes GUI peso-pesado do pacote `java.awt` pode variar de uma plataforma para outra. Componentes peso-pesado “aderem” à plataforma GUI “local”, o que varia de plataforma para plataforma.

A Fig. 12.3 mostra uma hierarquia de herança das classes que definem atributos e comportamentos que são comuns para a maioria dos componentes Swing. Cada classe é exibida com o nome de seu pacote e nome da classe completamente qualificado. Grande parte da funcionalidade de cada componente GUI deriva-se dessas classes. A classe que herda da classe **Component** é *um componente*. Por exemplo, a classe **Container** herda da classe **Component** e a classe **Component** herda de **Object**. Portanto, o **Container** é *um Component* e é *um Object* e um **Component** é *um Object*. A classe que herda da classe **Container** é *um Container*. Portanto, o **JComponent** é *um Container*.



Observação de engenharia de software 12.1

*Para utilizar efetivamente os componentes GUI, as hierarquias de herança do `javax.swing` e do `java.awt` devem ser compreendidas – especialmente a classe **Component**, a classe **Container** e a classe **JComponent**, que definem os recursos comuns à maioria dos componentes Swing.*

A classe **Component** define atributos e comportamentos comuns de todas as subclasses de **Component**. Com poucas exceções, a maioria dos componentes GUI estende a classe **Component** direta ou indiretamente. Um método que se origina na classe **Component** e que tem sido utilizado com freqüência até este ponto do texto é **paint**. Outros métodos anteriormente discutidos que se originam na **Component** são **repaint** e **update**. É importante entender os métodos de classe **Component**, porque muitas das funcionalidades herdadas por cada subclasse de **Component** é definida originalmente pela classe **Component**. As operações comuns para a maioria dos componentes GUI (tanto Swing como AWT) estão localizadas na classe **Component**.



Boa prática de programação 12.1

*Estude os métodos da classe **Component** na documentação on-line do Java 2 SDK para aprender os recursos comuns da maioria dos componentes GUI.*

O **Container** é uma coleção de componentes relacionados. Em aplicativos com **JFrames** e em *applets*, anexamos os componentes ao painel de conteúdo, que é um objeto da classe **Container**. A classe **Container** define os atributos e comportamentos comuns para todas as subclasses de **Container**. Um método que se origina na classe **Container** é **add**, para adicionar componentes a um **Container**. Outro método que se origina na classe **Container** é **setLayout**, que permite que o programa especifique o gerenciador de layout que ajuda um **Container** a posicionar e dimensionar seus componentes.



Boa prática de programação 12.2

Estude os métodos da classe Container na documentação on-line do Java 2 SDK para aprender os recursos comuns de cada contêiner para os componentes GUI.

A classe **JComponent** é a superclasse para a maioria dos componentes Swing. Essa classe define os atributos e comportamentos comuns para todas as subclasses de **JComponent**.

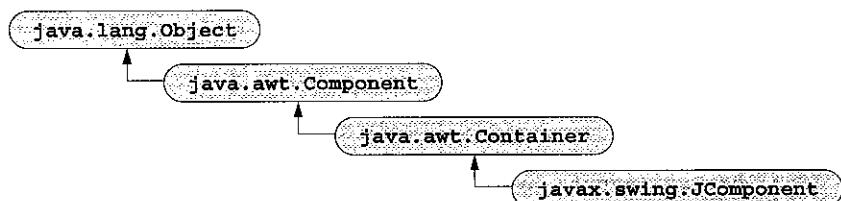


Fig. 12.3 Superclasses comuns de muitos dos componentes Swing.



Boa prática de programação 12.3

Estude os métodos da classe JComponent na documentação on-line do Java 2 SDK para aprender os recursos comuns de cada contêiner para os componentes GUI.

Os componentes Swing que derivam da subclasse **JComponent** têm muitos recursos, incluindo:

1. Uma *aparência* e um *comportamento plugáveis*, que podem ser utilizados para personalizar a aparência e o comportamento quando o programa é executado em plataformas diferentes.
2. Teclas de atalho (chamadas de *mneômicas*) para acesso direto a componentes GUI pelo teclado.
3. Capacidades comuns de tratamento de eventos para os casos em que vários componentes GUI iniciam as mesmas ações em um programa.
4. Breves descrições da finalidade de um componente GUI (chamadas de *dicas de ferramenta*) que são exibidas quando o cursor do mouse é posicionado sobre o componente por um breve instante.
5. Suporte a tecnologias de auxílio ao deficiente físico, como leitores de tela em braile para pessoas cegas.
6. Suporte para *localização* da interface com o usuário – personalizando a interface com o usuário para exibir em diferentes idiomas e convenções culturais.

Esses são apenas alguns dos muitos recursos dos componentes Swing. Discutimos vários desses recursos aqui e no Capítulo 13.

12.3 JLabel

Os *rótulos* fornecem instruções de texto ou informações em uma GUI. Os rótulos são definidos com a classe **JLabel** – uma subclasse de **JComponent**. O rótulo exibe uma única linha de *texto somente de leitura*, uma imagem

ou tanto texto quanto imagem. Os programas raramente alteram um conteúdo do rótulo depois de criá-lo. O aplicativo da Fig. 12.4 demonstra vários recursos de `JLabel`.

```

1 // Fig. 12.4: LabelTest.java
2 // Demonstrando a classe JLabel.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class LabelTest extends JFrame {
12     private JLabel label1, label2, label3;
13
14     // configura a GUI
15     public LabelTest()
16     {
17         super( "Testing JLabel" );
18
19         // obtém painel de conteúdo e configura seu leiaute
20         Container container = getContentPane();
21         container.setLayout( new FlowLayout() );
22
23         // construtor JLabel com um argumento string
24         label1 = new JLabel( "Label with text" );
25         label1.setToolTipText( "This is label1" );
26         container.add( label1 );
27
28         // construtor JLabel com argumentos
29         // string, Icon e alinhamento
30         Icon bug = new ImageIcon( "bug1.gif" );
31         label2 = new JLabel( "Label with text and icon",
32             bug, SwingConstants.LEFT );
33         label2.setToolTipText( "This is label2" );
34         container.add( label2 );
35
36         // construtor JLabel sem argumentos
37         label3 = new JLabel();
38         label3.setText( "Label with icon and text at bottom" );
39         label3.setIcon( bug );
40         label3.setHorizontalTextPosition( SwingConstants.CENTER );
41         label3.setVerticalTextPosition( SwingConstants.BOTTOM );
42         label3.setToolTipText( "This is label3" );
43         container.add( label3 );
44
45         setSize( 275, 170 );
46         setVisible( true );
47     }
48
49     // executa o aplicativo
50     public static void main( String args[] )
51     {
52         LabelTest application = new LabelTest();
53
54         application.setDefaultCloseOperation(
55             JFrame.EXIT_ON_CLOSE );
56     }

```

Fig. 12.4 Demonstrando a classe `JLabel` (parte 1 de 2).

```
57
58 } // fim da classe JLabelTest
```

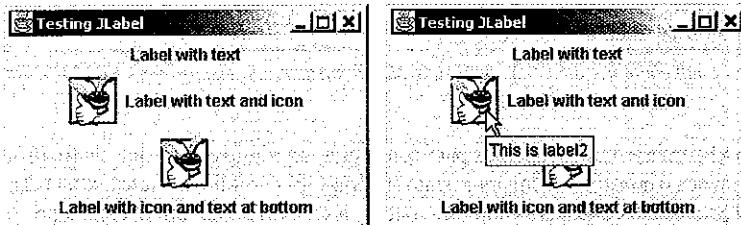


Fig. 12.4 Demonstrando a classe `JLabel` (parte 2 de 2).



Boa prática de programação 12.4

Estude os métodos da classe `javax.swing.JLabel` na documentação on-line do Java 2 SDK para aprender todos os recursos da classe antes de utilizá-la.

O programa declara três referências `JLabel` na linha 12. Os objetos `JLabel` são instanciados no construtor (linhas 15 a 47). A linha 24 cria um objeto `JLabel` com o texto "**Label with text**". O rótulo exibe este texto quando ele aparece na tela, isto é, quando a janela é exibida neste programa.

A linha 25 utiliza o método `setToolTipText` (herdado da classe `JComponent` pela classe `JLabel`) para especificar a dica de ferramenta (veja a segunda captura de tela na Fig. 12.4) que é exibida automaticamente quando o usuário posiciona o cursor do mouse sobre o rótulo na GUI. Ao executar esse programa, tente posicionar o mouse sobre cada rótulo para ver sua dica de ferramenta. A linha 26 adiciona `label1` ao painel de conteúdo.



Observação de aparência e comportamento 12.3

Utilize as dicas de ferramentas (configuradas com o método `setToolTipText` de `JComponent`) para adicionar texto descritivo aos seus componentes GUI. Esse texto ajuda o usuário a determinar a finalidade do componente GUI na interface com o usuário.

Muitos componentes Swing podem exibir imagens especificando um `Icon` como argumento para seu construtor ou utilizando um método que normalmente é chamado de `setIcon`. O `Icon` é um objeto de qualquer classe que implementa a interface `Icon` (pacote `javax.swing`). Uma dessas classes é `ImageIcon` (pacote `javax.swing`), que suporta vários formatos de imagem, incluindo *Graphics Interchange Format (GIF)*, *Portable Network Graphics (PNG)* e *Joint Photographic Experts Group (JPEG)*. Os nomes de arquivo para cada um desses tipos em geral acabam com `.gif`, `.png` ou `.jpg` (ou `.jpeg`), respectivamente. Discutimos imagens em mais detalhes no Capítulo 18. A linha 30 define um objeto `ImageIcon`. O arquivo `bug1.gif` contém a imagem para carregar e armazenar no objeto `ImageIcon`. Pressupõe-se que esse arquivo esteja no mesmo diretório que o programa (discutiremos a localização do arquivo em outra parte no Capítulo 18). O objeto `ImageIcon` é atribuído à referência `Icon bug`. Lembre-se, a classe `ImageIcon` implementa a interface `Icon`, portanto um `ImageIcon` é um `Icon`.

A classe `JLabel` suporta a exibição de `Icons`. As linhas 31 e 32 utilizam outro construtor `JLabel` para criar um rótulo que exibe o texto "**Label with text and icon**" e o `Icon` ao qual `bug` faz referência, e está *justificado à esquerda*, ou *alinhado à esquerda* (isto é, o ícone e o texto estão no lado esquerdo da área do rótulo na tela). A interface `SwingConstants` (pacote `javax.swing`) define um conjunto de constantes inteiras comuns (como `SwingConstants.LEFT`) que são utilizadas com muitos componentes Swing. Por *default*, o texto aparece à direita da imagem quando um rótulo contém texto e imagem. Os alinhamentos horizontal e vertical de um rótulo podem ser configurados com os métodos `setHorizontalAlignment` e `setVerticalAlignment`, respectivamente. A linha 33 especifica o texto da dica de ferramenta para `label2`. A linha 34 adiciona `label2` ao painel de conteúdo.



Erro comum de programação 12.1

Se você não adicionar explicitamente um componente GUI a um contêiner, o componente GUI não será exibido quando o contêiner aparecer na tela.



Erro comum de programação 12.2

Adicionar a um contêiner um componente que não foi instanciado dispara uma NullPointerException.

A classe **JLabel** fornece muitos métodos para configurar um rótulo depois que ele foi instanciado. A linha 37 cria um **JLabel** e invoca o construtor sem argumento (padrão). Esse rótulo não tem texto ou **Icon**. A linha 38 utiliza o método **setText** de **JLabel** para configurar o texto a ser exibido no rótulo. O método correspondente **getText** recupera o texto atual exibido em um rótulo. A linha 39 utiliza o método **setIcon** de **JLabel** para configurar o **Icon** exibido no rótulo. O método correspondente **getIcon** recupera o **Icon** atual exibido em um rótulo. As linhas 40 e 41 utilizam os métodos **setHorizontalTextPosition** e **setVerticalTextPosition** de **JLabel** para especificar a posição do texto no rótulo. Neste caso, o texto será centralizado horizontalmente e aparecerá na parte inferior do rótulo. Portanto, o **Icon** aparecerá acima do texto. A linha 42 configura o texto de dica de ferramenta para o **label3**. A linha 43 adiciona **label3** ao painel de conteúdo.

12.4 Modelo de tratamento de eventos

Na seção precedente, não discutimos o tratamento de eventos porque não há eventos específicos para os objetos **JLabel**. As GUIs são *baseadas em eventos* (isto é, geram *eventos* quando o usuário do programa interage com a GUI). Algumas interações comuns são mover o mouse, clicar no mouse, clicar em um botão, digitar um campo de texto, selecionar o item de um menu, fechar uma janela, etc. Quando ocorre uma interação com o usuário, um evento é enviado para o programa. Informações de eventos GUI são armazenadas em um objeto de uma classe que estende **AWTEvent**. A Fig. 12.5 ilustra uma hierarquia que contém muitas das classes de eventos que utilizamos do pacote **java.awt.event**. Muitas dessas classes de eventos serão discutidas por todo este capítulo e no Capítulo 13. Os tipos de evento do pacote **java.awt.event** são usados tanto em componentes AWT como em componentes Swing. Também foram adicionados outros tipos de evento que são específicos para vários tipos de componentes Swing. Novos tipos de evento de componente do Swing são definidos no pacote **javax.swing.event**.

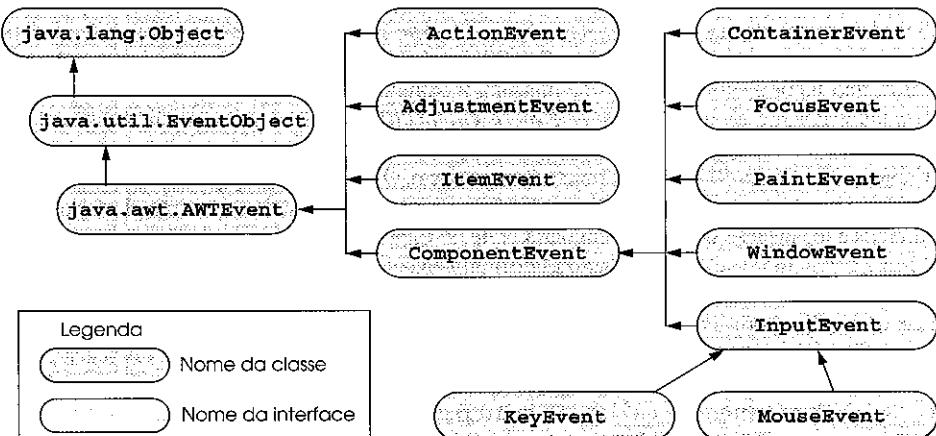


Fig. 12.5 Algumas classes de eventos do pacote **java.awt.event**.

O mecanismo de tratamento de eventos tem três partes – a *origem do evento*, o *objeto evento* e o “*ouvinte*” (*listener*) do evento. A origem do evento é o componente GUI particular com o qual o usuário interage. O objeto evento encapsula as informações sobre o evento que ocorreu. Estas informações incluem uma referência para a origem do evento e quaisquer informações específicas para o evento que possam ser necessárias para que o ouvinte do evento trate do evento. O ouvinte do evento é um objeto que é notificado pela origem do evento quando ocorre um evento. O do evento recebe um objeto evento quando é notificado do evento e, então, usa o objeto para responder ao evento. A origem do evento deve fornecer métodos que permitam que os ouvintes sejam registrados e tenham o registro cancelado. A origem do evento também precisa manter uma lista de seus ouvintes registrados e ser capaz de notificar seus ouvintes quando ocorre um evento.

O programador precisa executar duas tarefas fundamentais para processar um evento da interface gráfica com o usuário em um programa – registrar um *ouvinte de evento* para o componente GUI que se espera que vá gerar o evento e implementar um *método de tratamento de evento* (ou conjunto de métodos de tratamento de eventos). Normalmente, os métodos que tratam eventos são chamados de *tratadores de evento*. O ouvinte de eventos para um evento GUI é um objeto de uma classe que implementa uma ou mais das interfaces *listeners* de eventos dos pacotes `java.awt.event` e `javax.swing.event`. Muitos dos tipos de ouvintes de eventos são comuns aos componentes Swing e AWT. Esses tipos são definidos no pacote `java.awt.event` e muitos desses são mostrados na Fig. 12.6. Tipos adicionais de ouvintes de eventos que são específicos para componentes Swing são definidos no pacote `javax.swing.event`.

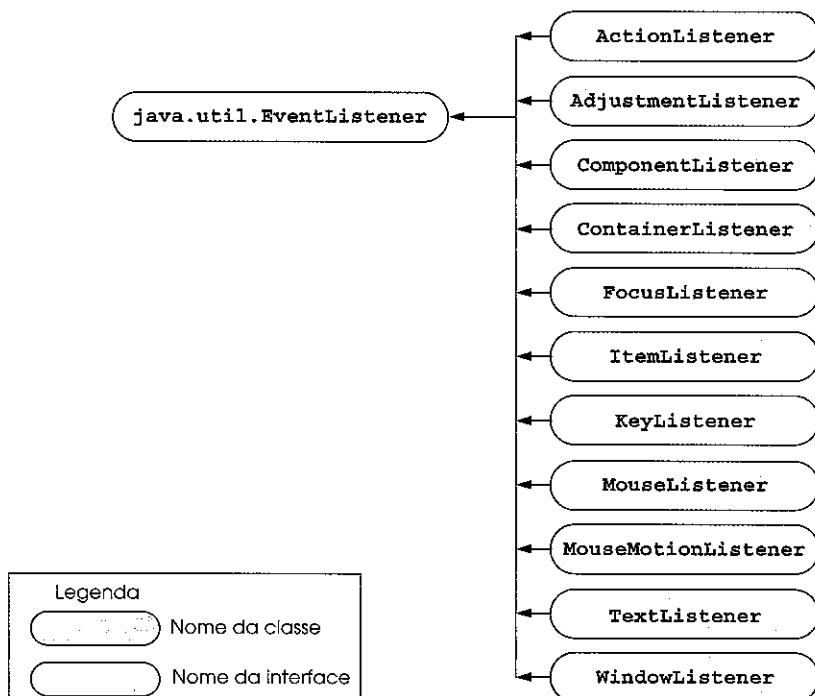


Fig. 12.6 Interfaces *listeners* de eventos do pacote `java.awt.event`.

O objeto *listener* de eventos “espera por” tipos específicos de eventos gerados por origens de eventos (normalmente componentes GUI) em um programa. O tratador de eventos é um método que é chamado em resposta a um tipo de evento em particular. Cada interface *listener* de eventos especifica um ou mais métodos de tratamento de evento que *devem* ser definidos na classe que implementa a interface *listener* de eventos. Lembre-se de que as interfaces definem métodos **abstract**. Qualquer classe que implementa uma interface deve definir todos os métodos dessa interface; caso contrário, a classe é uma classe **abstract** e não pode ser utilizada para criar objetos. O uso

de ouvintes de eventos em tratamento de eventos é conhecido como *modelo de delegação de evento* – o processamento de um evento é delegado a um objeto particular (o ouvinte) no programa.

Quando ocorre um evento, o componente GUI com o qual o usuário interagiu notifica seus ouvintes registrados chamando o método de tratamento de evento apropriado de cada ouvinte. Por exemplo, quando o usuário pressiona a tecla *Enter* em um **JTextField**, o método **actionPerformed** do ouvinte registrado é chamado. Como o tratador de eventos foi registrado? Como o componente GUI sabe que deve chamar **actionPerformed** em vez de algum outro método de tratamento de evento? Respondemos a essas perguntas e apresentamos o diagrama da interação como parte do próximo exemplo.

12.5 JTextField e JPasswordField

JTextFields e **JPasswordFields** (pacote `javax.swing`) são áreas de uma única linha em que o texto pode ser inserido pelo usuário pelo teclado ou o texto pode ser simplesmente exibido. O **JPasswordField** mostra que um caractere foi digitado quando o usuário insere os caracteres, mas oculta os caracteres assumindo que eles representam uma senha que deve permanecer conhecida apenas para o usuário. Quando o usuário digita os dados em um **JTextField** ou **JPasswordField** e pressiona a tecla *Enter*, ocorre um evento de ação. Se o programa registra um ouvinte de evento, o ouvinte processa o evento e pode usar os dados que estão no **JTextField** ou **JPasswordField** no momento em que o evento ocorreu no programa. A classe **JTextField** estende a classe **JTextComponent** (pacote `javax.swing.text`), que fornece muitos recursos comuns para os componentes baseados em texto do Swing. A classe **JPasswordField** estende **JTextField** e adiciona vários métodos que são específicos para o processamento de senhas.



Erro comum de programação 12.3

Utilizar uma letra *f* minúscula nos nomes das classes **JTextField** ou **JPasswordField** é um erro de sintaxe.

O aplicativo da Fig. 12.7 utiliza as classes **JTextField** e **JPasswordField** para criar e manipular quatro campos. Quando o usuário pressiona *Enter* no campo atualmente ativo (o componente atualmente ativo “está em *foco*”), é exibida uma caixa de diálogo de mensagem que contém o texto no campo. Quando ocorre um evento no **JPasswordField**, a senha é revelada.

```

1 // Fig. 12.7: TextFieldTest.java
2 // Demonstrando a classe JTextField.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class TextFieldTest extends JFrame {
12     private JTextField textField1, textField2, textField3;
13     private JPasswordField passwordField;
14
15     // configura a GUI
16     public TextFieldTest()
17     {
18         super( "Testing JTextField and JPasswordField" );
19
20         Container container = getContentPane();
21         container.setLayout( new FlowLayout() );
22
23         // constrói campo de texto com dimensões default
24         textField1 = new JTextField( 10 );

```

Fig. 12.7 Demonstrando **JTextFields** e **JPasswordFields** (parte 1 de 3).

```

25     container.add( textField1 );
26
27     // constrói campo de texto com texto default
28     textField2 = new JTextField( "Enter text here" );
29     container.add( textField2 );
30
31     // constrói campo de texto com texto default e 20
32     // elementos visíveis e sem tratador de eventos
33     textField3 = new JTextField( "Uneditable text field", 20 );
34     textField3.setEditable( false );
35     container.add( textField3 );
36
37     / constrói campo de texto com texto default
38     passwordField = new JPasswordField( "Hidden text" );
39     container.add( passwordField );
40
41     // registra tratadores de evento
42     TextFieldHandler handler = new TextFieldHandler();
43     textField1.addActionListener( handler );
44     textField2.addActionListener( handler );
45     textField3.addActionListener( handler );
46     passwordField.addActionListener( handler );
47
48     setSize( 325, 100 );
49     setVisible( true );
50 }
51
52 // executa o aplicativo
53 public static void main( String args[] )
54 {
55     TextFieldTest application = new TextFieldTest();
56
57     application.setDefaultCloseOperation(
58         JFrame.EXIT_ON_CLOSE );
59 }
60
61 // classe interna privativa para tratamento de eventos
62 private class TextFieldHandler implements ActionListener {
63
64     // processa eventos de campos de texto
65     public void actionPerformed( ActionEvent event )
66     {
67         String string = "";
68
69         // usuário pressionou Enter no JTextField textField1
70         if ( event.getSource() == textField1 )
71             string = "textField1: " + event.getActionCommand();
72
73         // usuário pressionou Enter no JTextField textField2
74         else if ( event.getSource() == textField2 )
75             string = "textField2: " + event.getActionCommand();
76
77         // usuário pressionou Enter no JTextField textField3
78         else if ( event.getSource() == textField3 )
79             string = "textField3: " + event.getActionCommand();
80
81         // usuário pressionou Enter no JPasswordField passwordField
82         else if ( event.getSource() == passwordField ) {
83             JPasswordField pwd =
84                 ( JPasswordField ) event.getSource();

```

Fig. 12.7 Demonstrando JTextFields e JPasswordFields (parte 2 de 3).

```

85         string = "PasswordField: " +
86             new String( passwordField.getPassword() );
87     }
88
89     JOptionPane.showMessageDialog( null, string );
90 }
91
92 } // fim da classe interna privativa TextFieldHandler
93
94 } // fim da classe TextFieldTest

```

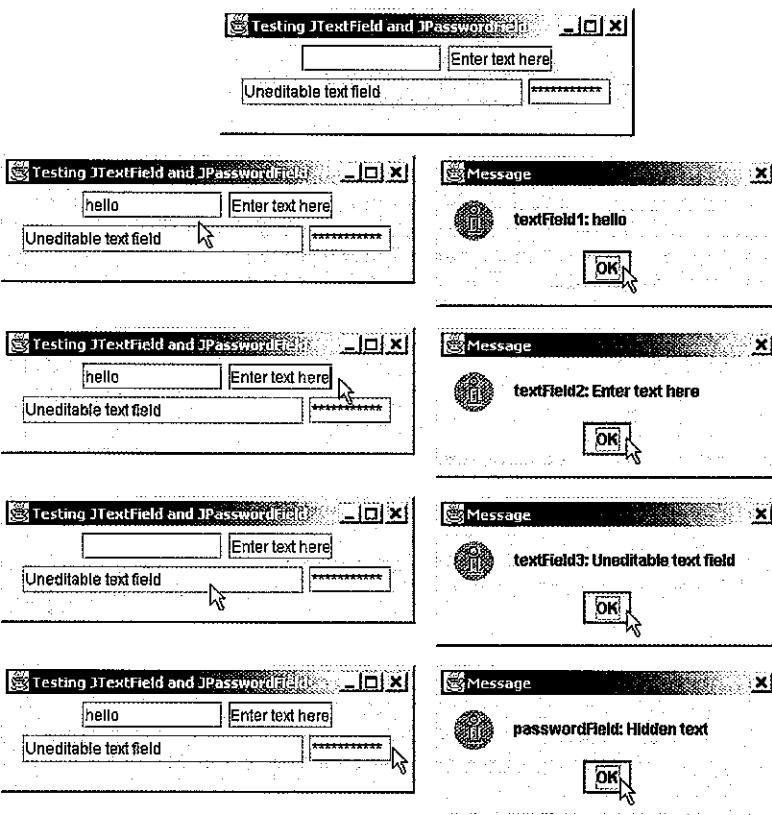


Fig. 12.7 Demonstrando JTextFields e JPasswordFields (parte 3 de 3).

As linhas 12 e 13 declaram três referências a **JTextFields** (**textField1**, **textField2** e **textField3**) e um **JPasswordField** (**passwordField**). Cada uma delas é instanciada no construtor (linhas 16 a 50). A linha 24 define o **JTextField** **textField1** com 10 colunas de texto. A largura do campo de texto será a largura em *pixels* do caractere médio na fonte atual do campo de texto multiplicada por 10. A linha 25 adiciona **textField1** ao painel de conteúdo.

A linha 28 define **textField2** com o texto inicial "Enter text here" para ser exibido no campo de texto. A largura do campo de texto é determinada pelo texto. A linha 29 adiciona **textField2** ao painel de conteúdo.

A linha 33 define **textField3** e chama o construtor **JTextField** com dois argumentos – o texto *default* "Uneditable text field" para ser exibido no campo de texto e o número de colunas (20). A largura do campo de texto é determinada pelo número de colunas especificado. A linha 34 utiliza o método **setEditable** (herdado pelo **JTextField** da classe **JTextComponent**) para indicar que o usuário não pode modificar o texto no campo de texto. A linha 35 adiciona **textField3** ao painel de conteúdo.

A linha 38 define o `JPasswordField passwordField` com o texto "`Hidden text`" para ser exibido no campo de texto. A largura do campo de texto é determinada pelo texto. Repare que o texto é exibido como *string* de asteriscos quando o programa é executado. A linha 39 adiciona `passwordField` ao painel de conteúdo.

Para o tratamento de eventos nesse exemplo, definimos a classe interna `TextFieldHandler` (linhas 62 a 92), que implementa a interface `ActionListener` (a classe `TextFieldHandler` é discutida em breve). Portanto, cada instância da classe `TextFieldHandler` é um `ActionListener`. A linha 42 define uma instância da classe `TextFieldHandler` e a atribui à referência `handler`. Essa mesma instância será utilizada como o objeto *listener* de eventos para os `JTextFields` e o `JPasswordField` nesse exemplo.

As linhas 43 a 46 são as instruções de registro de evento que especificam o objeto *listener* de eventos para cada um dos três `JTextFields` e para o `JPasswordField`. Depois que essas instruções são executadas, o objeto ao qual `handler` faz referência está *esperando eventos* (isto é, será notificado quando ocorrer um evento) desses quatro objetos. O programa chama o método `addActionListener` da classe `JTextField` para registrar o evento para cada componente. O método `addActionListener` recebe como argumento um objeto `ActionListener`. Portanto, qualquer objeto de uma classe que implemente a interface `ActionListener` (isto é, qualquer objeto que é um `ActionListener`) pode ser fornecido como argumento para esse método. O objeto ao qual `handler` faz referência é um `ActionListener` porque sua classe implementa a interface `ActionListener`. Agora, quando o usuário pressiona *Enter* em qualquer desses quatro campos, o método `actionPerformed` (linhas 65 a 90) na classe `TextFieldHandler` é chamado para tratar o evento.



Observação de engenharia de software 12.2

O “ouvinte” de evento para um evento deve implementar a sua interface apropriada.

O método `actionPerformed` utiliza o método `getSource` de seu argumento `ActionEvent` para determinar o componente GUI com o qual o usuário interagiu e cria um `String` para exibir em uma caixa de diálogo de mensagem. O método `getActionCommand` de `ActionEvent` devolve o texto no `JTextField` que gerou o evento. Se o usuário interagiu com o `JPasswordField`, as linhas 83 e 84 convertem a referência `Component` devolvida por `event.getSource()` para uma referência `JPasswordField`, de modo que as linhas 85 e 86 possam utilizar o método `getPassword` de `JPasswordField` para obter a senha e criar o `String` a ser exibido. O método `getPassword` devolve a senha como um *array* do tipo `char` que é utilizado como argumento para um construtor de `String` para criar um `String`. A linha 89 exibe uma caixa de mensagem que indica o nome da referência ao componente GUI e o texto que o usuário digitou no campo.

Observe que mesmo um `JTextField` não-editável pode gerar um evento. Simplesmente clique no campo de texto, depois tecle *Enter*. Também observe que o texto real da senha é exibido quando se pressiona *Enter* no `JPasswordField` (é claro que normalmente você não faria isso!)



Erro comum de programação 12.4

Esquecer de registrar um objeto tratador de evento para um tipo de evento particular de um componente GUI faz com que nenhum evento daquele tipo seja tratado para aquele componente.

Utilizar uma classe separada para definir um ouvinte de eventos é uma prática de programação comum para separar a interface GUI da implementação de seu tratador de evento. No restante deste capítulo e no Capítulo 13, muitos programas utilizam classes *listeners* de eventos separadas para processar eventos GUI.

12.5.1 Como funciona o tratamento de eventos

Vamos ilustrar como o mecanismo de tratamento de eventos funciona utilizando o `JTextField textField1` do exemplo precedente. Restam duas perguntas da Seção 12.4 em aberto:

1. Como o tratador de eventos foi registrado?
2. Como o componente GUI sabe que deve chamar `actionPerformed` em vez de algum outro método de tratamento de evento?

A primeira pergunta é respondida pelo registro de evento realizado nas linhas 43 a 46 do programa. A Fig. 12.8 é um diagrama que mostra a referência `JTextField textField1`, o objeto `JTextField` ao qual ela se refere e o objeto `listener` que está registrado para tratar o evento do `JTextField`.

Cada `JComponent` tem um objeto da classe `EventListenerList` (pacote `javax.swing.event`) chamado de `listenerList` como variável de instância. Todos os `listeners` registrados são armazenados na `listenerList` (representada como *array* na Fig. 12.8). Quando a instrução

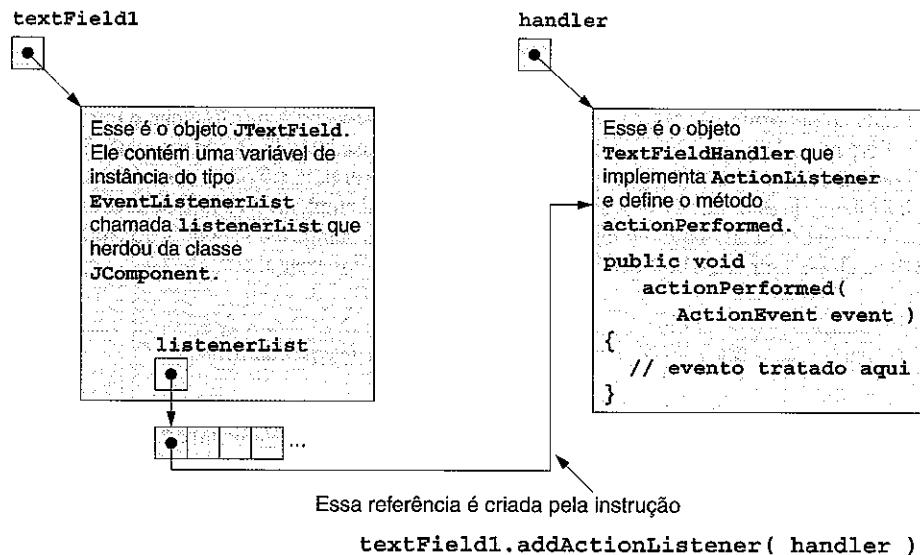


Fig. 12.8 Registro de evento para `textField1 JTextField`.

```
textField1.addActionListener( handler );
```

é executada na Fig. 12.7, uma nova entrada é colocada na `listenerList` para `textField1 JTextField` indicando tanto a referência ao objeto `listener` como o tipo de `listener` (nesse caso `ActionListener`).

O tipo é importante na resposta à segunda pergunta – como o componente GUI sabe que deve chamar `actionPerformed` e não algum outro método de tratamento de evento? Cada `JComponent`, na verdade, suporta vários tipos diferentes de evento, incluindo *eventos de mouse*, *eventos de teclas* e outros. Quando ocorre um evento, o evento é *despachado* apenas para os ouvintes de eventos do tipo apropriado. O despacho de um evento é simplesmente uma chamada ao método de tratamento de eventos para cada ouvinte registrado para esse tipo de evento.

Cada tipo de evento tem uma interface `listener` de eventos correspondente. Por exemplo, os `ActionEvents` são tratados por `ActionListeners`, `MouseEvents` são tratados por `MouseListeners` (e `MouseMotionListeners` como veremos) e `KeyEvents` são tratados por `KeyListeners`. Quando um evento é gerado por uma interação de usuário com um componente, o componente recebe um *ID (identificador)* de evento exclusivo, que especifica o tipo de evento que ocorreu. O componente GUI utiliza o ID de evento para decidir o tipo de ouvinte para o qual o evento deve ser despachado e o método a chamar. No caso de um `ActionEvent`, o evento é despachado para cada método `actionPerformed` de `ActionListener` registrado (o único método na interface `ActionListener`). No caso de um `MouseEvent`, o evento é despachado para cada `MouseListener` registrado (ou `MouseMotionListener`, dependendo do evento que ocorre). O ID de evento do `MouseEvent` determina qual dos sete métodos diferentes de tratamento de eventos de mouse é chamado. A lógica de toda essa decisão é tratada para você pelos componentes GUI. Discutiremos outros tipos de evento e interfaces `listeners` de evento à medida que se tornarem necessários para cada novo componente que abordarmos.

12.6 JButton

O botão é um componente em que o usuário clica para disparar uma ação específica. O programa Java pode utilizar vários tipos de botões, incluindo *botões de comando*, *caixas de marcação*, *botões de alternância* e *botões de opção*. A Fig. 12.9 mostra a hierarquia de herança dos botões do Swing que abordamos neste capítulo. Como você pode ver no diagrama, todos os tipos de botão são subclasses de **AbstractButton** (pacote `javax.swing`), que define muitos dos recursos que são comuns aos botões do Swing. Nesta seção, concentrarmo-nos nos botões que são geralmente utilizados para iniciar um comando. Outros tipos de botão são abordados nas próximas seções.

O botão de comando gera um **ActionEvent** quando o usuário clica no botão com o mouse. Os botões de comando são criados com a classe **JButton**, que herda da classe **AbstractButton**. O texto na face de um **JButton** chama-se *rótulo de botão*. A GUI pode ter muitos **JButtons**, mas cada rótulo de botão em geral deve ser único.

Observação de aparência e comportamento 12.4



Ter mais de um JButton com o mesmo rótulo torna os JButton ambíguos para o usuário. Certifique-se de fornecer um rótulo único para cada botão.

O aplicativo da Fig. 12.10 cria dois **JButtons** e demonstra que **JButtons** (como **JLabels**) suportam a exibição de **Icons**. O tratamento de eventos para os botões é realizado por uma única instância da classe interna **ButtonHandler** (linhas 53 a 62).

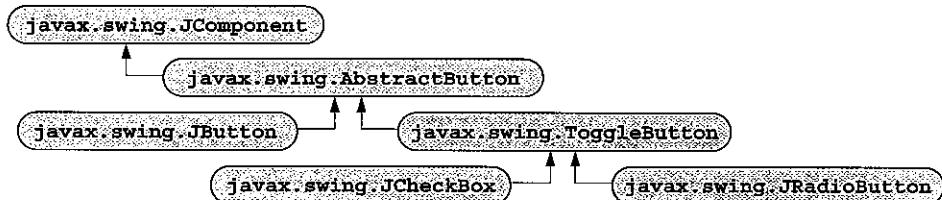


Fig. 12.9 A hierarquia de botões.

```

1 // Fig. 12.10: ButtonTest.java
2 // Criando JButton.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class ButtonTest extends JFrame {
12     private JButton plainButton, fancyButton;
13
14     // configura a GUI
15     public ButtonTest()
16     {
17         super( "Testing Buttons" );
18
19         // obtém painel de conteúdo e configura o layout
20         Container container = getContentPane();
21         container.setLayout( new FlowLayout() );
  
```

Fig. 12.10 Demonstrando botões de comando e eventos de ação (parte 1 de 3).

```

22
23     // cria botões
24     plainButton = new JButton( "Plain Button" );
25     container.add( plainButton );
26
27     Icon bug1 = new ImageIcon( "bug1.gif" );
28     Icon bug2 = new ImageIcon( "bug2.gif" );
29     fancyButton = new JButton( "Fancy Button", bug1 );
30     fancyButton.setRolloverIcon( bug2 );
31     container.add( fancyButton );
32
33     // cria uma instância da classe interna ButtonHandler
34     // para uso no tratamento de eventos de botão
35     ButtonHandler handler = new ButtonHandler();
36     fancyButton.addActionListener( handler );
37     plainButton.addActionListener( handler );
38
39     setSize( 275, 100 );
40     setVisible( true );
41 }
42
43 // executa o aplicativo
44 public static void main( String args[] )
45 {
46     ButtonTest application = new ButtonTest();
47
48     application.setDefaultCloseOperation(
49         JFrame.EXIT_ON_CLOSE );
50 }
51
52 // classe interna para tratamento de evento de botão
53 private class ButtonHandler implements ActionListener {
54
55     // trata evento de botão
56     public void actionPerformed( ActionEvent event )
57     {
58         JOptionPane.showMessageDialog( null,
59             "You pressed: " + event.getActionCommand() );
60     }
61
62 } // fim da classe interna privativa ButtonHandler
63
64 } // fim da classe ButtonTest

```

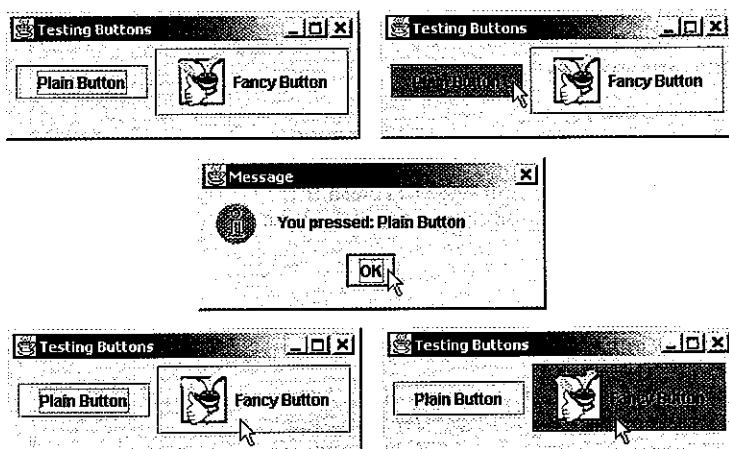


Fig. 12.10 Demonstrando botões de comando e eventos de ação (parte 2 de 3).



Fig. 12.10 Demonstrando botões de comando e eventos de ação (parte 3 de 3).

A linha 12 declara duas referências para as instâncias da classe `JButton` – `plainButton` e `fancyButton` – que são instanciadas no construtor.

A linha 24 cria `plainButton` com o rótulo de botão "Plain Button". A linha 25 adiciona o botão ao painel de conteúdo.

O `JButton` pode exibir `Icons`. Para fornecer ao usuário um nível extra de interatividade visual com a GUI, o `JButton` também pode ter um `Icon rollover` – um `Icon` que é exibido quando o mouse é posicionado sobre o botão. O ícone no botão muda quando o mouse se move para dentro e para fora da área do botão na tela. As linhas 27 e 28 criam dois objetos `ImageIcon` que representam o `Icon default` e o `Icon rollover` para o `JButton` criado na linha 29. Ambas as instruções supõem que os arquivos de imagem estão armazenados no mesmo diretório que o programa (o que, geralmente, é o caso para os aplicativos que utilizam imagens).

A linha 29 cria `fancyButton` com o texto `default` "Fancy Button" e o `Icon bug1`. Por `default`, o texto é exibido à direita do ícone. A linha 30 utiliza o método `setRolloverIcon` (herdado da classe `AbstractButton` pela classe `JButton`) para especificar a imagem exibida no botão quando o usuário posiciona o mouse sobre o botão. A linha 31 adiciona o botão ao painel de conteúdo.

Observação de aparência e de comportamento 12.5



Utilizar os ícones rollover para JButtons fornece feedback visual ao usuário, indicando que, se eles clicam com o mouse, a ação do botão ocorrerá.

`JButtons` (como `JTextFields`) geram `ActionEvents`. Como mencionado antes, o `ActionEvent` pode ser processado por qualquer objeto `ActionListener`. As linhas 35 a 37 registram um objeto `ActionListener` para cada `JButton`. A classe interna `ButtonHandler` (linhas 53 a 62) define `actionPerformed` para exibir uma caixa de diálogo de mensagem que contém antes o rótulo do botão que foi pressionado pelo usuário. O método `getActionCommand` de `ActionEvent` devolve o rótulo do botão que gerou o evento.

12.7 JCheckBox e JRadioButton

Os componentes GUI Swing contêm três tipos de *botões de estado* – `JToggleButton`, `JCheckBox` e `JRadioButton` – que têm valores ativados/desativados ou verdadeiro/falso. `JToggleButtons` são freqüentemente utilizados com *barra de ferramentas* (conjuntos de pequenos botões em geral localizados em uma barra ao longo da parte superior de uma janela) e são abordados no Capítulo 13. As classes `JCheckBox` e `JRadioButton` são subclasses de `JToggleButton`. O `JRadioButton` é diferente do `JCheckBox` porque normalmente existem vários `JRadioButtons` agrupados e apenas um dos `JRadioButtons` do grupo pode estar selecionado (verdadeiro) a qualquer momento. Primeiro discutiremos a classe `JCheckBox`.

Observação de aparência e comportamento 12.6



Como a classe AbstractButton suporta exibição de texto e imagens em um botão, todas as subclasses de AbstractButton também suportam exibição de texto e imagens.

O aplicativo da Fig. 12.11 utiliza dois objetos `JCheckBox` para alterar o estilo da fonte do texto exibido em um `JTextField`. O `JCheckBox` aplica um estilo em negrito quando selecionado e o outro aplica um estilo em itálico quando selecionado. Se ambos forem selecionados, o estilo da fonte é negrito e itálico. Quando a execução do programa inicia, nenhuma `JCheckBox` está marcada (`true`).

```

1 // Fig. 12.11: CheckBoxTest.java
2 // Criando botões Checkbox.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class CheckBoxTest extends JFrame {
12     private JTextField field;
13     private JCheckBox bold, italic;
14
15     // configura a GUI
16     public CheckBoxTest()
17     {
18         super( "JCheckBox Test" );
19
20         // obtém painel de conteúdo e configura o leiaute
21         Container container = getContentPane();
22         container.setLayout( new FlowLayout() );
23
24         // configura o JTextField e configura a fonte
25         field =
26             new JTextField( "Watch the font style change", 20 );
27         field.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
28         container.add( field );
29
30         // cria objetos caixa de marcação
31         bold = new JCheckBox( "Bold" );
32         container.add( bold );
33
34         italic = new JCheckBox( "Italic" );
35         container.add( italic );
36
37         // registra ouvintes para JCheckboxes
38         CheckBoxHandler handler = new CheckBoxHandler();
39         bold.addItemListener( handler );
40         italic.addItemListener( handler );
41
42         setSize( 275, 100 );
43         setVisible( true );
44     }
45
46     // executa o aplicativo
47     public static void main( String args[] )
48     {
49         CheckBoxTest application = new CheckBoxTest();
50
51         application.setDefaultCloseOperation(
52             JFrame.EXIT_ON_CLOSE );
53     }
54
55     // classe interna privativa para tratamento de eventos de ItemListener
56     private class CheckBoxHandler implements ItemListener {
57         private int valBold = Font.PLAIN;
58         private int valItalic = Font.PLAIN;
59     }

```

Fig. 12.11 Programa que cria dois botões JCheckBox (parte 1 de 2).

```

60      // responde a eventos de caixas de marcação
61      public void itemStateChanged( ItemEvent event )
62      {
63          // processa eventos da caixa de marcação Bold
64          if ( event.getSource() == bold )
65
66              if ( event.getStateChange() == ItemEvent.SELECTED )
67                  valBold = Font.BOLD;
68              else
69                  valBold = Font.PLAIN;
70
71          // processa eventos da caixa de marcação Italic
72          if ( event.getSource() == italic )
73
74              if ( event.getStateChange() == ItemEvent.SELECTED )
75                  valItalic = Font.ITALIC;
76              else
77                  valItalic = Font.PLAIN;
78
79          // configura a fonte do campo de texto
80          field.setFont(
81              new Font( "Serif", valBold + valItalic, 14 ) );
82      }
83
84  } // fim da classe interna privativa CheckBoxHandler
85
86 } // fim da classe CheckBoxTest

```

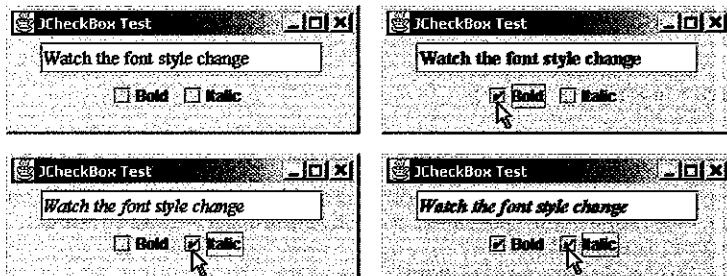


Fig. 12.11 Programa que cria dois botões JCheckBox (parte 2 de 2).

Depois que o `JTextField` é criado e inicializado, a linha 27 configura a fonte do `JTextField` como `Serif`, no estilo `PLAIN` e com o tamanho de 14 pontos. Em seguida, o construtor cria dois objetos `JCheckBox` com as linhas 31 e 34. O `String` passado para o construtor é o *rótulo da caixa de marcação* que aparece à direita do `JCheckBox` por default.

Quando o usuário clica em um `JCheckBox`, é gerado um `ItemEvent` que pode ser tratado por um `ItemListener` (qualquer objeto de uma classe que implementa a interface `ItemListener`). O `ItemListener` deve definir o método `itemStateChanged`. Nesse exemplo, o tratamento de evento é executado por uma instância da classe interna `CheckBoxHandler` (linhas 56 a 84). As linhas 38 a 40 criam uma instância da classe `CheckBoxHandler` e a registram com o método `addItemListener` como o `ItemListener` para as duas `Jcheckboxes`, `bold` e `italic`.

O método `itemStateChanged` (linhas 61 a 82) é chamado quando o usuário clica na `JCheckBox bold` ou na `italic`. O método utiliza `event.getSource()` para determinar em qual `JCheckBox` se clicou. Se foi a `JCheckBox bold`, a estrutura `if/else` nas linhas 66 a 69 utiliza o método `getStateChange` de `ItemEvent` para determinar o estado do botão (`ItemEvent.SELECTED` ou `ItemEvent.DESELECTED`). Se o estudo for selecionado, `Font.BOLD` é atribuído ao inteiro `valBold`; caso contrário, `Font.PLAIN` é atribuído a `valBold`. Uma estrutura `if/else` semelhante é executada caso se clique na `JCheckBox italic`. Se o estudo `italic` for selecionado, `Font.ITALIC` é atribuído ao inteiro `valItalic`; caso contrário, `Font.PLAIN` é atribuí-

do a `valItalic`. A soma de `valBold` e `valItalic` é utilizada nas linhas 80 e 81 como o estilo da nova fonte para o `JTextField`.

Os *botões de opção* (definidos na classe `JRadioButton`) são semelhantes às caixas de marcação no sentido de que têm dois estados – *selecionado* e *não-selecionado*. Entretanto, os botões de opção normalmente aparecem como um *grupo* em que apenas um botão de opção pode ser selecionado por vez. Selecionar um botão de opção diferente no grupo automaticamente força todos os outros botões de opção no grupo a passar para o estado *não-selecionado*. Os botões de opção são utilizados para representar um conjunto de opções *mutuamente exclusivas* (isto é, múltiplas opções no grupo não seriam selecionadas ao mesmo tempo). O relacionamento lógico entre os botões de opção é mantido por um objeto `ButtonGroup` (pacote `javax.swing`). O objeto `ButtonGroup` em si não é um componente GUI. Portanto, um objeto `ButtonGroup` não é exibido em uma interface com o usuário. Em seu lugar, os objetos individuais `JRadioButton` do grupo são exibidos na GUI.



Erro comum de programação 12.5

Adicionar um objeto `ButtonGroup` (ou um objeto de qualquer outra classe não-derivada de `Component`) a um contêiner é um erro de sintaxe.

O aplicativo da Fig. 12.12 é semelhante ao programa precedente. O usuário pode alterar o estilo da fonte do texto de um `JTextField`. O programa utiliza botões de opção que permitem que apenas um único estilo de fonte do grupo esteja selecionado a cada momento.

```

1 // Fig. 12.12: RadioButtonTest.java
2 // Criação de botões de opção usando ButtonGroup e JRadioButton.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class RadioButtonTest extends JFrame {
12     private JTextField field;
13     private Font plainFont, boldFont, italicFont, boldItalicFont;
14     private JRadioButton plainButton, boldButton, italicButton,
15         boldItalicButton;
16     private ButtonGroup radioGroup;
17
18     // cria GUI e fontes
19     public RadioButtonTest()
20     {
21         super( "RadioButton Test" );
22
23         // obtém painel de conteúdo e configura o layout
24         Container container = getContentPane();
25         container.setLayout( new FlowLayout() );
26
27         // configura o JTextField
28         field =
29             new JTextField( "Watch the font style change", 25 );
30         container.add( field );
31
32         // cria botões de opção
33         plainButton = new JRadioButton( "Plain", true );
34         container.add( plainButton );
35
36         boldButton = new JRadioButton( "Bold", false );
37         container.add( boldButton );

```

Fig. 12.12 Criando e manipulando botões de opção (parte 1 de 3).

```

38         italicButton = new JRadioButton( "Italic", false );
39         container.add( italicButton );
40
41         boldItalicButton = new JRadioButton(
42             "Bold/Italic", false );
43         container.add( boldItalicButton );
44
45         // registra eventos para JRadioButtons
46         RadioButtonHandler handler = new RadioButtonHandler();
47         plainButton.addItemListener( handler );
48         boldButton.addItemListener( handler );
49         italicButton.addItemListener( handler );
50         boldItalicButton.addItemListener( handler );
51
52         // cria relacionamento lógico entre JRadioButtons
53         radioGroup = new ButtonGroup();
54         radioGroup.add( plainButton );
55         radioGroup.add( boldButton );
56         radioGroup.add( italicButton );
57         radioGroup.add( boldItalicButton );
58
59         // cria objetos Font
60         plainFont = new Font( "Serif", Font.PLAIN, 14 );
61         boldFont = new Font( "Serif", Font.BOLD, 14 );
62         italicFont = new Font( "Serif", Font.ITALIC, 14 );
63         boldItalicFont =
64             new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
65         field.setFont( plainFont );
66
67         setSize( 300, 100 );
68         setVisible( true );
69     }
70
71     // executa o aplicativo
72     public static void main( String args[] )
73     {
74         RadioButtonTest application = new RadioButtonTest();
75
76         application.setDefaultCloseOperation(
77             JFrame.EXIT_ON_CLOSE );
78     }
79
80
81     // classe interna privativa para tratar de eventos de botões de opção
82     private class RadioButtonHandler implements ItemListener {
83
84         // trata eventos de botões de opção
85         public void itemStateChanged( ItemEvent event )
86         {
87             // usuário clicou no plainButton
88             if ( event.getSource() == plainButton )
89                 field.setFont( plainFont );
90
91             // usuário clicou no boldButton
92             else if ( event.getSource() == boldButton )
93                 field.setFont( boldFont );
94
95             // usuário clicou no italicButton
96             else if ( event.getSource() == italicButton )
97                 field.setFont( italicFont );

```

Fig. 12.12 Criando e manipulando botões de opção (parte 2 de 3).

```

98
99      // usuário clicou no boldItalicButton
100     else if ( event.getSource() == boldItalicButton )
101         field.setFont( boldItalicFont );
102     }
103
104 } // fim da classe interna privativa RadioButtonHandler
105
106 } // fim da classe RadioButtonTest

```

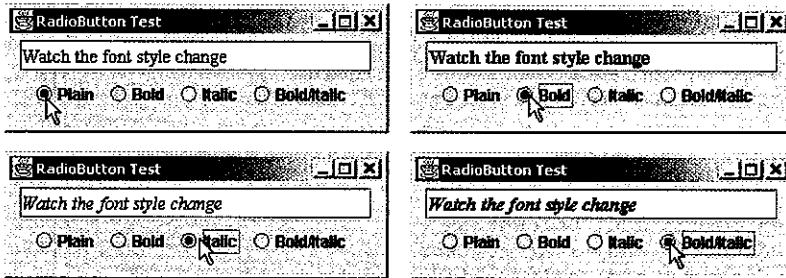


Fig. 12.12 Criando e manipulando botões de opção (parte 3 de 3).

As linhas 33 a 44 no construtor definem cada objeto **JRadioButton** e o adicionam ao painel de conteúdo da janela do aplicativo. Cada **JRadioButton** é inicializado com uma chamada de construtor como a linha 33. Esse construtor fornece o rótulo que aparece à direita do **JRadioButton** por *default* e o estado inicial do **JRadioButton**. Um segundo argumento **true** indica que o **JRadioButton** deve aparecer selecionado quando for exibido.

JRadioButtons, como **JCheckboxes**, geram **ItemEvents** quando se clica neles. As linhas 47 a 51 criam uma instância da classe interna **RadioButtonHandler** (definida nas linhas 82 a 104) e a registram para tratar o **ItemEvent** gerado quando o usuário clica em qualquer um dos **JRadioButtons**.

A linha 54 instancia um objeto **ButtonGroup** e o atribui à referência **radioGroup**. Esse objeto é a “cola” que une os quatro objetos **JRadioButton** entre si para formar o relacionamento lógico que permite que apenas um dos quatro botões seja selecionado por vez. As linhas 55 a 58 utilizam o método **add** de **ButtonGroup** para associar cada um dos **JRadioButtons** com o **radioGroup**. Se mais de um objeto **JRadioButton** selecionado for adicionado ao grupo, o primeiro **JRadioButton** adicionado estará selecionado quando a GUI for exibida.

A classe **RadioButtonHandler** (linhas 82 a 104) implementa a interface **ItemListener** para que ela possa tratar os eventos de item gerados pelos **JRadioButtons**. Cada **JRadioButton** no programa tem uma instância dessa classe (**handler**) registrada como seu **ItemListener**. Quando o usuário clica em um **JRadioButton**, **radioGroup** desativa o **JRadioButton** anteriormente selecionado e o método **itemStateChanged** (linhas 85 a 102) é executado. O método determina em qual **JRadioButton** se clicou, utilizando o método **getSource** (herdado por **ItemEvent** indiretamente de **EventObject**) e depois configura a fonte no **JTextField** com um dos objetos **Font** criados no construtor.

12.8 JComboBox

A *caixa de combinação* (ou *combo box*, às vezes também chamada de *lista escamoteável*) fornece uma lista de itens da qual o usuário pode fazer uma seleção. As caixas de combinação são implementadas com a classe **JComboBox**, que herda da classe **JComponent**. **JComboBoxes**, assim como as **JCheckboxes** e os **JRadioButtons**, geram **ItemEvents**.

O aplicativo da Fig. 12.13 utiliza uma **JComboBox** para fornecer uma lista de quatro nomes de arquivos de imagem. Quando se seleciona um nome de arquivo de imagem, a imagem correspondente é exibida como um **Icon** em um **JLabel**. As capturas de tela para esse programa mostram a lista **JComboBox** depois que a seleção foi feita para ilustrar qual nome de arquivo de imagem foi selecionado.

As linhas 17 a 19 declararam e inicializaram o *array* de ícones com quatro novos objetos **ImageIcon**. O *array* **String names** (definido nas linhas 15 e 16) contém os nomes dos quatro arquivos de imagem, que estão armazenados no mesmo diretório que o aplicativo.

A linha 31 cria um objeto `JComboBox` utilizando os `Strings` no array `names` como os elementos da lista. O índice numérico monitora a ordem dos itens na `JComboBox`. O primeiro item é adicionado no índice 0; o próximo item é adicionado no índice 1, e assim por diante. O primeiro item adicionado a uma `JComboBox` aparece como o item atualmente selecionado quando a `JComboBox` é exibida. Outros itens são selecionados clicando-se na `JComboBox`. Ao se clicar na `JComboBox`, ela se expande para uma lista, na qual o usuário pode fazer uma seleção.

A linha 32 utiliza o método `setMaximumRowCount` de `JComboBox` para estabelecer o número máximo de elementos que são exibidos quando o usuário clica na `JComboBox`. Se houver mais itens na `JComboBox` que o número máximo de elementos que são exibidos, a `JComboBox` fornece automaticamente uma *barra de rolagem* (veja a primeira captura de tela) que permite ao usuário visualizar todos os elementos na lista. O usuário pode clicar nas *setas de rolagem* na parte superior e inferior da barra de rolagem para subir e descer a lista, um elemento por vez, ou o usuário pode arrastar a *caixa de rolagem* no meio da barra de rolagem para cima e para baixo para percorrer a lista. Para arrastar a caixa de rolagem, mantenha o botão do mouse pressionado, com o cursor sobre a caixa de rolagem, e move o mouse.

```

1 // Fig. 12.13: ComboBoxTest.java
2 // Usando uma JComboBox para selecionar uma imagem a ser exibida.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class ComboBoxTest extends JFrame {
12     private JComboBox imagesComboBox;
13     private JLabel label;
14
15     private String names[] =
16         { "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };
17     private Icon icons[] = { new ImageIcon( names[ 0 ] ),
18         new ImageIcon( names[ 1 ] ), new ImageIcon( names[ 2 ] ),
19         new ImageIcon( names[ 3 ] ) };
20
21     // configur a GUI
22     public ComboBoxTest()
23     {
24         super( "Testing JComboBox" );
25
26         // obtém painel de conteúdo e configura o leiaute
27         Container container = getContentPane();
28         container.setLayout( new FlowLayout() );
29
30         // configura a JComboBox e registra o tratador de eventos
31         imagesComboBox = new JComboBox( names );
32         imagesComboBox.setMaximumRowCount( 3 );
33
34         imagesComboBox.addItemListener(
35
36             // classe interna anônima para tratar de eventos de JComboBox
37             new ItemListener() {
38
39                 // trata evento da JComboBox
40                 public void itemStateChanged( ItemEvent event )
41             {

```

Fig. 12.13 Programa que usa uma `JComboBox` para selecionar um ícone (parte 1 de 2).

```

42         // determina se a caixa de marcação está selecionada
43         if ( event.getStateChange() == ItemEvent.SELECTED )
44             label.setIcon( icons[
45                 imagesComboBox.getSelectedIndex() ] );
46     }
47
48 } // fim da classe interna anônima
49
50 ); // fim da chamada para addItemListener
51
52 container.add( imagesComboBox );
53
54 // configura o JLabel para exibir ImageIcons
55 label = new JLabel( icons[ 0 ] );
56 container.add( label );
57
58 setSize( 350, 100 );
59 setVisible( true );
60 }
61
62 // executa o aplicativo
63 public static void main( String args[] )
64 {
65     ComboBoxTest application = new ComboBoxTest();
66
67     application.setDefaultCloseOperation(
68         JFrame.EXIT_ON_CLOSE );
69 }
70
71 } // fim da classe ComboBoxTest

```

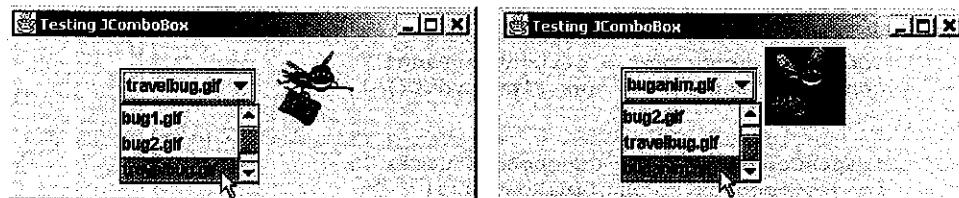
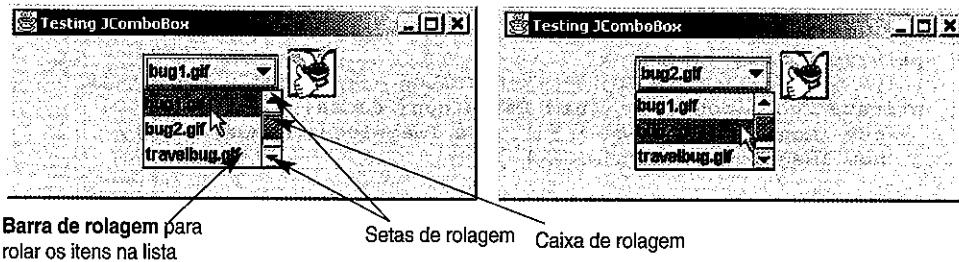


Fig. 12.13 Programa que usa uma JComboBox para selecionar um ícone (parte 2 de 2).



Observação de aparência e comportamento 12.7

Estabeleça o valor máximo de linhas para uma JComboBox como um número de linhas que impeça a lista de se expandir para fora dos limites da janela ou do applet em que ela é utilizada. Isso assegurará que a lista seja exibida corretamente quando for expandida pelo usuário.

As linhas 34 a 50 registram uma instância de uma classe interna anônima que implementa `ItemListener` como o ouvinte para a `JComboBox images`. Quando o usuário faz uma seleção de `images`, o método `itemStateChanged` (linhas 40 a 46) configura o `Icon` para `label1`. O `Icon` é selecionado do array `icons` determinando o número do índice do item selecionado na `JComboBox` através do método `getSelectedIndex` na linha 45. Observe que a linha 43 muda o ícone somente para um item selecionado. A razão para ter a estrutura `if` aqui é que, para cada item que é selecionado de uma `JComboBox`, outro item deixa de estar selecionado. Assim, ocorrem dois eventos cada vez que um item é selecionado. Desejamos exibir somente o ícone para o item que o usuário acabou de selecionar.

12.9 JList

A `lista` exibe uma série de itens da qual o usuário pode selecionar um ou mais itens. As listas são criadas com a classe `JList`, que herda da classe `JComponent`. A classe `JList` suporta *listas de uma única seleção* (isto é, listas que permitem que apenas um item seja selecionado por vez) e *listas de seleção múltipla* (listas que permitem que um número qualquer de itens seja selecionado). Nesta seção, discutimos listas de uma única seleção.

O aplicativo da Fig. 12.14 cria um `JList` de 13 cores. Quando se clica num nome de cor na `JList`, ocorre um `ListSelectionEvent` e muda a cor de fundo do painel de conteúdo da janela do aplicativo.

```

1 // Fig. 12.14: ListTest.java
2 // Selecionando cores de uma JList.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9 import javax.swing.event.*;
10
11 public class ListTest extends JFrame {
12     private JList colorList;
13     private Container container;
14
15     private String colorNames[] = { "Black", "Blue", "Cyan",
16         "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
17         "Orange", "Pink", "Red", "White", "Yellow" };
18
19     private Color colors[] = { Color.black, Color.blue,
20         Color.cyan, Color.darkGray, Color.gray, Color.green,
21         Color.lightGray, Color.magenta, Color.orange, Color.pink,
22         Color.red, Color.white, Color.yellow };
23
24     // configura a GUI
25     public ListTest()
26     {
27         super( "List Test" );
28
29         // obtém painel de conteúdo e configura o layout
30         container = getContentPane();
31         container.setLayout( new FlowLayout() );
32
33         // cria uma lista com itens do array colorNames
34         colorList = new JList( colorNames );
35         colorList.setVisibleRowCount( 5 );
36
37         // não permite seleções múltiplas
38         colorList.setSelectionMode(
39             ListSelectionModel.SINGLE_SELECTION );

```

Fig. 12.14 Selecionando cores a partir de uma `JList` (parte 1 de 2).

```

40
41     // adiciona um JScrollPane que contém JList ao painel de conteúdo
42     container.add( new JScrollPane( colorList ) );
43
44     // configura tratador de eventos
45     colorList.addListSelectionListener(
46
47         // classe interna anônima para eventos de seleção em lista
48         new ListSelectionListener() {
49
50             // trata eventos de seleção em lista
51             public void valueChanged( ListSelectionEvent event ) {
52                 {
53                     container.setBackground(
54                         colors[ colorList.getSelectedIndex() ] );
55                 }
56
57             } // fim da classe interna anônima
58
59         ); // fim da chamada para addListSelectionListener
60
61         setSize( 350, 150 );
62         setVisible( true );
63     }
64
65     // executa o aplicativo
66     public static void main( String args[] )
67     {
68         ListTest application = new ListTest();
69
70         application.setDefaultCloseOperation(
71             JFrame.EXIT_ON_CLOSE );
72     }
73
74 } // fim da classe ListTest

```

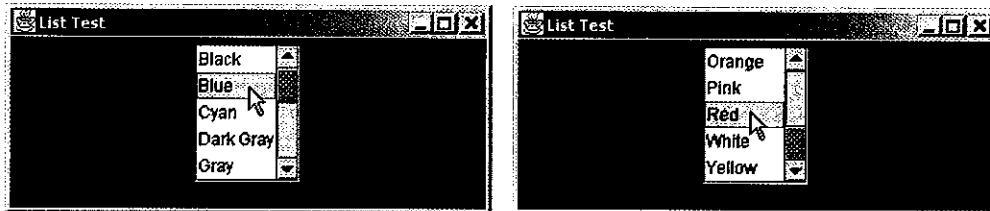


Fig. 12.14 Selezionando cores a partir de uma JList (parte 2 de 2).

O objeto `JList` é instanciado na linha 34 e atribuído à referência `colorList` no construtor. O argumento para o construtor `JList` é o array de `Objects` (nesse caso `Strings`) para exibir na lista. A linha 35 utiliza o método `setVisibleRowCount` de `JList` para determinar o número de itens que são visíveis na lista.

As linhas 38 e 39 utilizam o método `setSelectionMode` de `JList` para especificar o *modo de seleção* para a lista. A classe `ListSelectionModel` (pacote `javax.swing`) define as constantes `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` e `MULTIPLE_INTERVAL_SELECTION` para especificar um modo de seleção de uma `JList`. A lista `SINGLE_SELECTION` permite que apenas um item seja selecionado por vez. A lista `SINGLE_INTERVAL_SELECTION` é uma lista de seleção múltipla que permite que vários itens em um intervalo contíguo na lista possam ser selecionados. A lista `MULTIPLE_INTERVAL_SELECTION` é uma lista de seleção múltipla que não restringe os itens que podem ser selecionados.

Ao contrário de `JComboBox`, `JLists` não fornecem uma barra de rolagem se houver mais itens na lista que o número de linhas visíveis. Nesse caso, usa-se um objeto `JScrollPane` para fornecer a capacidade de rolagem

automática para a **JList**. A linha 42 adiciona uma nova instância de classe **JScrollPane** ao painel de conteúdo. O construtor **JScrollPane** recebe como argumento o **JComponent** para o qual ele vai fornecer a funcionalidade de rolagem automática (nesse caso, a **JList colorList**). Observe nas capturas de tela que uma barra de rolagem criada pelo **JScrollPane** aparece no lado direito da **JList**. Por *default*, a barra de rolagem aparece apenas quando o número de itens na **JList** excede o número de itens visíveis.

As linhas 45 a 59 utilizam o método **addListSelectionListener** de **JList** para registrar uma instância de uma classe interna anônima que implementa **ListSelectionListener** (definido no pacote **javax.swing.event**) como o ouvinte para a **JList colorList**. Quando o usuário faz uma seleção a partir de **colorList**, o método **valueChanged** (linhas 51 a 55) é executado e configura a cor de fundo do painel de conteúdo com o método **setBackground** (herdado da classe **Component** pela classe **Container**). A cor é selecionada do array **colors** com o índice do item selecionado na lista que é devolvido pelo método **getSelectedIndex** de **JList** (como em *arrays*, a numeração dos índices em **JList** inicia em zero).

12.10 Listas de seleção múltipla

A *lista de seleção múltipla* permite selecionar muitos itens de uma **JList**. A lista **SINGLE_INTERVAL_SELECTION** permite a seleção de um intervalo contíguo de itens na lista clicando-se no primeiro item e depois mantendo-se pressionada a tecla *Shift* enquanto se clica no último item para selecionar no intervalo. A lista **MULTIPLE_INTERVAL_SELECTION** permite a seleção de um intervalo contínuo, como descrito para uma lista **SINGLE_INTERVAL_SELECTION**, e permite que itens misturados sejam selecionados mantendo-se pressionada a tecla *Ctrl* (às vezes chamada de tecla *Control*) enquanto se clica em cada item a ser selecionado. Para anular a seleção de um item, mantenha pressionada a tecla *Ctrl* enquanto clica no item uma segunda vez.

O aplicativo da Fig. 12.15 utiliza listas de seleção múltipla para copiar itens de um **JList** para outro. Uma lista é uma lista **MULTIPLE_INTERVAL_SELECTION** e a outra é uma lista **SINGLE_INTERVAL_SELECTION**. Quando você executar o programa, tente utilizar as técnicas de seleção já descritas para selecionar os itens nas duas listas.

```

1 // Fig. 12.15: MultipleSelection.java
2 // Copiando itens de uma lista para outra.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class MultipleSelection extends JFrame {
12     private JList colorList, copyList;
13     private JButton copyButton;
14
15     private String colorNames[] = { "Black", "Blue", "Cyan",
16         "Dark Gray", "Gray", "Green", "Light Gray",
17         "Magenta", "Orange", "Pink", "Red", "White", "Yellow" };
18
19     // configura a GUI
20     public MultipleSelection()
21     {
22         super( "Multiple Selection Lists" );
23
24         // obtém painel de conteúdo e configura o layout
25         Container container = getContentPane();
26         container.setLayout( new FlowLayout() );
27
28         // configura a colorList da JList
29         colorList = new JList( colorNames );

```

Fig. 12.15 Usando uma **JList** de seleção múltipla (parte 1 de 2).

```

30     colorList.setVisibleRowCount( 5 );
31     colorList.setFixedCellHeight( 15 );
32     colorList.setSelectionMode(
33         ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
34     container.add( new JScrollPane( colorList ) );
35
36     // cria botão de cópia e registra o ouvinte
37     copyButton = new JButton( "Copy >>>" );
38
39     copyButton.addActionListener(
40
41         // classe interna anônima para evento de botão
42         new ActionListener() {
43
44             // trata evento de botão
45             public void actionPerformed( ActionEvent event )
46             {
47                 // coloca os valores selecionados na copyList
48                 copyList.setListData(
49                     colorList.getSelectedValues() );
50             }
51
52         } // fim da classe interna anônima
53
54     ); // fim da chamada para addActionListener
55
56     container.add( copyButton );
57
58     // configura a copyList da JList
59     copyList = new JList( );
60     copyList.setVisibleRowCount( 5 );
61     copyList.setFixedCellWidth( 100 );
62     copyList.setFixedCellHeight( 15 );
63     copyList.setSelectionMode(
64         ListSelectionModel.SINGLE_INTERVAL_SELECTION );
65     container.add( new JScrollPane( copyList ) );
66
67     setSize( 300, 120 );
68     setVisible( true );
69 }
70
71 // executa o aplicativo
72 public static void main( String args[] )
73 {
74     MultipleSelection application = new MultipleSelection();
75
76     application.setDefaultCloseOperation(
77         JFrame.EXIT_ON_CLOSE );
78 }
79
80 } // fim da classe MultipleSelection

```

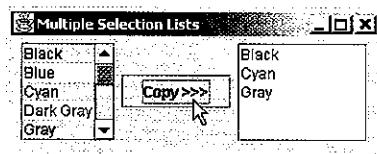


Fig. 12.15 Usando uma JList de seleção múltipla (parte 2 de 2).

A linha 29 cria a `JList colorList` e a inicializa com os `Strings` do array `colorNames`. A linha 30 estabelece o número de linhas visíveis em `colorList` como 5. A linha 31 utiliza o método `setFixedCellHeight` de `JList` para especificar a altura em *pixels* de cada item na `JList`. Fazemos isto para assegurar que as linhas nas duas `JLists` no exemplo tenham a mesma altura. As linhas 32 e 33 especificam que `colorList` é uma lista `MULTIPLE_INTERVAL_SELECTION`. A linha 34 adiciona um novo `JScrollPane` que contém a `colorList` ao painel de conteúdo. As linhas 59 a 65 realizam tarefas semelhantes para a `JList copyList`, que é definida como uma lista `SINGLE_INTERVAL_SELECTION`. A linha 61 utiliza o método `setFixedCellWidth` de `JList` para configurar a largura da `copyList` como 100 *pixels*.

Uma lista de seleção múltipla não tem um evento específico associado à realização de múltiplas seleções. Normalmente, um evento gerado por outro componente GUI (conhecido como *evento externo*) especifica quando as múltiplas seleções em uma `JList` devem ser processadas. Nesse exemplo, o usuário clica no `JButton copyButton` para disparar o evento que copia os itens selecionados em `colorList` para `copyList`.

Quando o usuário clica em `copyButton`, o método `actionPerformed` (linhas 45 a 50) é chamado. As linhas 48 e 49 utilizam o método `setListData` de `JList` para configurar os itens exibidos em `copyList`. A linha 49 chama o método `getSelectedValues` de `colorList`, o qual devolve um array de `Objects` que representa os itens selecionados em `colorList`. Neste exemplo, o array devolvido é passado como argumento para o método `setListData` de `colorList`.

Muitos estudantes perguntam como a referência `colorList` pode ser usada na linha 48, quando o programa não cria o objeto para o qual ela faz referência antes da linha 59. Lembre-se de que o método `actionPerformed`, nas linhas 45 a 50, não é executado até que o usuário pressione o `copyButton`, o que não pode ocorrer antes de a chamada para o construtor ser completada. Neste ponto da execução do programa, a linha 59 já inicializou `colorList` com um novo objeto `JList`.

12.11 Tratamento de eventos de mouse

Esta seção apresenta as interfaces *listeners* de eventos `MouseListener` e `MouseMotionListener` para tratar *eventos de mouse*. Os eventos de mouse podem ser capturados por qualquer componente GUI que se derive de `java.awt.Component`. Os métodos das interfaces `MouseListener` e `MouseMotionListener` são resumidos na Fig. 12.16.

Cada um dos métodos de tratamento de eventos do mouse recebe um objeto `MouseEvent` como argumento. O objeto `MouseEvent` contém as informações sobre o evento que ocorreu, incluindo as coordenadas *x* e *y* da posição em que o evento ocorreu. Os métodos `MouseListener` e `MouseMotionListener` são chamados automaticamente quando o mouse interage com um `Component` se existirem objetos *listeners* registrados para um `Component` particular. O método `mousePressed` é chamado quando um botão do mouse é pressionado com o cursor do mouse sobre um componente. Utilizando métodos e constantes da classe `InputEvent` (a superclasse de `MouseEvent`), o programa pode determinar em que botão do mouse o usuário clicou. O método `mouseClicked` é chamado sempre que um botão do mouse é liberado sem mover o mouse, depois de uma operação `mousePressed`. O método `mouseReleased` é chamado sempre que um botão do mouse é liberado. O método `mouseEntered` é chamado quando o cursor do mouse entra nos limites físicos de um `Component`. O método `mouseExited` é chamado quando o cursor de mouse deixa os limites físicos de um `Component`. O método `mouseDragged` é chamado quando o botão do mouse é pressionado e mantido pressionado e o mouse é movido (processo conhecido como *arrastar*). O evento `mouseDragged` é precedido por um evento `mousePressed` e seguido por um evento `mouseReleased`. O método `mouseMoved` é chamado quando o mouse é movido com o cursor sobre um componente (e nenhum botão do mouse está pressionado).

Métodos das interfaces `MouseListener` e `MouseMotionListener`

Métodos da interface `MouseListener`

```
public void mousePressed( MouseEvent event )
```

Chamado quando se pressiona um botão do mouse com o cursor sobre um componente.

Fig. 12.16 Métodos das interfaces `MouseListener` e `MouseMotionListener` (parte 1 de 2).

Métodos das interfaces MouseListener e MouseMotionListener

```

public void mouseClicked( MouseEvent event )
    Chamado quando se pressiona e se libera um botão do mouse sobre um componente, sem mover o cursor.
public void mouseReleased( MouseEvent event )
    Chamado quando se libera um botão do mouse depois de ser pressionado. Esse evento sempre é precedido por um evento
        mousePressed.
public void mouseEntered( MouseEvent event )
    Chamado quando o cursor do mouse entra nos limites de um componente.
public void mouseExited( MouseEvent event )
    Chamado quando o cursor do mouse deixa os limites de um componente.

```

Métodos da interface MouseMotionListener

```

public void mouseDragged( MouseEvent event )
    Chamado quando se pressiona o botão do mouse com o cursor sobre um componente e se move o mouse. Esse evento é
        sempre precedido por uma chamada para mousePressed.
public void mouseMoved( MouseEvent event )
    Chamado quando se move o mouse com o cursor sobre um componente.

```

Fig. 12.16 Métodos das interfaces **MouseListener** e **MouseMotionListener** (parte 2 de 2).*Observação de aparência e comportamento 12.8*

As chamadas de método para **mouseDragged** são enviadas para o **MouseMotionListener** do **Component** sobre o qual a operação de arrastar iniciou. De forma semelhante, a chamada para o método **mouseReleased** é enviada para o **MouseListener** do **Component** sobre o qual a operação de arrastar iniciou.

O aplicativo **MouseTracker** (Fig. 12.17) demonstra os métodos **MouseListener** e **MouseMotionListener**. A classe do aplicativo implementa as duas interfaces de modo a poder esperar seus próprios eventos de mouse. Observe que todos os sete métodos dessas duas interfaces devem ser definidos pelo programador quando uma classe implementa as duas interfaces. A caixa de diálogo de mensagem no exemplo da janela de saída aparece quando o usuário move o mouse para dentro da janela do aplicativo.

```

1 // Fig. 12.17: MouseTracker.java
2 // Demonstrando eventos de mouse.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class MouseTracker extends JFrame
12     implements MouseListener, MouseMotionListener {
13
14     private JLabel statusBar;
15
16     // configura a GUI e registra os tratadores de eventos do mouse
17     public MouseTracker()
18     {

```

Fig. 12.17 Demonstrando o tratamento de eventos do mouse (parte 1 de 3).

```

19     super( "Demonstrating Mouse Events" );
20
21     statusBar = new JLabel();
22     getContentPane().add( statusBar, BorderLayout.SOUTH );
23
24     // aplicativo espera seus próprios eventos do mouse
25     addMouseListener( this );
26     addMouseMotionListener( this );
27
28     setSize( 275, 100 );
29     setVisible( true );
30 }
31
32 // tratadores de eventos MouseListener
33
34 // trata evento quando o mouse é liberado imediatamente após ser pressionado
35 public void mouseClicked( MouseEvent event )
36 {
37     statusBar.setText( "Clicked at [" + event.getX() +
38         ", " + event.getY() + "]" );
39 }
40
41 // trata evento quando o mouse é pressionado
42 public void mousePressed( MouseEvent event )
43 {
44     statusBar.setText( "Pressed at [" + event.getX() +
45         ", " + event.getY() + "]" );
46 }
47
48 // trata evento quando o mouse é liberado após ser arrastado
49 public void mouseReleased( MouseEvent event )
50 {
51     statusBar.setText( "Released at [" + event.getX() +
52         ", " + event.getY() + "]" );
53 }
54
55 // trata evento quando o mouse entra na área da janela na tela
56 public void mouseEntered( MouseEvent event )
57 {
58     JOptionPane.showMessageDialog( null, "Mouse in window" );
59 }
60
61 // trata evento quando o mouse sai da área da janela na tela
62 public void mouseExited( MouseEvent event )
63 {
64     statusBar.setText( "Mouse outside window" );
65 }
66
67 // tratadores de eventos MouseMotionListener
68
69 // trata evento quando o usuário arrasta o mouse com o botão pressionado
70 public void mouseDragged( MouseEvent event )
71 {
72     statusBar.setText( "Dragged at [" + event.getX() +
73         ", " + event.getY() + "]" );
74 }
75
76 // trata evento quando o usuário movimenta o mouse
77 public void mouseMoved( MouseEvent event )
78 {

```

Fig. 12.17 Demonstrando o tratamento de eventos do mouse (parte 2 de 3).

```

79     statusBar.setText( "Moved at [" + event.getX() +
80         ", " + event.getY() + "] " );
81 }
82
83 // executa o aplicativo
84 public static void main( String args[] )
85 {
86     MouseTracker application = new MouseTracker();
87
88     application.setDefaultCloseOperation(
89         JFrame.EXIT_ON_CLOSE );
90 }
91
92 } // fim da classe MouseTracker

```

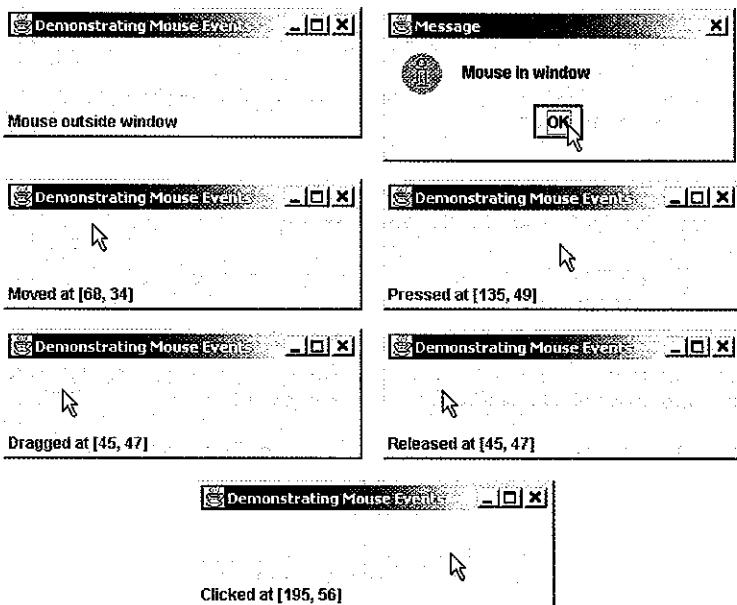


Fig. 12.17 Demonstrando o tratamento de eventos do mouse (parte 3 de 3).

Cada evento do mouse resulta na exibição de um `String` no `JLabel statusBar` na parte inferior da janela.

As linhas 21 e 22 no construtor definem o `JLabel statusBar` e o anexam ao painel de conteúdo. Até agora, cada vez que utilizamos o painel de conteúdo, o método `setLayout` foi chamado para configurar o gerenciador de layout do painel de conteúdo como um `FlowLayout`. Isso permitiu que o painel de conteúdo exibisse os componentes GUI que anexamos a ele da esquerda para a direita. Se os componentes GUI não cabem em uma linha, o `FlowLayout` cria linhas adicionais para continuar exibindo os componentes GUI. Na verdade, o gerenciador `default` de layout é um `BorderLayout` que divide a área do painel de conteúdo em cinco regiões – norte, sul, leste, oeste e centro. A linha 22 utiliza uma nova versão do método `add` de `Container` para anexar `statusBar` à região `BorderLayout.SOUTH`, que se estende ao longo de toda a parte inferior do painel de conteúdo. Discutimos `BorderLayout` e vários outros gerenciadores de layout em detalhes mais adiante neste capítulo.

As linhas 25 e 26 no construtor registram o objeto de janela `MouseTracker` como o ouvinte para seus próprios eventos de mouse. Os métodos `addMouseListener` e `addMouseMotionListener` são os métodos de `Component` que podem ser utilizados para registrar ouvintes de eventos de mouse para um objeto de qualquer classe que estenda `Component`.

Quando o mouse entra na área do aplicativo ou sai dela, o método `mouseEntered` (linhas 56 a 59) e o método `mouseExited` (linhas 62 a 65) são chamados, respectivamente. O método `mouseExited` exibe uma mensa-

gem na `statusBar` indicando que o mouse está fora do aplicativo (veja o primeiro exemplo de janela de saída). O método `mouseEntered` exibe uma caixa de diálogo de mensagem indicando que o mouse entrou na janela do aplicativo. [Nota: assegure-se de pressionar *Enter* para fechar o diálogo de mensagem, em vez de usar o mouse. Se você usar o mouse para fechar o diálogo, quando você mover o mouse sobre a janela novamente, `mouseEntered` exibe novamente o diálogo. Isto impedirá que você teste os outros eventos de mouse.]

Quando ocorrer qualquer um dos outros cinco eventos, eles exibem uma mensagem na `statusBar` que inclui um `String` representando o evento que ocorreu e as coordenadas em que o evento ocorreu. As coordenadas *x* e *y* do mouse no momento em que o evento ocorreu são obtidas com os métodos `getX` e `getY` de `MouseEvent`, respectivamente.

12.12 Classes adaptadoras

Muitas das interfaces *listeners* de eventos fornecem múltiplos métodos; `MouseListener` e `MouseMotionListener` são alguns exemplos. Nem sempre é desejável definir todos os métodos em uma interface *listener* de evento. Por exemplo, o programa pode precisar apenas do tratador `mouseClicked` da interface `MouseListener` ou do tratador `mouseDragged` de `MouseMotionListener`. Em nossos aplicativos com janela (subclasses de `JFrame`), o término do aplicativo foi tratado com o `windowClosing` da interface `WindowListener`, que na realidade especifica sete métodos de tratamento de eventos de janela. Para muitas das interfaces *listeners* que contêm múltiplos métodos, os pacotes `java.awt.event` e `javax.swing.event` fornecem *classes adaptadoras* de ouvintes de eventos. Uma classe adaptadora implementa uma interface e fornece uma implementação *default* (com o corpo de método vazio) de cada método na interface. As classes adaptadoras `java.awt.event` são mostradas na Fig. 12.18 junto com as interfaces que elas implementam.

O programador pode estender a classe adaptadora para herdar a implementação *default* de cada método e depois sobrescrever o(s) método(s) necessário(s) para o tratamento de eventos. A implementação *default* de cada método na classe adaptadora tem um corpo vazio. Isso é exatamente o que temos feito em cada exemplo de aplicativo que estende `JFrame` e define o método `windowClosing` para tratar o fechamento da janela e o encerramento do aplicativo.



Observação de engenharia de software 12.3

Quando uma classe implementa uma interface, a classe tem um relacionamento “é um” com aquela interface. Todas as subclasses diretas e indiretas daquela classe herdaram este relacionamento. Portanto, um objeto de uma classe que estende uma classe adaptadora de evento é um objeto do tipo listener de evento correspondente (por exemplo, um objeto de uma subclasse de `MouseAdapter` é um `MouseListener`).

Classe adaptadora de eventos	Implementa a interface
<code>ComponentAdapter</code>	<code>ComponentListener</code>
<code>ContainerAdapter</code>	<code>ContainerListener</code>
<code>FocusAdapter</code>	<code>FocusListener</code>
<code>KeyAdapter</code>	<code>KeyListener</code>
<code>MouseAdapter</code>	<code>MouseListener</code>
<code>MouseMotionAdapter</code>	<code>MouseMotionListener</code>
<code>WindowAdapter</code>	<code>WindowListener</code>

Fig. 12.18 As classes adaptadoras de eventos e as interfaces que elas implementam.

O aplicativo `Painter` da Fig. 12.19 utiliza o tratador de evento `mouseDragged` para criar um programa simples de desenho. O usuário pode desenhar figuras com o mouse, arrastando-o sobre o fundo da janela. Esse exemplo não utiliza o método `mouseMoved`, de modo que nosso `MouseMotionListener` é definido como uma subclasse de `MouseMotionAdapter`. Essa classe já define `mouseMoved` e `mouseDragged`, de modo que podemos simplesmente sobrescrever `mouseDragged` para fornecer a funcionalidade de desenho.

```

1 // Fig. 12.19: Painter.java
2 // Usando a classe MouseMotionAdapter.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class Painter extends JFrame {
12     private int xValue = -10, yValue = -10;
13
14     // configura a GUI e registra o tratador de eventos do mouse
15     public Painter()
16     {
17         super( "A simple paint program" );
18
19         // cria um rótulo e o coloca na posição SOUTH do BorderLayout
20         getContentPane().add(
21             new Label( "Drag the mouse to draw" ),
22             BorderLayout.SOUTH );
23
24         addMouseMotionListener(
25
26             // classe interna anônima
27             new MouseMotionAdapter() {
28
29                 // armazena coordenadas de arrasto do mouse e redesenha janela
30                 public void mouseDragged( MouseEvent event )
31                 {
32                     xValue = event.getX();
33                     yValue = event.getY();
34                     repaint();
35                 }
36
37             } // fim da classe interna anônima
38
39         ); // fim da chamada para addMouseMotionListener
40
41         setSize( 300, 150 );
42         setVisible( true );
43     }
44
45     // desenha elipse em uma caixa delimitadora 4 por 4
46     // na posição especificada da janela
47     public void paint( Graphics g )
48     {
49         // não chamamos super.paint( g ) aqui de propósito,
50         // para evitar que a janela seja redesenhadada
51
52         g.fillOval( xValue, yValue, 4, 4 );
53     }
54
55     // executa o aplicativo
56     public static void main( String args[] )
57     {
58         Painter application = new Painter();
59
60         application.addWindowListener(

```

Fig. 12.19 Programa que demonstra classes adaptadoras (parte 1 de 2).

```

61      // adaptadora para tratar somente do evento windowClosing
62      new WindowAdapter() {
63
64          public void windowClosing( WindowEvent event )
65          {
66              System.exit( 0 );
67          }
68
69      } // fim da classe interna anônima
70
71      ); // fim da chamada para addWindowListener
72  }
73
74 }
75 } // fim da classe Painter

```

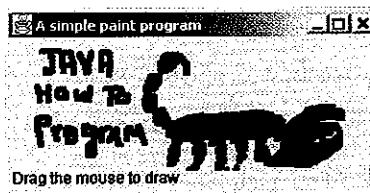


Fig. 12.19 Programa que demonstra classes adaptadoras (parte 2 de 2).

As variáveis de instância `xValue` e `yValue` armazenam as coordenadas do evento `mouseDragged`. Inicialmente, as coordenadas são configuradas fora da área da janela para evitar que uma elipse seja desenhada na área de fundo na primeira chamada a `paint` quando a janela é exibida. As linhas 24 a 39 registram um `MouseMotionListener` para esperar os eventos de movimento do mouse na janela (lembre-se de que a chamada para um método que não é precedida por uma referência e por um operador de ponto é, na realidade, precedida por “`this.`”, indicando que o método é chamado para a instância atual da classe durante a execução). As linhas 27 a 37 definem uma classe interna anônima que estende a classe `MouseMotionAdapter` (que implementa `MouseMotionListener`). A classe interna anônima herda uma implementação *default* tanto do método `mouseMoved` quanto de `mouseDragged`. Portanto, a classe interna anônima já satisfaz o requisito de que todos os métodos de uma interface devem ser implementados. Entretanto, os métodos *default* não fazem nada quando são chamados. Portanto, sobrescrevemos o método `mouseDragged` nas linhas 30 a 35 para capturar as coordenadas `x` e `y` do evento de mouse arrastado e as armazenamos nas variáveis de instância `xValue` e `yValue`; depois, chamamos `repaint` para começar a desenhar a próxima elipse sobre o fundo (o que é feito pelo método `paint` nas linhas 47 a 53). Note que `paint` não chama a versão de `paint` herdada da superclasse `JFrame`. A versão da superclasse normalmente limpa o fundo da janela. Não chamar a versão da superclasse permite que nosso programa mantenha todas as elipses na janela ao mesmo tempo. Entretanto, repare que, se você cobrir a janela com uma outra janela, somente a última elipse exibida ainda aparecerá, porque seu programa não mantém registro de todas as elipses exibidas anteriormente.

As linhas 60 a 72 registram um `WindowListener` para esperar os eventos de janela da janela do aplicativo (tais como fechar a janela). As linhas 63 a 70 definem uma classe interna anônima que estende a classe `WindowAdapter` (que implementa `WindowListener`). A classe interna anônima herda uma implementação *default* de sete métodos diferentes de tratamento de eventos de janelas. Portanto, a classe interna anônima já satisfaz o requisito de que todos os métodos de uma interface devem ser implementados. Entretanto, os métodos *default* não fazem nada quando são chamados. Então, sobrescrevemos o método `windowClosing` nas linhas 65 a 68 para terminar o aplicativo quando o usuário clica no botão de fechamento da janela do aplicativo.

O aplicativo `MouseDetails` da Fig. 12.20 demonstra como determinar o número de cliques do mouse (isto é, a quantidade de cliques) e como distinguir os diferentes botões do mouse. O ouvinte (*listener*) de eventos nesse programa é um objeto da classe interna `MouseClickHandler` (linhas 47 a 75) que estende `MouseListener` para que possamos definir apenas o método `mouseClicked` de que precisamos nesse exemplo.

```

1 // Fig. 12.20: MouseDetails.java
2 // Demonstrando cliques do mouse e
3 // distinguindo os botões do mouse.
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class MouseDetails extends JFrame {
13     private int xPos, yPos;
14
15     // configura o String da barra de título, registra o
16     // listener do mouse, dimensiona e mostra a janela
17     public MouseDetails()
18     {
19         super( "Mouse clicks and buttons" );
20
21         addMouseListener( new MouseClickHandler() );
22
23         setSize( 350, 150 );
24         setVisible( true );
25     }
26
27     // desenha o String na posição na qual se clicou com o mouse
28     public void paint( Graphics g )
29     {
30         // chama o método paint da superclasse
31         super.paint( g );
32
33         g.drawString( "Clicked @ [" + xPos + ", " + yPos + "]",
34             xPos, yPos );
35     }
36
37     // executa o aplicativo
38     public static void main( String args[] )
39     {
40         MouseDetails application = new MouseDetails();
41
42         application.setDefaultCloseOperation(
43             JFrame.EXIT_ON_CLOSE );
44     }
45
46     // classe interna para tratar eventos do mouse
47     private class MouseClickHandler extends MouseAdapter {
48
49         // trata evento de clique do mouse e determina
50         // qual botão foi pressionado
51         public void mouseClicked( MouseEvent event )
52         {
53             xPos = event.getX();
54             yPos = event.getY();
55
56             String title =
57                 "Clicked " + event.getClickCount() + " time(s)";
58

```

Fig. 12.20 Distinguindo o clique do botão esquerdo, do meio e direito do mouse (parte 1 de 2).

```

59         // botão direito do mouse
60     if ( event.isMetaDown() )
61         title += " with right mouse button";
62
63     // botão do meio do mouse
64     else if ( event.isAltDown() )
65         title += " with center mouse button";
66
67     // botão esquerdo do mouse
68     else
69         title += " with left mouse button";
70
71     setTitle( title ); // configura a barra de título da janela
72     repaint();
73 }
74
75 } // fim da classe interna privativa MouseClickHandler
76
77 } // fim da classe MouseDetails

```

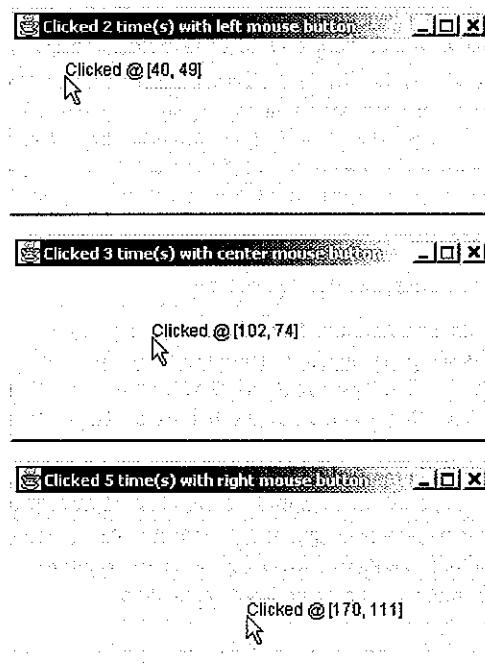


Fig. 12.20 Distinguindo o clique do botão esquerdo, do meio e direito do mouse (parte 2 de 2).

O usuário de um programa Java pode estar em um sistema com um mouse de um, dois ou três botões. Java fornece um mecanismo para distinguir entre os botões do mouse. A classe **MouseEvent** herda vários métodos da classe **InputEvent** que podem distinguir os botões do mouse em um mouse de múltiplos botões ou podem simular um mouse de múltiplos botões com um pressionamento de tecla combinado com um clique do botão do mouse. A Fig. 12.21 mostra os métodos **InputEvent** utilizados para distinguir os cliques do botão do mouse. Java supõe que cada mouse contém um botão esquerdo. Portanto, é simples testar um clique com o botão esquerdo. Entretanto, os usuários que têm mouse de um ou dois botões devem utilizar uma combinação de pressionar teclas no teclado e clicar no mouse ao mesmo tempo, para simular os botões inexistentes. No caso de um mouse com um ou dois botões, esse programa supõe que se clica no botão do meio se o usuário mantém pressionada a tecla *Alt* e clica no bo-

tão esquerdo do mouse em um mouse de dois botões, ou no único botão num mouse de um botão. No caso de um mouse de um botão, esse programa supõe que se clica no botão direito do mouse se o usuário mantém pressionada a tecla *Meta* e clica no botão.

O método **mouseClicked** (linhas 51 a 73) primeiro captura as coordenadas nas quais o evento ocorreu e as armazena nas variáveis de instância **xPos** e **yPos** da classe **MouseDetails**. As linhas 56 e 57 criam um *string* que contém o número de cliques do mouse (como devolvido pelo método **getClickCount** de **MouseEvent** na linha 57). A estrutura aninhada **if** nas linhas 60 a 69 utiliza os métodos **isMetaDown** e **isAltDown** para determinar com qual botão do mouse o usuário clicou e acrescenta um *string* apropriado a **title** em cada caso. O *string* resultante é exibido na barra de título da janela com o método **setTitle** (herdado pela classe **JFrame** da classe **Frame**) na linha 71. A linha 72 chama **repaint** para iniciar uma chamada a **paint** para desenhar um *string* na posição da janela em que o usuário clicou.

Método InputEvent	Descrição
isMetaDown()	Esse método devolve true quando o usuário clica no botão direito de um mouse com dois ou três botões. Para simular um clique com o botão direito em um mouse de um botão, o usuário pode pressionar a tecla <i>Meta</i> no teclado e clicar no botão.
isAltDown()	Esse método devolve true quando o usuário clica no botão do meio do mouse em um mouse que tem três botões. Para simular um clique com o botão do meio do mouse em um mouse com um ou dois botões, o usuário pode pressionar a tecla <i>Alt</i> no teclado e clicar no botão.

Fig. 12.21 Métodos **InputEvent** que ajudam a distinguir entre cliques com o botão esquerdo, do meio e direito do mouse.

12.13 Tratamento de eventos de teclado

Esta seção apresenta a interface *listener* de eventos **KeyListener** para tratar os *eventos de teclas*. Os eventos de teclas são gerados quando as teclas são pressionadas e liberadas. Uma classe que implementa **KeyListener** deve fornecer definições para os métodos **keyPressed**, **keyReleased** e **keyTyped**, e cada um deles recebe um **KeyEvent** como argumento. A classe **KeyEvent** é uma subclasse de **InputEvent**. O método **keyPressed** é chamado em resposta ao pressionamento de qualquer tecla. O método **keyTyped** é chamado em resposta ao pressionamento de qualquer tecla que não é uma *tecla de ação* (por exemplo, uma tecla de seta, *Home*, *End*, *Page Up*, *Page Down*, uma tecla de função, *Num Lock*, *Print Screen*, *Scroll Lock*, *Caps Lock* e *Pause*). O método **keyReleased** é chamado quando se solta a tecla depois de qualquer evento **keyPressed** ou **keyTyped**.

A Fig. 12.22 demonstra os métodos de **KeyListener**. A classe **KeyDemo** implementa a interface **KeyListener**, de modo que todos os três métodos são definidos no aplicativo.

```

1 // Fig. 12.22: KeyDemo.java
2 // Demonstrando eventos de teclas.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class KeyDemo extends JFrame implements KeyListener {
12     private String line1 = "", line2 = "";
13     private String line3 = "";
14     private JTextArea textArea;
15

```

Fig. 12.22 Demonstrationando tratamento de eventos de teclas (parte 1 de 3).

```

16    // configura a GUI
17    public KeyDemo()
18    {
19        super( "Demonstrating Keystroke Events" );
20
21        // configura a JTextArea
22        textArea = new JTextArea( 10, 15 );
23        textArea.setText( "Press any key on the keyboard..." );
24        textArea.setEnabled( false );
25        getContentPane().add( textArea );
26
27        // permite que a frame processe os eventos de teclas
28        addKeyListener( this );
29
30        setSize( 350, 100 );
31        setVisible( true );
32    }
33
34    // trata o pressionamento de qualquer tecla
35    public void keyPressed( KeyEvent event )
36    {
37        line1 = "Key pressed: " +
38            event.getKeyText( event.getKeyCode() );
39        setLines2and3( event );
40    }
41
42    // trata a liberação de qualquer tecla
43    public void keyReleased( KeyEvent event )
44    {
45        line1 = "Key released: " +
46            event.getKeyText( event.getKeyCode() );
47        setLines2and3( event );
48    }
49
50    // trata o pressionamento de uma tecla de ação
51    public void keyTyped( KeyEvent event )
52    {
53        line1 = "Key typed: " + event.getKeyChar();
54        setLines2and3( event );
55    }
56
57    // configura a segunda e a terceira linhas de saída
58    private void setLines2and3( KeyEvent event )
59    {
60        line2 = "This key is " +
61            ( event.isActionKey() ? "" : "not " ) +
62            "an action key";
63
64        String temp =
65            event.getKeyModifiersText( event.getModifiers() );
66
67        line3 = "Modifier keys pressed: " +
68            ( temp.equals( "" ) ? "none" : temp );
69
70        textArea.setText(
71            line1 + "\n" + line2 + "\n" + line3 + "\n" );
72    }
73
74    // executa o aplicativo
75    public static void main( String args[] )
76    {

```

Fig. 12.22 Demonstrando tratamento de eventos de teclas (parte 2 de 3).

```

77     KeyDemo application = new KeyDemo();
78
79     application.setDefaultCloseOperation(
80         JFrame.EXIT_ON_CLOSE );
81 }
82
83 } // fim da classe KeyDemo

```

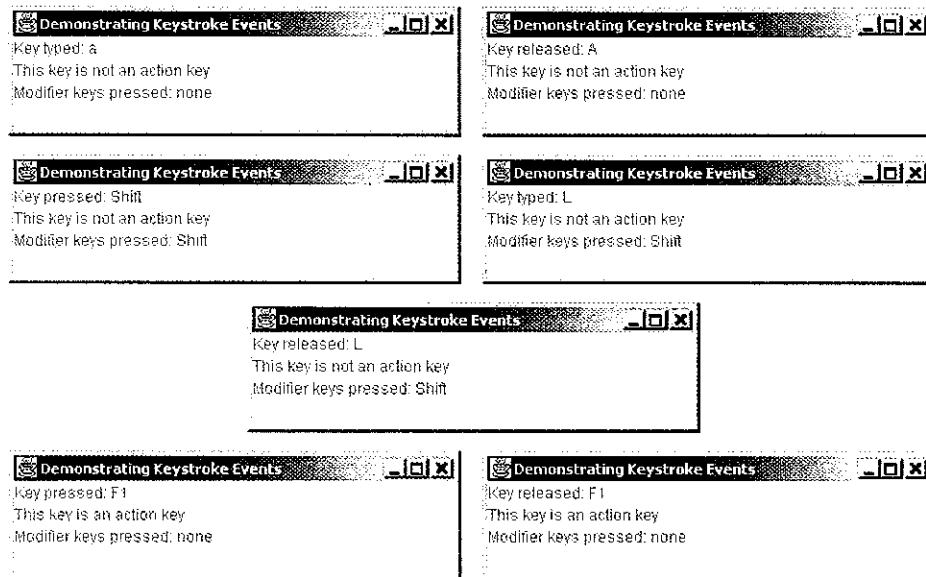


Fig. 12.22 Demonstrando tratamento de eventos de teclas (parte 3 de 3).

O construtor (linhas 17 a 32) registra o aplicativo para tratar seus próprios eventos de teclas com o método `addKeyListener` na linha 28. O método `addKeyListener` é definido na classe `Component`, depois cada sub-classe de `Component` pode notificar `KeyListeners` de eventos de teclas para aquele `Component`.

A linha 25 no construtor adiciona a `JTextArea textArea` (em que a saída do programa é exibida) ao painel de conteúdo. Repare, nas capturas de tela, que `textArea` ocupa a janela inteira. Isso se deve ao layout `default BorderLayout` do painel de conteúdo (discutido na Seção 12.14.2 e demonstrado na Fig. 12.25). Quando um único `Component` é adicionado a um `BorderLayout`, o `Component` ocupa o `Container` inteiro.

Os métodos `keyPressed` (linhas 35 a 40) e `keyReleased` (linhas 43 a 48) utilizam o método `getKeyCode` de `KeyEvent` para obter o *código de tecla virtual* da chave que foi pressionada. A classe `KeyEvent` mantém um conjunto de constantes – as constantes de códigos de tecla virtual – que representam todas as teclas do teclado. Essas constantes podem ser comparadas com o valor de retorno de `getKeyCode` para testar as teclas individuais no teclado. O valor devolvido por `getKeyCode` é passado para o método `getKeyText` de `KeyEvent`, que devolve um `String` que contém o nome da tecla que foi pressionada. Para uma lista completa de constantes de teclas virtuais, veja a documentação *on-line* da classe `KeyEvent` (pacote `java.awt.event`). O método `keyTyped` (linhas 51 a 55) utiliza o método `getKeyChar` de `KeyEvent` para obter o valor Unicode do caractere digitado.

Todos os três métodos de tratamento de eventos terminam chamando o método `setLines2and3` (linhas 58 a 72) e passando para ele o objeto `KeyEvent`. Esse método utiliza o método `isActionKey` de `KeyEvent` para determinar se a tecla no evento era uma tecla de ação. Além disso, o método `getModifiers` de `InputEvent` é chamado para determinar se alguma tecla modificadora (como *Shift*, *Alt* e *Ctrl*) estava pressionada quando ocorreu o evento de tecla. O resultado desse método é passado para o método `getModifiersText` de `KeyEvent`, que produz um `String` que contém os nomes das teclas modificadoras pressionadas.

[*Nota:* se você precisa testar uma tecla específica no teclado, a classe `KeyEvent` fornece uma *constante de tecla* para cada tecla no teclado. Essas constantes podem ser utilizadas a partir dos tratadores de eventos de teclas para determinar se uma tecla particular foi pressionada. Além disso, para determinar se as teclas *Alt*, *Ctrl*, *Meta* e *Shift*

estão pressionadas individualmente, cada um dos métodos `isAltDown`, `isControlDown`, `isMetaDown` e `isShiftDown` de `InputEvent` devolve um `boolean` que indica se a tecla particular foi pressionada durante o evento de tecla.]

12.14 Gerenciadores de leiaute

Os *gerenciadores de leiaute* são fornecidos para organizar componentes GUI em um contêiner para fins de apresentação. Os gerenciadores de leiaute fornecem recursos básicos de leiaute que são mais fáceis de utilizar do que determinar a posição e o tamanho exatos de cada componente GUI. Isso permite que o programador se concentre na “aparência e no comportamento” básicos e deixar os gerenciadores de leiaute processar a maioria dos detalhes de leiaute.



Observação de aparência e comportamento 12.9

A maioria dos ambientes de programação Java fornece ferramentas GUI de projeto que ajudam o programador a projetar graficamente uma GUI e depois escrever automaticamente o código Java para criar a GUI.

Algumas ferramentas de projeto de GUI também permitem que o programador utilize os gerenciadores de leiaute descritos aqui e no Capítulo 13. A Fig. 12.23 resume os gerenciadores de leiaute apresentados neste capítulo. Outros gerenciadores de leiaute são discutidos no Capítulo 13.

Gerenciador de leiaute	Descrição
<code>FlowLayout</code>	Padrão para <code>java.awt.Applet</code> , <code>java.awt.Panel</code> e <code>javax.swing.JPanel</code> . Coloca os componentes seqüencialmente (da esquerda para a direita) na ordem em que foram adicionados. Também é possível especificar a ordem dos componentes utilizando o método <code>add</code> de <code>Container</code> que recebe um <code>Component</code> e um índice de posição inteiro como argumentos.
<code>BorderLayout</code>	Padrão para os painéis de conteúdo de <code>JFrames</code> (e outras janelas) e <code>JApplets</code> . Organiza os componentes em cinco áreas: norte, sul, leste, oeste e centro.
<code>GridLayout</code>	Organiza os componentes em linhas e colunas.

Fig. 12.23 Gerenciadores de leiaute.

A maioria dos exemplos de *applets* e de aplicativos anteriores em que criamos nossa própria GUI utilizou o gerenciador de leiaute `FlowLayout`. A classe `FlowLayout` herda da classe `Object` e implementa a interface `LayoutManager`, que define os métodos que um gerenciador de leiaute utiliza para organizar e dimensionar os componentes GUI em um contêiner.

12.14.1 `FlowLayout`

`FlowLayout` é o gerenciador de leiaute mais básico. Os componentes GUI são colocados em um contêiner da esquerda para a direita, na ordem em que são adicionados ao contêiner. Quando se alcança a borda do contêiner, os componentes continuam na próxima linha. A classe `FlowLayout` permite que os componentes GUI sejam *alinhados à esquerda, centralizados (o padrão) e alinhados à direita*.

O aplicativo da Fig. 12.24 cria três objetos `JButton` e os adiciona ao aplicativo, utilizando um gerenciador de leiaute `FlowLayout`. Os componentes são automaticamente alinhados no centro. Quando o usuário clica em `Left`, o alinhamento para o gerenciador de leiaute é alterado para um `FlowLayout` alinhado à esquerda. Quando o usuário clica em `Right`, o alinhamento é alterado para um `FlowLayout` alinhado à direita. Quando o usuário clica em `Center`, o alinhamento é alterado para um `FlowLayout` alinhado no centro. Cada botão tem seu próprio tratador de eventos, que é definido com uma classe interna que implementa `ActionListener`. Os exemplos de janela de saída mostram cada um dos alinhamentos de `FlowLayout`. Além disso, o último exemplo de janela de saída mostra o alinhamento centrado depois que a janela foi redimensionada para uma largura menor. Repare que o botão `Right` agora aparece em uma nova linha.

```
1 // Fig. 12.24: FlowLayoutDemo.java
2 // Demonstrando alinhamento de componentes num FlowLayout.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class FlowLayoutDemo extends JFrame {
12     private JButton leftButton, centerButton, rightButton;
13     private Container container;
14     private FlowLayout layout;
15
16     // configura a GUI e registra as ouvintes de botões
17     public FlowLayoutDemo()
18     {
19         super( "FlowLayout Demo" );
20
21         layout = new FlowLayout();
22
23         // obtém painel de conteúdo e configura o leiaute
24         container = getContentPane();
25         container.setLayout( layout );
26
27         // configura leftButton e registra ouvintes
28         leftButton = new JButton( "Left" );
29
30         leftButton.addActionListener(
31
32             // classe interna anônima
33             new ActionListener() {
34
35                 // processa evento do leftButton
36                 public void actionPerformed( ActionEvent event )
37                 {
38                     layout.setAlignment( FlowLayout.LEFT );
39
39                     // realinha componentes anexados
40                     layout.layoutContainer( container );
41                 }
42             }
43
44         } // fim da classe interna anônima
45
46     ); // fim da chamada para addActionListener
47
48     container.add( leftButton );
49
50     // configura o centerButton e registra ouvinte
51     centerButton = new JButton( "Center" );
52
53     centerButton.addActionListener(
54
55         // classe interna anônima
56         new ActionListener() {
57
58             // processa evento do centerButton
59             public void actionPerformed( ActionEvent event )
60             {
61                 layout.setAlignment( FlowLayout.CENTER );
62             }
63         }
64     );
65 }
```

Fig. 12.24 Programa que demonstra componentes em um FlowLayout (parte 1 de 2).

```

62             // realinha componentes anexados
63             layout.layoutContainer( container );
64         }
65     }
66 }
67 );
68
69 container.add( centerButton );
70
71 // configura o rightButton e registra ouvinte
72 rightButton = new JButton( "Right" );
73
74 rightButton.addActionListener(
75
76     // classe interna anônima
77     new ActionListener() {
78
79         // processa evento do rightButton
80         public void actionPerformed( ActionEvent event )
81         {
82             layout.setAlignment( FlowLayout.RIGHT );
83
84             // realinha componentes anexados
85             layout.layoutContainer( container );
86         }
87     }
88 );
89
90 container.add( rightButton );
91
92 setSize( 300, 75 );
93 setVisible( true );
94 }
95
96 // executa o aplicativo
97 public static void main( String args[] )
98 {
99     FlowLayoutDemo application = new FlowLayoutDemo();
100
101 application.setDefaultCloseOperation(
102     JFrame.EXIT_ON_CLOSE );
103 }
104
105 } // fim da classe FlowLayoutDemo

```

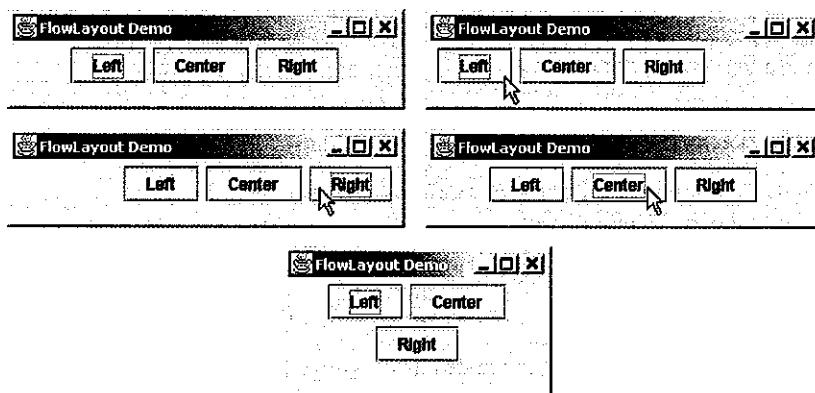


Fig. 12.24 Programa que demonstra componentes em um FlowLayout (parte 2 de 2).

Como visto antes, estabelece-se um leiaute de contêiner com o método `setLayout` da classe `Container`. A linha 25 configura o gerenciador de leiaute do painel de conteúdo como o `FlowLayout` definido na linha 21. Normalmente, o leiaute é configurado antes de quaisquer componentes GUI serem adicionados a um contêiner.



Observação de aparência e comportamento 12.10

Cada contêiner pode ter apenas um gerenciador de leiaute por vez (os contêineres separados no mesmo programa podem ter gerenciadores de leiaute diferentes).

O tratador de eventos `actionPerformed` de cada botão executa duas instruções. Por exemplo, a linha 38 no método `actionPerformed` para o botão `left` utiliza o método `setAlignment` de `FlowLayout` para alterar o alinhamento do `FlowLayout` para um `FlowLayout` alinhado à esquerda (`FlowLayout.LEFT`). A linha 41 utiliza o método `layoutContainer` da interface `LayoutManager` para especificar que o painel de conteúdo deve ser reorganizado com base no leiaute ajustado.

De acordo com qual botão se clica, o método `actionPerformed` para cada botão configura o alinhamento do `FlowLayout` como `FlowLayout.LEFT`, `FlowLayout.CENTER` ou `FlowLayout.RIGHT`.

12.14.2 BorderLayout

O gerenciador de leiaute `BorderLayout` (o gerenciador *default* de leiaute para o painel de conteúdo) organiza componentes em cinco regiões: `NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER` (`NORTH` corresponde à parte superior do contêiner). A classe `BorderLayout` herda de `Object` e implementa a interface `LayoutManager2` (uma subinterface de `LayoutManager` que adiciona vários métodos para processamento de leiaute aprimorado).

Até cinco componentes podem ser adicionados diretamente a um `BorderLayout` – um para cada região. Os componentes colocados em cada região podem ser um contêiner ao qual outros componentes são anexados. Os componentes colocados nas regiões `NORTH` e `SOUTH` estendem-se horizontalmente para os lados do contêiner e têm a mesma altura que o componente mais alto colocado nessas regiões. As regiões `EAST` e `WEST` expandem-se verticalmente entre as regiões `NORTH` e `SOUTH` e têm a mesma largura que o componente mais largo colocado nessas regiões. O componente colocado na região `CENTER` expande-se para ocupar todo o espaço restante no leiaute (essa é a razão pela qual `JTextArea` na Fig. 12.22 ocupa a janela inteira). Se todas as cinco regiões estiverem ocupadas, o espaço do contêiner inteiro é coberto por componentes GUI. Se a região `NORTH` ou `SOUTH` não for ocupada, os componentes GUI nas regiões `EAST`, `CENTER` e `WEST` expandem-se verticalmente para preencher o espaço restante. Se a região `EAST` ou `WEST` não for ocupada, o componente GUI na região `CENTER` expande-se horizontalmente para preencher o espaço restante. Se a região `CENTER` não for ocupada, a área é deixada vazia – os outros componentes GUI não se expandem para preencher o espaço restante.

O aplicativo da Fig. 12.25 demonstra o gerenciador de leiaute `BorderLayout` com cinco `JButtons`.

```

1 // Fig. 12.25: BorderLayoutDemo.java
2 // Demonstrando o BorderLayout.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class BorderLayoutDemo extends JFrame
12     implements ActionListener {
13
14     JButton buttons[];
15     private String names[] = { "Hide North", "Hide South",
16         "Hide East", "Hide West", "Hide Center" };

```

Fig. 12.25 Demonstrando componentes em `BorderLayout` (parte 1 de 3).

```

17     private BorderLayout layout;
18
19     // configura a GUI e o tratamento de eventos
20     public BorderLayoutDemo()
21     {
22         super( "BorderLayout Demo" );
23
24         layout = new BorderLayout( 5, 5 );
25
26         // obtém painel de conteúdo e configura o leiaute
27         Container container = getContentPane();
28         container.setLayout( layout );
29
30         // instancia objetos botão
31         buttons = new JButton[ names.length ];
32
33         for ( int count = 0; count < names.length; count++ ) {
34             buttons[ count ] = new JButton( names[ count ] );
35             buttons[ count ].addActionListener( this );
36         }
37
38         // coloca botões em BorderLayout; a ordem não é importante
39         container.add( buttons[ 0 ], BorderLayout.NORTH );
40         container.add( buttons[ 1 ], BorderLayout.SOUTH );
41         container.add( buttons[ 2 ], BorderLayout.EAST );
42         container.add( buttons[ 3 ], BorderLayout.WEST );
43         container.add( buttons[ 4 ], BorderLayout.CENTER );
44
45         setSize( 300, 200 );
46         setVisible( true );
47     }
48
49     // trata eventos de botões
50     public void actionPerformed( ActionEvent event )
51     {
52         for ( int count = 0; count < buttons.length; count++ )
53
54             if ( event.getSource() == buttons[ count ] )
55                 buttons[ count ].setVisible( false );
56             else
57                 buttons[ count ].setVisible( true );
58
59             // refaz leiaute do painel de conteúdo
60             layout.layoutContainer( getContentPane() );
61     }
62
63     // executa o aplicativo
64     public static void main( String args[] )
65     {
66         BorderLayoutDemo application = new BorderLayoutDemo();
67
68         application.setDefaultCloseOperation(
69             JFrame.EXIT_ON_CLOSE );
70     }
71
72 } // fim da classe BorderLayoutDemo

```

Fig. 12.25 Demonstrando componentes em BorderLayout (parte 2 de 3).

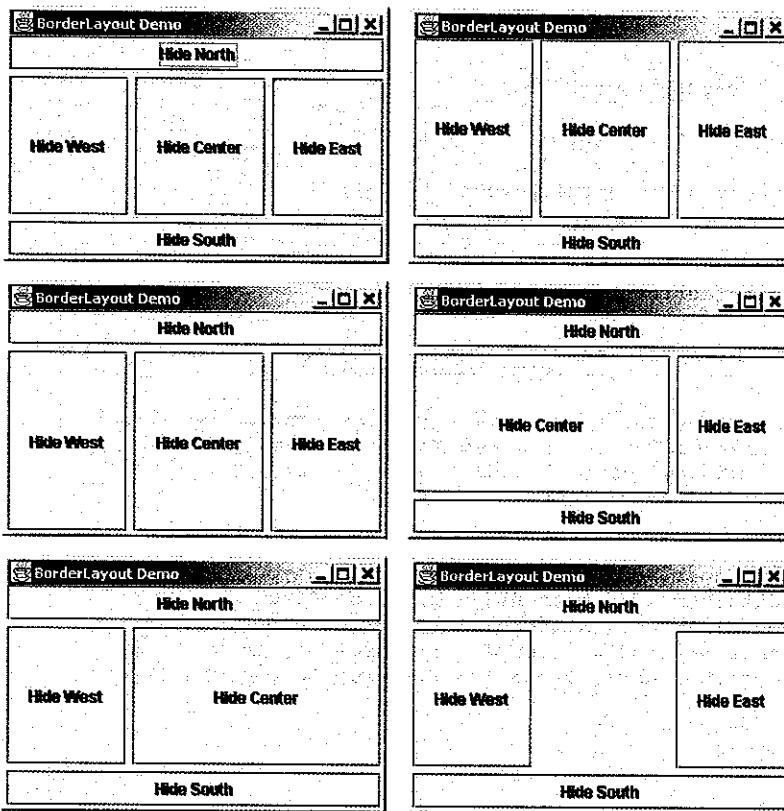


Fig. 12.25 Demonstrando componentes em BorderLayout (parte 3 de 3).

A linha 24 no construtor define um `BorderLayout`. Os argumentos especificam o número de *pixels* entre os componentes que são organizados horizontalmente (*espacamento horizontal*) e o número de *pixels* entre os componentes que são organizados verticalmente (*espacamento vertical*), respectivamente. O construtor *default* `BorderLayout` deixa 0 *pixels* de espaçamento horizontal e vertical. A linha 28 utiliza o método `setLayout` para configurar o leiaute do painel de conteúdo como `layout`.

Adicionar `Components` a um `BorderLayout` exige um método `add` diferente da classe `Container`, que recebe dois argumentos – o `Component` para adicionar e a região em que o `Component` será colocado. Por exemplo, a linha 39 especifica que o `buttons[0]` deve ser colocado na posição `NORTH`. Os componentes podem ser adicionados em qualquer ordem, mas apenas um componente pode ser adicionado a cada região.



Observação de aparência e comportamento 12.11

Se nenhuma região for especificada ao se adicionar um Component a um BorderLayout, supõe-se que o Component deve ser adicionado à região BorderLayout.CENTER.



Erro comum de programação 12.6

Adicionar mais de um componente a uma região particular em um BorderLayout faz com que apenas o último componente adicionado seja exibido. Não há mensagem de erro para indicar esse problema.

Quando o usuário clica em um `JButton` particular no leiaute, o método `actionPerformed` (linhas 50 a 61) é executado. O laço `for` nas linhas 52 a 57 utiliza a seguinte estrutura `if/else` para ocultar o `JButton` particular que gerou o evento. O método `setVisible` (herdado por `JButton` da classe `Component`) é chamado com um argumento `false` para ocultar o `JButton`. Se o `JButton` atual no array não é o que gerou o evento, o

método `setVisible` é chamado com um argumento `true` para assegurar que o `JButton` é exibido na tela. A linha 60 utiliza o método `layoutContainer` de `LayoutManager` para recalcular o leiaute do painel de conteúdo. Repare, nas capturas de tela da Fig. 12.25, que certas regiões no `BorderLayout` alteram a forma quando os `JButtons` são ocultados e exibidos em outras regiões. Tente redimensionar a janela de aplicativo para ver como as várias regiões se redimensionam de acordo com a largura e a altura da janela.

12.14.3 GridLayout

O gerenciador de leiaute `GridLayout` divide o contêiner em uma grade de modo que os componentes podem ser colocados nas linhas e colunas. A classe `GridLayout` herda diretamente da classe `Object` e implementa a interface `LayoutManager`. Cada `Component` em um `GridLayout` tem a mesma largura e altura. Os componentes são adicionados a um `GridLayout` que começa na célula da parte superior esquerda da grade e vai da esquerda para a direita até a linha estar cheia. Depois, o processo continua da esquerda para a direita na próxima linha da grade, etc. A Fig. 12.26 demonstra o gerenciador de leiaute `GridLayout` com seis `JButtons`.

```

1 // Fig. 12.26: GridLayoutDemo.java
2 // Demonstrando o GridLayout.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class GridLayoutDemo extends JFrame
12     implements ActionListener {
13
14     private JButton buttons[];
15     private String names[] =
16         { "one", "two", "three", "four", "five", "six" };
17     private boolean toggle = true;
18     private Container container;
19     private GridLayout grid1, grid2;
20
21     // configura a GUI
22     public GridLayoutDemo()
23     {
24         super( "GridLayout Demo" );
25
26         // configura leiautes
27         grid1 = new GridLayout( 2, 3, 5, 5 );
28         grid2 = new GridLayout( 3, 2 );
29
30         // obtém painel de conteúdo e configura o leiaute
31         container = getContentPane();
32         container.setLayout( grid1 );
33
34         // cria e adiciona botões
35         buttons = new JButton[ names.length ];
36
37         for ( int count = 0; count < names.length; count++ ) {
38             buttons[ count ] = new JButton( names[ count ] );
39             buttons[ count ].addActionListener( this );
40             container.add( buttons[ count ] );
41         }
42     }

```

Fig. 12.26 Programa que demonstra componentes em um `GridLayout` (parte 1 de 2).

```

43     setSize( 300, 150 );
44     setVisible( true );
45 }
46
47 // trata os eventos de botão chaveando entre os leiautes
48 public void actionPerformed( ActionEvent event )
49 {
50     if ( toggle )
51         container.setLayout( grid2 );
52     else
53         container.setLayout( grid1 );
54
55     toggle = !toggle; // configura o valor de toggle para o valor oposto
56     container.validate();
57 }
58
59 // executa o aplicativo
60 public static void main( String args[] )
61 {
62     GridLayoutDemo application = new GridLayoutDemo();
63
64     application.setDefaultCloseOperation(
65         JFrame.EXIT_ON_CLOSE );
66 }
67
68 } // fim da classe GridLayoutDemo

```

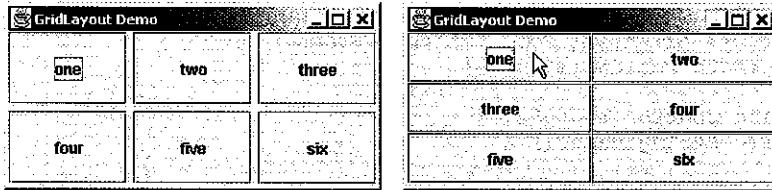


Fig. 12.26 Programa que demonstra componentes em um `GridLayout` (parte 2 de 2).

As linhas 27 e 28 no construtor definem dois objetos `GridLayout`. O construtor `GridLayout` utilizado na linha 27 especifica um `GridLayout` com 2 linhas, 3 colunas, 5 pixels de espaçamento horizontal entre os `Components` na grade e 5 pixels de espaçamento vertical entre `Components` na grade. O construtor `GridLayout` utilizado na linha 28 especifica um `GridLayout` com 3 linhas, 2 colunas e nenhum espaçamento.

Os objetos `JButton` nesse exemplo são inicialmente organizados com `grid1` (associado ao painel de conteúdo na linha 32 com o método `setLayout`). O primeiro componente é adicionado à primeira coluna da primeira linha. O próximo componente é adicionado à segunda coluna da primeira linha, etc. Quando um `JButton` é pressionado, o método `actionPerformed` (linhas 48 a 57) é chamado. Cada chamada para `actionPerformed` alterna o leiaute entre `grid2` e `grid1`.

A linha 56 ilustra outra maneira de reorganizar um contêiner cujo leiaute foi alterado. O método `validate` de `Container` recalcula o leiaute do contêiner com base no gerenciador de leiaute atual para o `Container` e o conjunto atual de componentes GUI exibidos.

12.15 Painéis

As GUIs complexas (como a Fig. 12.1) exigem que cada componente seja colocado em uma posição exata. Elas consistem, com freqüência, em múltiplos *painéis* com os componentes de cada painel organizados em um leiaute específico. Os painéis são criados com a classe `JPanel` – uma subclasse de `JComponent`. A classe `JComponent` herda da classe `java.awt.Container`, de modo que cada `JPanel` é um `Container`. Assim, `JPanels` podem ter componentes, incluindo outros painéis, adicionado a eles.

O programa da Fig. 12.27 demonstra como se pode utilizar um **JPanel** para criar um leiaute mais complexo para **Components**.

```

1 // Fig. 12.27: PanelDemo.java
2 // Usando um JPanel para ajudar a dispor o leiaute dos componentes.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class PanelDemo extends JFrame {
12     private JPanel buttonPanel;
13     private JButton buttons[];
14
15     // configura a GUI
16     public PanelDemo()
17     {
18         super( "Panel Demo" );
19
20         // obtém painel de conteúdo
21         Container container = getContentPane();
22
23         // cria array de botões
24         buttons = new JButton[ 5 ];
25
26         // configura o painel e estabelece o leiaute
27         buttonPanel = new JPanel();
28         buttonPanel.setLayout(
29             new GridLayout( 1, buttons.length ) );
30
31         // cria e adiciona botões
32         for ( int count = 0; count < buttons.length; count++ ) {
33             buttons[ count ] =
34                 new JButton( "Button " + ( count + 1 ) );
35             buttonPanel.add( buttons[ count ] );
36         }
37
38         container.add( buttonPanel, BorderLayout.SOUTH );
39
40         setSize( 425, 150 );
41         setVisible( true );
42     }
43
44     // executa o aplicativo
45     public static void main( String args[] )
46     {
47         PanelDemo application = new PanelDemo();
48
49         application.setDefaultCloseOperation(
50             JFrame.EXIT_ON_CLOSE );
51     }
52
53 } // fim da classe PanelDemo

```

Fig. 12.27 Um JPanel com cinco JButtons em um GridLayout anexado à região SOUTH de um BorderLayout (parte 1 de 2).

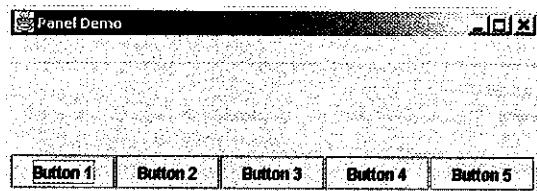


Fig. 12.27 Um JPanel com cinco JButtons em um GridLayout anexado à região SOUTH de um BorderLayout (parte 2 de 2).

Depois que o JPanel buttonPanel é criado na linha 27, as linhas 28 e 29 configuram o leiaute do buttonPanel como um GridLayout de uma linha e cinco colunas (há cinco JButtons no array buttons). Os cinco JButtons no array buttons são adicionados ao JPanel no laço, com a linha 35. Repare que os botões são adicionados diretamente ao JPanel – a classe JPanel não tem um painel de conteúdo como applet nem como JFrame. A linha 38 utiliza o leiaute default de painel de conteúdo, BorderLayout, para adicionar buttonPanel à região SOUTH. Observe que a região SOUTH é tão alta quanto os botões no buttonPanel. O JPanel é dimensionado de acordo com os componentes que ele contém. À medida que mais componentes são adicionados, o JPanel cresce (de acordo com as restrições de seu gerenciador de leiaute) para acomodar os componentes. Redimensione a janela para ver como o gerenciador de leiaute afeta o tamanho dos JButtons.

12.16 (Estudo de caso opcional) Pensando em objetos: casos de uso

As oito seções “Pensando em objetos” anteriores se concentraram no modelo de simulação do elevador. Identificamos e afiamos a estrutura e o comportamento de nosso sistema. Nessa seção, modelamos a interação entre o usuário e nossa simulação de elevador através do diagrama de casos de uso da UML, que descreve o conjunto de cenários que ocorrem entre o sistema e o usuário.

Diagramas de caso de uso

Quando os desenvolvedores iniciam um projeto, eles raramente começam com uma definição detalhada do problema, como a que fornecemos na Seção 2.9. Este documento e outros são o resultado da fase de *análise orientada a objetos* (OOA – object-oriented analysis). Nesta fase, você entrevista as pessoas que querem que você construa um sistema e as pessoas que vão em algum momento usar o sistema. Você usa a informação obtida nestas reuniões para compilar uma lista de *requisitos do sistema*. Estes requisitos guiam-no e seus companheiros desenvolvedores enquanto vocês projetam o sistema. Em nosso estudo de caso, a definição do problema descreveu os requisitos de nossa simulação de elevador em detalhes suficientes para que você não precisasse passar por uma fase de análise. A fase de análise é extremamente importante – você deve consultar as referências que fornecemos na Seção 2.9 para aprender mais sobre a análise orientada a objetos.

A UML oferece o *diagrama de casos de uso* para facilitar o processo de se reunir os requisitos. Esse diagrama modela as interações entre os clientes externos do sistema e os *casos de uso* do sistema. Cada caso de uso representa um recurso diferente que o sistema oferece aos clientes. Por exemplo, as máquinas de caixa automático têm diversos casos de uso, incluindo “depositar dinheiro”, “retirar dinheiro” e “transferência de fundos”.

Nos sistemas maiores, os diagramas de caso de uso são ferramentas indispensáveis que ajudam os projetistas do sistema a manter o foco na satisfação das necessidades dos usuários. O objetivo do diagrama de caso de uso é mostrar os tipos de interações que os usuários têm com um sistema, sem fornecer os detalhes daquelas interações: tais detalhes são, é claro, fornecidos em outros diagramas da UML.

A Fig. 12.28 mostra o diagrama de caso de uso para nossa simulação de elevador. A figura estilizada representa um *ator*, o qual, por sua vez, representa um conjunto de papéis que uma *entidade externa* – como uma pessoa ou outro sistema – pode desempenhar. Considere novamente nosso exemplo da máquina de caixa automático (ATM). O ator é um *ClienteDoBanco* que pode depositar, retirar e transferir fundos a partir da ATM. Neste sentido, *ClienteDoBanco* é mais parecido com uma classe do que com um objeto – ele não é uma pessoa de verdade, mas descreve os papéis que uma pessoa real – quando age como *ClienteDoBanco* – pode desempenhar enquanto interage com a ATM (depositar, retirar e transferir fundos). Uma pessoa é uma entidade externa que pode desempe-

nhar o papel de um **ClienteDoBanco**. Da mesma maneira que um objeto é uma instância de uma classe, a pessoa que desempenha o papel de um **ClienteDoBanco** executando um de seus papéis (como fazer um depósito) é uma instância do ator **ClienteDoBanco**. Por exemplo, quando uma pessoa chamada Maria age como **ClienteDoBanco** ao fazer um depósito, Maria – no papel do depositante – se torna uma instância do ator **ClienteDoBanco**. Mais tarde neste dia, outra pessoa chamada João pode ser uma outra instância do ator **ClienteDoBanco**. Ao longo do dia, diversas centenas de pessoas poderiam usar a máquina ATM – algumas são “depositantes”, algumas são “sacadores” e algumas são “transferidores”, mas todas elas são instâncias do ator **ClienteDoBanco**.

A definição do problema em nossa simulação de elevador fornece os atores – “O usuário exige a habilidade de criar uma pessoa na simulação e situar aquela pessoa em um dado andar”. Portanto, o ator de nosso sistema é o usuário que controla a simulação (isto é, o usuário que clica nos botões para criar novas **Persons** na simulação). Uma entidade externa – uma pessoa real – age como usuário para controlar a simulação. Em nosso sistema, o caso de uso é “Criar pessoa”, que engloba criar um objeto **Person** e depois colocar aquela **Person** no primeiro ou no segundo **Floor**. A Fig. 12.28 modela um ator chamado “Usuário”. O nome do ator aparece sob o ator.

A *caixa do sistema* (isto é, o retângulo de fechamento na figura) contém os casos de uso para o sistema. Repare que a caixa é rotulada “Simulação de elevador”. Este título mostra que este modelo de caso de uso se concentra no único caso de uso que nossa simulação fornece para os usuários (isto é, “Criar pessoa”). A UML modela cada caso de uso como uma elipse. A caixa do sistema para um sistema com múltiplos casos de uso teria uma elipse por caso de uso.

Existe uma visão alternativa razoável do caso de uso de nossa simulação de elevador. A definição do problema na Seção 2.9 mencionou que a empresa solicitou a simulação do elevador para “determinar se o elevador atenderá às necessidades da empresa”. Estamos projetando uma simulação de um cenário do mundo real – o objeto **Person** na simulação representa um ser humano de verdade usando um elevador de verdade. Assim, podemos encarar o usuário da simulação do elevador como usuário do elevador. Portanto, especificar um caso de uso da perspectiva do objeto **Person** ajuda a modelar como uma pessoa real usa um sistema de elevador real. Oferecemos o caso de uso da Fig. 12.29, denominado “Deslocar pessoa”. Este caso de uso descreve a **Person** se movendo (deslocando-se) para o outro **Floor** (a **Person** viaja para o segundo **Floor** se estiver iniciando no primeiro **Floor** e para o primeiro **Floor** se estiver iniciando no segundo **Floor**). Este caso de uso abrange todas as ações que a **Person** executa ao longo de sua jornada, como caminhar através de um **Floor** para o **Elevator**, pressionar **Buttons** e andar no **Elevator** até o outro **Floor**.

Precisamos assegurar que nossos casos de uso não modelam interações entre o cliente externo e o sistema que sejam específicas demais. Por exemplo, não subdividimos cada caso de uso em dois casos de uso separados – como



Fig. 12.28 Diagrama de caso de uso para a simulação de elevador da perspectiva do usuário.

“Criar pessoa no primeiro andar” e “Criar pessoa no segundo andar”, ou “Deslocar pessoa para o primeiro andar” e “Deslocar pessoa para o segundo andar” – porque a funcionalidade de tais casos de uso é repetitiva (isto é, estes casos de uso aparentemente alternativos são na realidade um só). A divisão imprópria e repetitiva de casos de uso pode criar problemas durante a implementação. Por exemplo, se o projetista de uma máquina de caixa automática separasse seu caso de uso “Retirar dinheiro” em casos de uso “Retirar quantias específicas” (por exemplo, “Retirar R\$ 1,00”, “Retirar R\$ 2,00”, etc.), poderia existir uma quantidade enorme de casos de uso para o sistema. Isto tornaria a implementação tediosa (nossa sistema de elevador contém apenas dois andares – separar o caso de uso em dois não causaria tanto trabalho extra; se nosso sistema contivesse 100 andares, no entanto, criar 100 casos de uso seria algo difícil de manejar).

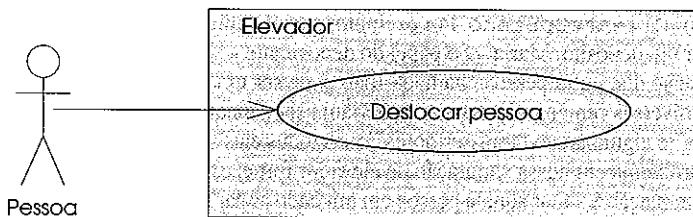


Fig. 12.29 Diagrama de caso de uso para a simulação de elevador da perspectiva de uma pessoa.

Construindo a interface gráfica com o usuário

Nossa simulação implementa tanto o caso de uso “Criar pessoa” como “Deslocar pessoa”. Estudamos o caso de uso “Deslocar pessoa” através do diagrama de atividades da **Person** na Fig. 5.29 – implementamos este caso de uso e o diagrama de atividades no Apêndice H quando criamos a classe **Person**. Implementamos o caso de uso “Criar pessoa” através de uma interface gráfica com o usuário (GUI). Implementamos nossa GUI na classe **ElevatorController** (Fig. 12.30, linha 17), que é uma subclasse **JPanel** que contém dois objetos **JButton** – **firstControllerButton** (linha 21) e **secondControllerButton** (linha 22). Cada **JButton** corresponde a um **Floor** no qual colocar uma **Person**¹. As linhas 33 a 38 instanciam estes **JButtons** e os adicionam ao **ElevatorController**.

Discutimos na Seção 13.17 como a classe **ElevatorModel** congrega todos os objetos que compõem nosso modelo de simulação de elevador. A linha 25 da classe **ElevatorController** declara uma referência para o **ElevatorModel**, porque o **ElevatorController** permite ao usuário interagir com o modelo. As linhas 42 a 56 e 60 a 74 declaram dois objetos anônimos **ActionListener** e os registram com **firstControllerButton** e **secondControllerButton**, respectivamente, para **ActionEvents**. Quando o usuário pressiona qualquer um dos **JButtons**, as linhas 49 e 50 e 67 e 68 dos métodos **actionPerformed** chamam o método **addPerson** de **ElevatorModel**, que instancia um objeto **Person** no **ElevatorModel** no **Floor** especificado. O método **addPerson** recebe como argumento um **String** definido na interface **ElevatorConstants** (Fig. 12.31). Esta interface – usada por classes como **ElevatorController**, **ElevatorModel**, **Elevator**, **Floor** e **ElevatorView** – fornece constantes que especificam os nomes das **Locations** em nossa simulação.

As linhas 53 e 71 dos métodos **actionPerformed** inibem os respectivos **JButtons** para evitar que o usuário crie mais de uma **Person** por **Floor**. As linhas 78 a 116 da classe **ElevatorController** declaram um **PersonMoveListener** anônimo que é registrado com o **ElevatorModel** para reabilitar os **JButtons**. O método **personEntered** (linhas 82 a 97) do **PersonMoveListener** reabilita o **JButton** associado com o

```

1 // ElevatorController.java
2 // Controlador para a simulação do elevador.
3 package com.deitel.jhttp4.elevator.controller;
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 // Pacotes Deitel
13 import com.deitel.jhttp4.elevator.model.*;

```

Fig. 12.30 Classe **ElevatorController** processa dados de entrada do usuário (parte 1 de 3).

¹ Esta abordagem é viável com apenas dois andares. Se o edifício tivesse 100 andares, poderíamos ter feito o usuário especificar o andar desejado em um **JTextField** e pressionar um **JButton** para processar o pedido.

```

14 import com.deitel.jhttp4.elevator.event.*;
15 import com.deitel.jhttp4.elevator.ElevatorConstants;
16
17 public class ElevatorController extends JPanel
18     implements ElevatorConstants {
19
20     // controlador contém dois JButtons
21     private JButton firstControllerButton;
22     private JButton secondControllerButton;
23
24     // referência para o modelo
25     private ElevatorModel elevatorModel;
26
27     public ElevatorController( ElevatorModel model )
28     {
29         elevatorModel = model;
30         setBackground( Color.white );
31
32         // adiciona primeiro botão ao controlador
33         firstControllerButton = new JButton( "First Floor" );
34         add( firstControllerButton );
35
36         // adiciona segundo botão ao controlador
37         secondControllerButton = new JButton( "Second Floor" );
38         add( secondControllerButton );
39
40         // registros da classe interna anônima para receber ActionEvents
41         // do JButton firstController
42         firstControllerButton.addActionListener(
43             new ActionListener() {
44
45                 // invocado quando um JButton foi pressionado
46                 public void actionPerformed( ActionEvent event )
47                 {
48                     // coloca a Person no primeiro Floor
49                     elevatorModel.addPerson(
50                         FIRST_FLOOR_NAME );
51
52                     // inibe entrada de dados do usuário
53                     firstControllerButton.setEnabled( false );
54                 }
55             } // fim da classe interna anônima
56         );
57
58         // registros da classe interna anônima para receber ActionEvents
59         // do JButton secondController
60         secondControllerButton.addActionListener(
61             new ActionListener() {
62
63                 // invocado quando um JButton foi pressionado
64                 public void actionPerformed( ActionEvent event )
65                 {
66                     // coloca a Person no segundo Floor
67                     elevatorModel.addPerson(
68                         SECOND_FLOOR_NAME );
69
70                     // inibe entrada de dados do usuário
71                     secondControllerButton.setEnabled( false );
72                 }
73             } // fim da classe interna anônima
74         );

```

Fig. 12.30 Classe `ElevatorController` processa dados de entrada do usuário (parte 2 de 3).

```

74      );
75
76      // classe interna anônima permite que o usuário informe
77      // o Floor se a Person entrar no Elevator naquele Floor
78      elevatorModel.addPersonMoveListener(
79          new PersonMoveListener() {
80
81              // invocado quando a Person entrou no Elevator
82              public void personEntered(
83                  PersonMoveEvent event )
84              {
85                  // obtém Floor de partida
86                  String location =
87                      event.getLocation().getLocationName();
88
89                  // habilita o primeiro JButton se é do primeiro Floor
90                  if ( location.equals( FIRST_FLOOR_NAME ) )
91                      firstControllerButton.setEnabled( true );
92
93                  // habilita o segundo JButton se é do segundo Floor
94                  else
95                      secondControllerButton.setEnabled( true );
96
97              } // fim do método personEntered
98
99              // outros métodos que implementam PersonMoveListener
100             public void personCreated(
101                 PersonMoveEvent event ) {}
102
103             public void personArrived(
104                 PersonMoveEvent event ) {}
105
106             public void personExited(
107                 PersonMoveEvent event ) {}
108
109             public void personDeparted(
110                 PersonMoveEvent event ) {}
111
112             public void personPressedButton(
113                 PersonMoveEvent event ) {}
114
115         } // fim da classe interna anônima
116     );
117 } // fim do construtor ElevatorController
118 }
```

Fig. 12.30 Classe `ElevatorController` processa dados de entrada do usuário (parte 3 de 3).

`Floor` que o `Elevator` está atendendo – depois que a `Person` entrou no `Elevator`, o usuário pode colocar outra `Person` no `Floor`.

Mencionamos na Seção 9.23 que as classes `Elevator` e `Floor` herdaram o atributo `capacity` da superclasse `Location` – no Apêndice H, iríamos usar este atributo para evitar que mais de uma `Person` ocupasse uma `Location`. Entretanto, o método `personEntered` do `PersonMoveListener` na classe `ElevatorController` impede o usuário de criar mais de uma `Person` por `Floor`. Portanto, negamos a necessidade do atributo `capacity` na classe `Location`. A Fig. 12.32 é o diagrama de classes da Fig. 9.18 modificado para remover este atributo.

Nesta seção, mencionamos que o objetivo da análise orientada a objetos é produzir um documento de requisitos do sistema. Apresentamos o diagrama de casos de uso da UML que deixa mais fácil coletar os requisitos do sis-

tema e examinamos os dois casos de uso em nossa simulação de elevador. Implementamos a interface gráfica com o usuário de nosso simulador em Java.

```

1 // ElevatorConstants.java
2 // Constantes usadas entre ElevatorModel e ElevatorView
3 package com.deitel.jhtp4.elevator;
4
5 public interface ElevatorConstants {
6
7     public static final String FIRST_FLOOR_NAME = "firstFloor";
8     public static final String SECOND_FLOOR_NAME = "secondFloor";
9     public static final String ELEVATOR_NAME = "elevator";
10 }
```

Fig. 12.31 Interface `ElevatorConstants` fornece constantes com nome das `Locations`.

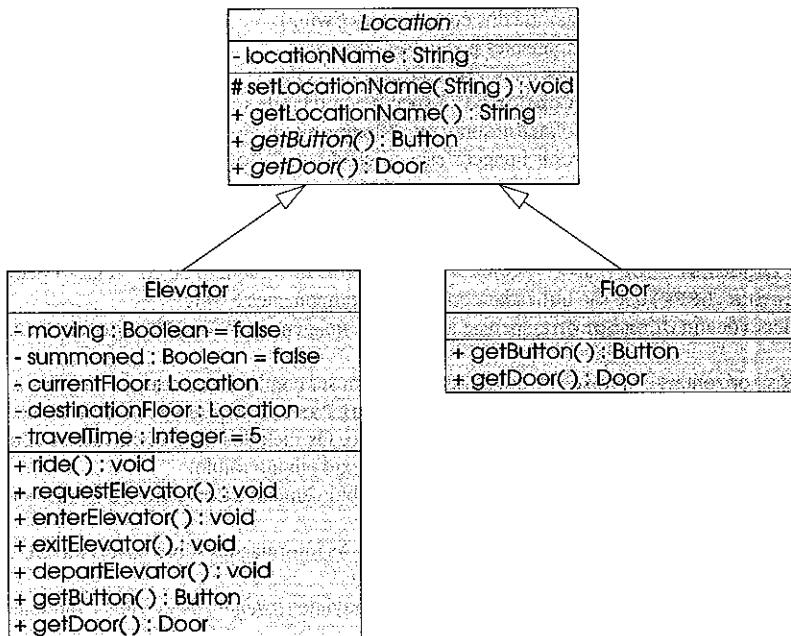


Fig. 12.32 Diagrama de classes modificado mostrando a generalização da superclasse `Location` e das subclasses `Elevator` e `Floor`.

Esta seção termina a discussão sobre a interação entre o usuário e o modelo de simulação. Na Seção 13.17 de “Pensando em objetos”, integramos a classe `ElevatorController` com o resto da simulação. Também apresentamos o diagrama de componentes da UML, que modela os arquivos `.class`, `.java`, de imagem e de som que compreendem nosso sistema.

Resumo

- A interface gráfica com o usuário (GUI) apresenta uma interface pictórica para um programa. Ela fornece a um programa uma “aparência” e um “comportamento” diferenciados.
- Fornecendo a diferentes aplicativos um conjunto consistente de componentes intuitivos de interface com o usuário, as GUIs permitem ao usuário gastar mais tempo utilizando o programa de uma maneira produtiva.
- As GUIs são construídas a partir de componentes GUI (às vezes chamados de controles ou *widgets*). O componente GUI é um objeto visual com o qual o usuário interage através do mouse ou do teclado.
- Os componentes GUI Swing são definidos no pacote `javax.swing`. Os componentes Swing são escritos, manipulados e exibidos completamente em Java.
- Os componentes GUI originais do pacote `java.awt` do Abstract Windowing Toolkit estão diretamente associados com as capacidades da interface gráfica com o usuário da plataforma local.
- Os componentes Swing são componentes peso-leve. Os componentes AWT estão vinculados à plataforma local e são chamados de componentes peso-pesado – eles devem contar com o sistema de janelas da plataforma local para determinar sua funcionalidade e sua aparência e comportamento.
- Vários componentes GUI Swing são componentes GUI peso-pesado: em particular, as subclasses de `java.awt.Window` (como `JFrame`) que exibem janelas na tela. Os componentes GUI Swing peso-pesado são menos flexíveis que os componentes peso-leve.
- Grande parte da funcionalidade de cada componente GUI Swing é herdada das classes `Component`, `Container` e `JComponent` (a superclasse para a maioria dos componentes Swing).
- O contêiner é uma área em que os componentes podem ser colocados.
- `JLabels` fornecem texto com instruções ou informações em uma GUI.
- O método `setToolTipText` de `JComponent` especifica a dica de ferramenta que é exibida automaticamente quando o usuário posiciona o cursor do mouse sobre um `JComponent` na GUI.
- Muitos componentes Swing podem exibir imagens especificando um `Icon` como argumento para seu construtor ou utilizando um método `setIcon`.
- A classe `ImageIcon` (pacote `javax.swing`) suporta diversos formatos de imagem, incluindo Portable Network Graphics (PNG), Graphics Interchange Format (GIF) e Joint Photographic Experts Group (JPEG).
- A interface `SwingConstants` (pacote `javax.swing`) define um conjunto de constantes inteiras comuns (como `SwingConstants.LEFT`) que são utilizadas com muitos componentes Swing.
- Por *default*, o texto de um `JComponent` aparece à direita da imagem quando o `JComponent` contém texto e imagem.
- Os alinhamentos horizontais e verticais de um `JLabel` podem ser configurados com os métodos `setHorizontalAlignment` e `setVerticalAlignment`. O método `setText` configura o texto exibido no rótulo. O método `getText` recupera o texto atualmente exibido em um rótulo. Os métodos `setHorizontalTextPosition` e `setVerticalTextPosition` especificam a posição do texto em um rótulo.
- O método `setIcon` de `JComponent` configura o `Icon` exibido em um `JComponent`. O método `getIcon` recupera o `Icon` atual exibido em um `JComponent`.
- As GUIs geram eventos quando o usuário interage com a GUI. As informações sobre um evento GUI são armazenadas em um objeto de uma classe que estende `AWTEvent`.
- Para processar um evento, o programador deve registrar um ouvinte (*listener*) de eventos e implementar um ou mais tratadores de eventos.
- O uso de *ouvintes* de eventos no tratamento de eventos é conhecido como o modelo de delegação de eventos – o processamento de um evento é delegado para um objeto particular no programa.
- Quando ocorre um evento, o componente GUI com que o usuário interagiu notifica seus ouvintes registrados chamando o método de tratamento de evento apropriado de cada ouvinte.
- `JTextFields` e `JPasswordFields` são áreas de uma única linha em que o usuário pode inserir texto a partir do teclado ou o texto pode ser simplesmente exibido. O `JPasswordField` mostra que um caractere foi digitado quando o usuário insere os caracteres, mas oculta automaticamente os caracteres.
- Quando o usuário digita os dados em um `JTextField` ou `JPasswordField` e pressiona a tecla *Enter*, ocorre um `ActionEvent`.
- O método `setEditable` de `JTextComponent` determina se o usuário pode modificar o texto em um `JTextComponent`.
- O método `getPassword` de `JPasswordField` devolve a senha como um *array* de tipo `char`.
- Cada `JComponent` contém um objeto da classe `EventListenerList` (pacote `javax.swing.event`) chamado `listenerList` em que todos os ouvintes registrados são armazenados.

- Cada **JComponent** suporta vários tipos diferentes de eventos, incluindo eventos de mouse, eventos de teclas e outros. Quando ocorre um evento, o evento é despachado apenas para os ouvintes de evento do tipo apropriado. Cada tipo de evento tem uma interface *listener* de eventos correspondente.
- Quando um evento é gerado por uma interação do usuário com um componente, o componente recebe um ID de evento exclusivo especificando o tipo de evento. O componente GUI utiliza o ID de evento para decidir o tipo de ouvinte para o qual o evento deve ser despachado e o método de tratamento do evento a ser chamado.
- O **JButton** gera um **ActionEvent** quando o usuário clica no botão do mouse.
- O **AbstractButton** pode ter um **Icon rollover** que é exibido quando o mouse é posicionado sobre o botão. O ícone muda quando o mouse é movido para dentro e para fora da área do botão na tela. O método **setRolloverIcon** de **AbstractButton** especifica a imagem exibida em um botão quando o usuário posiciona o mouse sobre o botão.
- Os componentes GUI Swing contêm três tipos de estado de botão – **JToggleButton**, **JCheckBox** e **JRadioButton** – que têm valores ativados/desativados ou verdadeiro/falso. As classes **JCheckBox** e **JRadioButton** são subclasses de **JToggleButton**.
- Quando o usuário clica em um **JCheckBox**, gera-se um **ItemEvent** que pode ser tratado por um **ItemListener**. **ItemListeners** devem definir o método **itemStateChanged**. O método **getStateChanged** de **ItemEvent** determina o estado de um **JToggleButton**.
- **JRadioButtons** são semelhantes a **JCheckboxes** no sentido de que têm dois estados – selecionado e não-selecionado. **JRadioButtons** normalmente aparecem como um grupo em que apenas um botão de opção pode ser selecionado de cada vez.
- O relacionamento lógico entre botões de opção é mantido por um objeto **ButtonGroup**.
- O construtor **JRadioButton** fornece o rótulo que aparece à direita do **JRadioButton** por *default* e o estado inicial do **JRadioButton**. Um segundo argumento **true** indica que o **JRadioButton** deve aparecer selecionado quando exibido.
- **JRadioButtons** geram **ItemEvents** quando se clica neles.
- O método **add** de **ButtonGroup** associa um **JRadioButton** com um **ButtonGroup**. Se mais de um objeto **JRadioButton** selecionado for adicionado ao grupo, o último **JRadioButton** adicionado estará selecionado quando a GUI for exibida.
- A **JComboBox** (às vezes chamada de caixa de combinação ou lista escamoteável) fornece uma lista de itens a partir da qual o usuário pode fazer uma seleção. **JComboBoxes** geram **ItemEvents**. Um índice numérico monitora o ordenamento dos itens em uma **JComboBox**. O primeiro item é adicionado no índice 0; o próximo item é adicionado no índice 1, e assim por diante. O primeiro item adicionado a uma **JComboBox** aparece como o item atualmente selecionado quando a **JComboBox** é exibida. O método **getSelectedIndex** de **JComboBox** devolve o número de índice do item selecionado.
- A **JList** exibe uma série de itens da qual o usuário pode selecionar um ou mais itens. A classe **JList** suporta listas de seleção única e múltipla. Quando se clica num item em uma **JList**, ocorre um **ListSelectionEvent**.
- O método **setVisibleRowCount** de **JList** determina o número de itens que são visíveis na lista. O método **setSelectionMode** especifica o modo de seleção para a lista.
- A classe **JList** não fornece automaticamente uma barra de rolagem se houver mais itens na lista que o número de linhas visíveis. Usa-se um objeto **JScrollPane** para fornecer a capacidade de rolagem automática para uma **JList**.
- A lista **SINGLE_INTERVAL_SELECTION** permite a seleção de um intervalo contíguo de itens clicando-se no primeiro item e então mantendo-se a tecla **Shift** pressionada enquanto se clica no último item do intervalo a ser selecionado.
- A lista **MULTIPLE_INTERVAL_SELECTION** permite seleção em um intervalo contínuo, como descrito para uma lista **SINGLE_INTERVAL_SELECTION**, e permite que diversos itens sejam selecionados mantendo-se a tecla **Ctrl** pressionada enquanto se clica em cada item para selecionar.
- O método **setFixedCellHeight** de **JList** especifica a altura em *pixels* de cada item em uma **JList**. O método **setFixedCellWidth** configura a largura em *pixels* de uma **JList**.
- Normalmente, um evento gerado por outro componente GUI (conhecido como evento externo) especifica quando devem ser processadas as seleções múltiplas em uma **JList**.
- O método **setListData** de **JList** configura os itens exibidos em uma **JList**. O método **getSelectedValues** devolve os itens selecionados como um *array* de **Objects**.
- Os eventos de mouse podem ser capturados por qualquer componente GUI que derive de **java.awt.Component** utilizando **MouseListeners** e **MouseMotionListeners**.
- Cada método de tratamento de eventos de mouse aceita como argumento um objeto **MouseEvent** que contém as informações sobre o evento de mouse e a posição em que o evento ocorreu.
- Os métodos **addMouseListener** e **addMouseMotionListener** são métodos **Component** utilizados para registrar ouvintes de eventos de mouse para um objeto de qualquer classe que estende **Component**.

- Muitas das interfaces *listeners* de eventos fornecem múltiplos métodos. Para cada uma, há uma classe adaptadora de ouvinte de eventos correspondente que fornece uma implementação *default* de cada método da interface. O programador pode estender a classe adaptadora para herdar a implementação *default* de cada método e simplesmente sobrepor o método ou os métodos necessários para o tratamento de eventos no programa.
- O método `getClickCount` de `MouseEvent` retorna o número de cliques do mouse.
- Os métodos `isMetaDown` e `isAltDown` de `InputEvent` são utilizados para determinar em qual botão do mouse o usuário clicou.
- `KeyListeners` tratam eventos de teclas que são gerados quando as teclas são pressionadas e liberadas. O `KeyListener` deve fornecer definições para os métodos `keyPressed`, `keyReleased` e `keyTyped`, e cada um deles recebe um `KeyEvent` como argumento.
- O método `keyPressed` é chamado em resposta ao pressionamento de qualquer tecla. O método `keyTyped` é chamado em resposta ao pressionamento de qualquer tecla que não seja uma tecla de ação (isto é, uma tecla de seta, *Home*, *End*, *Page Up*, *Page Down*, uma tecla de função, *Num Lock*, *Print Screen*, *Scroll Lock*, *Caps Lock* e *Pause*). O método `keyReleased` é chamado quando a tecla é liberada depois de qualquer evento `keyPressed` ou `keyTyped`.
- O método `getKeyCode` de `KeyEvent` obtém o código de tecla virtual da tecla que foi pressionada. A classe `KeyEvent` mantém um conjunto de constantes de códigos de tecla virtual que representa cada tecla no teclado.
- O método `getKeyText` de `KeyEvent` devolve um `String` que contém o nome da tecla que corresponde ao seu argumento de código de tecla virtual. O método `getKeyChar` obtém o valor Unicode do caractere digitado. O método `isActionKey` determina se a tecla do evento era uma tecla de ação.
- O método `getModifiers` de `InputEvent` determina se quaisquer teclas modificadoras (como *Shift*, *Alt* e *Ctrl*) estavam pressionadas quando ocorreu o evento de tecla. O método `getKeyModifiersText` de `KeyEvent` produz um `string` que contém os nomes das teclas modificadoras pressionadas.
- Os gerenciadores de leiaute organizam componentes GUI em um contêiner para fins de apresentação.
- `FlowLayout` dispõe os componentes da esquerda para a direita, na ordem em que são adicionados ao contêiner. Quando a borda do contêiner é alcançada, os componentes continuam na próxima linha.
- O método `setAlignment` de `FlowLayout` altera o alinhamento do `FlowLayout` para `FlowLayout.LEFT`, `FlowLayout.CENTER` ou `FlowLayout.RIGHT`.
- O gerenciador de leiaute `BorderLayout` organiza os componentes em cinco regiões: norte, sul, leste, oeste e centro. Pode-se adicionar um componente a cada região.
- O método `layoutContainer` de `LayoutManager` recalcula o leiaute de seu argumento `Container`.
- O gerenciador de leiaute `GridLayout` divide o contêiner em uma grade de linhas e colunas. Os componentes são adicionados a um `GridLayout` que começa na parte superior esquerda da célula e vai da esquerda para a direita, até que a linha esteja completa. Depois, o processo continua da esquerda para a direita na próxima linha da grade, e assim por diante.
- O método `validate` de `Container` recalcula o leiaute do contêiner com base no gerenciador de leiaute atual para o `Container` e para o conjunto atual de componentes GUI exibidos.
- Os painéis são criados com a classe `JPanel`, que herda da classe `JComponent`. `JPanels` podem ter componentes, inclusive outros painéis, adicionado a eles.

Terminologia

<i>Abstract Windowing Toolkit</i>	
<i>alinhado à direita</i>	<i>botão de comando</i>
<i>alinhado à esquerda</i>	<i>botão de opção</i>
<i>aparência e comportamento</i>	<i>caixa de marcação</i>
<i>aparência e comportamento plugável</i>	<i>caixa de rolagem</i>
<i>arrastar</i>	<i>centralizado</i>
<i>barra de ferramentas</i>	<i>classe AbstractButton</i>
<i>barra de menus</i>	<i>classe ActionEvent</i>
<i>barra de rolagem</i>	<i>classe adaptadora</i>
<i>baseado em evento</i>	<i>classe BorderLayout</i>
<i>BorderLayout.CENTER</i>	<i>classe ButtonGroup</i>
<i>BorderLayout.EAST</i>	<i>classe Component</i>
<i>BorderLayout.NORTH</i>	<i>classe ComponentAdapter</i>
<i>BorderLayout.SOUTH</i>	<i>classe Container</i>
<i>BorderLayout.WEST</i>	<i>classe ContainerAdapter</i>
<i>botão</i>	<i>classe EventListenerList</i>
	<i>classe EventObject</i>

<i>classe FlowLayout</i>	<i>interface ListSelectionModel</i>
<i>classe FocusAdapter</i>	<i>interface MouseListener</i>
<i>classe GridLayout</i>	<i>interface MouseMotionListener</i>
<i>classe ImageIcon</i>	<i>interface SwingConstants</i>
<i>classe InputEvent</i>	<i>interface WindowListener</i>
<i>classe ItemEvent</i>	<i>ItemEvent.DISELECTED</i>
<i>classe JButton</i>	<i>ItemEvent.SELECTED</i>
<i>classe JCheckBox</i>	<i>janeta</i>
<i>classe JComboBox</i>	<i>Joint Photographic Experts Group (JPEG)</i>
<i>classe JComponent</i>	<i>lista de seleção múltipla</i>
<i>classe JLabel</i>	<i>lista de uma única seleção</i>
<i>classe JList</i>	<i>lista escamoteável</i>
<i>classe JPanel</i>	<i>localização de interface com o usuário</i>
<i>classe JPasswordField</i>	<i>menu</i>
<i>classe JRadioButton</i>	<i>método actionPerformed</i>
<i>classe JScrollPane</i>	<i>método add da classe Container</i>
<i>classe JTextComponent</i>	<i>método add de ButtonGroup</i>
<i>classe JTextField</i>	<i>método addItemListener</i>
<i>classe JToggleButton</i>	<i>método addKeyListener</i>
<i>classe KeyAdapter</i>	<i>método addListSelectionListener</i>
<i>classe KeyEvent</i>	<i>método addMouseListener</i>
<i>classe ListSelectionEvent</i>	<i>método addMouseMotionListener</i>
<i>classe MouseAdapter</i>	<i>método getActionCommand</i>
<i>classe MouseEvent</i>	<i>método getClickCount</i>
<i>classe MouseMotionAdapter</i>	<i>método getIcon</i>
<i>classe WindowAdapter</i>	<i>método getKeyChar de KeyEvent</i>
<i>componente GUI</i>	<i>método getKeyCode de KeyEvent</i>
<i>componente peso-leve</i>	<i>método getKeyModifiersText</i>
<i>componente peso-pesado</i>	<i>método getKeyText de KeyEvent</i>
<i>controle</i>	<i>método getModifiers de InputEvent</i>
<i>despachar um evento</i>	<i>método getPassword de JPasswordField</i>
<i>dicas de ferramenta</i>	<i>método getSelectedIndex de JComboBox</i>
<i>espacamento horizontal</i>	<i>método getSelectedIndex de JList</i>
<i>espacamento vertical</i>	<i>método getSelectedValues de JList</i>
<i>"esperar" um evento</i>	<i>método getSource de ActionEvent</i>
<i>evento</i>	<i>método getStateChange de ItemEvent</i>
<i>extensão de nome de arquivo .gif</i>	<i>método getText de JLabel</i>
<i>extensão de nome de arquivo .jpg</i>	<i>método getX de MouseEvent</i>
<i>FlowLayout.CENTER</i>	<i>método getY de MouseEvent</i>
<i>FlowLayout.LEFT</i>	<i>método isActionKey de KeyEvent</i>
<i>FlowLayout.RIGHT</i>	<i>método isAltDown de InputEvent</i>
<i>foco</i>	<i>método isMetaDown de InputEvent</i>
<i>Font.BOLD</i>	<i>método itemStateChanged</i>
<i>Font.ITALIC</i>	<i>método keyPressed de KeyListener</i>
<i>Font.PLAIN</i>	<i>método keyReleased de KeyListener</i>
<i>gerenciador de leiaute</i>	<i>método keyTyped de KeyListener</i>
<i>Graphics Interchange Format (GIF)</i>	<i>método layoutContainer</i>
<i>ícone rollover</i>	<i>método mouseClicked</i>
<i>ID de eventos</i>	<i>método mouseDragged</i>
<i>interface ActionListener</i>	<i>método mouseEntered</i>
<i>interface ComponentListener</i>	<i>método mouseExited</i>
<i>interface ContainerListener</i>	<i>método mouseMoved</i>
<i>interface FocusListener</i>	<i>método mousePressed</i>
<i>interface Icon</i>	<i>método mouseReleased</i>
<i>interface ItemListener</i>	<i>método setAlignment</i>
<i>interface KeyListener</i>	<i>método setBackground</i>
<i>interface LayoutManager</i>	<i>método setEditable</i>
<i>interface listener de eventos</i>	<i>método setFixedCellHeight</i>
<i>interface ListSelectionListener</i>	<i>método setFixedCellWidth</i>

método <code>setHorizontalAlignment</code>	ouvinte de eventos
método <code>setHorizontalTextPosition</code>	pacote <code>java.awt</code>
método <code>setIcon</code>	pacote <code>java.awt.event</code>
método <code>setLayout</code> da classe <code>Container</code>	pacote <code>javax.swing</code>
método <code>setListData</code> de <code>JList</code>	pacote <code>javax.swing.event</code>
método <code>setMaximumRowCount</code>	registrar um ouvinte de eventos
método <code>setRolloverIcon</code>	rótulo
método <code>setSelectionMode</code>	rótulo de botão
método <code>setToolTipText</code>	rótulo de caixa de marcação
método <code>setVerticalAlignment</code>	senha
método <code>setVerticalTextPosition</code>	seta de rolagem
método <code>setVisible</code>	sistema de janela
método <code>setVisibleRowCount</code>	tecla de atalho (mnemônica)
método <code>validate</code>	tecnologias de ajuda
método <code>valueChanged</code>	texto somente de leitura
método <code>windowClosing</code>	tratador de eventos
modelo de delegação de evento	widget
modo de seleção	

Exercícios de auto-revisão

12.1 Preencha as lacunas em cada uma das seguintes afirmações:

- a) O método _____ é chamado quando se move o mouse e um “ouvinte” de eventos está registrado para tratar o evento.
- b) O texto que não pode ser modificado pelo usuário é chamado de texto _____.
- c) O _____ organiza componentes GUI em um `Container`.
- d) O método `add` para anexar componentes GUI é um método da classe _____.
- e) GUI é um acrônimo de _____.
- f) O método _____ é utilizado para configurar o gerenciador de layout para um contêiner.
- g) A chamada do método `mouseDragged` é precedida por uma chamada do método _____ e seguida por uma chamada do método _____.

12.2 Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.

- a) `BorderLayout` é o gerenciador *default* de layout para um painel de conteúdo.
- b) Quando o cursor do mouse é movido dentro dos limites de um componente GUI, o método `mouseOver` é chamado.
- c) O `JPanel` não pode ser adicionado a outro `JPanel`.
- d) Em um `BorderLayout`, dois botões adicionados à região `NORTH` serão colocados lado a lado.
- e) Ao utilizar `BorderLayout`, pode-se usar um máximo de cinco componentes.

12.3 Localize o(s) erro(s) em cada uma das instruções seguintes e explique como corrigi-lo(s).

```

a) buttonName = JButton( "Caption" );
b) JLabel aLabel, JLabel; // cria referências
c) txtField = new JTextField( 50, "Default Text" );
d) Container container = getContentPane();
   setLayout( new BorderLayout() );
   button1 = new JButton( "North Star" );
   button2 = new JButton( "South Pole" );
   container.add( button1 );
   container.add( button2 );

```

Respostas aos exercícios de auto-revisão

12.1 a) `mouseMoved`. b) não-editável (somente de leitura). c) gerenciador de layout. d) `Container`. e) interface gráfica com o usuário (*graphical user interface*). f) `setLayout`. g) `mousePressed`, `mouseReleased`.

12.2 a) Verdadeira.

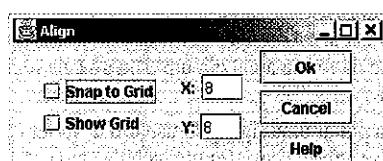
b) Falsa. O método `mouseEntered` é chamado.

- c) Falsa. O **JPanel** pode ser adicionado a outro **JPanel** porque **JPanel** deriva indiretamente de **Component**. Portanto, o **JPanel** é um **Component**. Qualquer **Component** pode ser adicionado a um **Container**.
 - d) Falsa. Será exibido apenas o último botão adicionado. Lembre-se de que apenas um componente pode ser adicionado a cada região em um **BorderLayout**.
 - e) Verdadeira.
- 12.3**
- a) **new** é necessário para instanciar o objeto.
 - b) **JLabel** é um nome de classe e não pode ser utilizado como nome de variável.
 - c) Os argumentos passados para o construtor estão invertidos. O **String** deve ser passado primeiro.
 - d) **BorderLayout** foi configurado e os componentes estão sendo adicionados sem se especificar a região. As instruções **add** adequadas podem ser

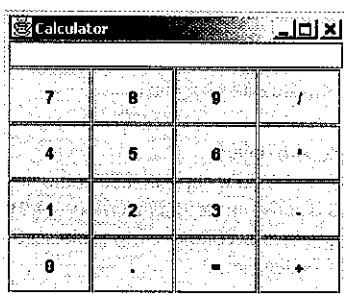
```
container.add( button1, BorderLayout.NORTH );
container.add( button2, BorderLayout.SOUTH );
```

Exercícios

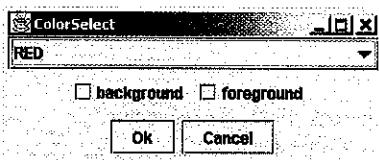
- 12.4** Preencha as lacunas em cada uma das afirmações seguintes:
- a) A classe **JTextField** herda diretamente de _____.
 - b) Os gerenciadores de layout discutidos neste capítulo são _____, _____ e _____.
 - c) O método _____ de **Container** anexa um componente GUI a um contêiner.
 - d) O método _____ é chamado quando se solta um botão do mouse (sem se mover o mouse).
 - e) A classe _____ é utilizada para criar um grupo de **JRadioButtons**.
- 12.5** Diga se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- a) Apenas um gerenciador de layout pode ser utilizado por **Container**.
 - b) Os componentes GUI podem ser adicionados a um **Container** em qualquer ordem em um **BorderLayout**.
 - c) **JRadioButton** fornece uma série de opções mutuamente exclusivas (apenas uma pode ser **true** por vez).
 - d) Utiliza-se o método **setFont** de **Graphics** para configurar a fonte para campos de texto.
 - e) A **JList** exibe uma barra de rolagem se houver mais itens na lista do que podem ser exibidos.
 - f) O objeto **Mouse** contém um método chamado **mouseDragged**.
- 12.6** Diga se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- a) O **JApplet** não tem um painel de conteúdo.
 - b) O **JPanel** é um **JComponent**.
 - c) O **JPanel** é um **Component**.
 - d) O **JLabel** é um **Container**.
 - e) A **JList** é um **JPanel**.
 - f) O **AbstractButton** é um **JButton**.
 - g) O **JTextField** é um **Object**.
 - h) **ButtonGroup** herda de **JComponent**.
- 12.7** Localize o(s) erro(s) em cada uma das instruções seguintes e explique como corrigi-lo(s).
- a) `import javax.swing.*` // inclui o pacote swing
 - b) `panelObject.setLayout(8, 8);` // configura GridLayout
 - c) `container.setLayout(`
 `new FlowLayout(FlowLayout.DEFAULT));`
 - d) `container.add(eastButton, EAST);` // BorderLayout
- 12.8** Crie a seguinte GUI. Você não tem de fornecer nenhuma funcionalidade.



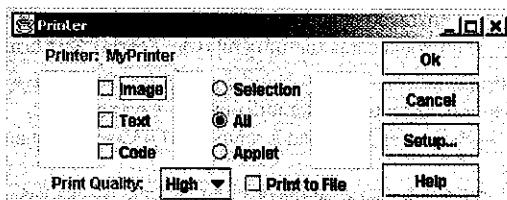
12.9 Crie a seguinte GUI. Você não tem de fornecer nenhuma funcionalidade.



12.10 Crie a seguinte GUI. Você não tem de fornecer nenhuma funcionalidade.



12.11 Crie a seguinte GUI. Você não tem de fornecer nenhuma funcionalidade.



12.12 Escreva um programa de conversão de temperatura que converte de Fahrenheit para Celsius. A temperatura em Fahrenheit deve ser inserida pelo teclado (por um `JTextField`). É necessário se utilizar um `JLabel` deve ser utilizado para exibir a temperatura convertida. Utilize a seguinte fórmula para a conversão:

$$\text{Celsius} = \frac{5}{9} \times (\text{Fahrenheit} - 32)$$

12.13 Aprimore o programa de conversão de temperatura do Exercício 12.12 adicionando a escala de temperatura Kelvin. O programa também deve permitir que o usuário faça conversões entre duas escalas quaisquer. Utilize a seguinte fórmula para a conversão entre Kelvin e Celsius (além da fórmula no Exercício 12.12):

$$\text{Kelvin} = \text{Celsius} + 273$$

12.14 Escreva um aplicativo que permite desenhar um retângulo arrastando-se o mouse na janela do aplicativo. A coordenada superior esquerda deve ser a posição em que o usuário pressiona o botão do mouse e a coordenada inferior direita deve ser a posição em que o usuário solta o botão. Também exiba a área do retângulo em um `JLabel` na região SOUTH de um `BorderLayout`. Utilize a seguinte fórmula para a área:

$$\text{área} = \text{largura} \times \text{altura}$$

12.15 Modifique o programa do Exercício 12.14 para desenhar formas diferentes. O usuário deve poder escolher entre uma elipse, um arco, uma linha, um retângulo com cantos arredondados e um polígono predefinido. Também exiba as coordenadas do mouse na barra de estado.

12.16 Escreva um programa que permita ao usuário desenhar uma forma com o mouse. A forma a ser desenhada deve ser determinada por um `KeyEvent` que utiliza as seguintes teclas: `c` desenha um círculo, `e` desenha uma elipse, `r` desenha um retângulo e `l` desenha uma linha. O tamanho e a posição da forma devem ser determinados pelos eventos `mousePressed` e `mouseReleased`. Exiba o nome da forma atual em um `JLabel` na região SOUTH de um `BorderLayout`. A forma inicial `default` deve ser um círculo.

12.17 Crie um aplicativo que permita pintar uma figura. O usuário deve ser capaz de escolher a forma a ser desenhada, a cor em que a forma deve aparecer e se a forma deve ser preenchida com cor. Utilize os componentes de interface grá-

fica com o usuário que discutimos neste capítulo, como `JComboBoxes`, `JRadioButtons` e `JCheckboxes`, para permitir ao usuário selecionar várias opções. O programa deve fornecer um objeto `JButton` que permita apagar a janela.

12.18 Escreva um programa que utiliza instruções `System.out.println` para imprimir os eventos quando eles ocorrem. Forneça uma `JComboBox` com um mínimo de quatro itens. O usuário deve ser capaz de escolher um evento da `JComboBox` para “monitorar”. Quando ocorre esse evento em particular, exiba as informações sobre o evento em uma caixa de diálogo de mensagem. Utilize o método `toString` sobre o objeto de evento para convertê-lo em uma representação de `string`.

12.19 Escreva um programa que desenha um quadrado. Enquanto o mouse se move sobre a área de desenho, repinte o quadrado com o canto superior esquerdo do quadrado, seguindo o caminho exato do cursor do mouse.

12.20 Modifique o programa da Fig. 12.19 para incorporar as cores. Em uma janela separada, forneça uma “barra de ferramentas” com objetos `JRadioButton` na parte inferior da janela que lista as seguintes seis cores: vermelho, preto, magenta, azul, verde e amarelo. A barra de ferramentas deve ser implementada como uma subclasse de `JFrame` chamada `ToolBarWindow` e deve consistir em seis botões, cada um com o nome da corpropriada. Quando se seleciona uma nova cor, o desenho deve ser feito com a nova cor. Determine a cor atual selecionada no tratador de eventos `mousePressed` da janela principal, chamando o método público `getCurrentColor` sobre o `ToolBarWindows`. [Nota: no Capítulo 13, discutimos como combinar componentes GUI e desenhos usando `JFrame` separado para cada componente. Isso confere aos programas maior flexibilidade para dispor os componentes e desenhos.]

12.21 Escreva um programa que reproduz o jogo “adivinhe o número” da seguinte forma: o programa escolhe o número a ser adivinhado selecionando aleatoriamente um inteiro no intervalo 1–1000. O programa então exibe em um rótulo:

`Eu tenho um número entre 1 e 1000; você consegue adivinhar meu número?
Por favor, digite o primeiro palpite.`

Deve-se utilizar um `JTextField` para ler o palpite. À medida que cada palpite é digitado, a cor de fundo deve alterar para vermelho ou azul. Vermelho indica que o usuário “está ficando mais quente” e azul indica que o usuário “está ficando mais frio”. Um `JLabel` deve exibir “*Muito alto*” ou “*Muito baixo*” para ajudar o usuário a se aproximar da resposta correta. Quando o usuário obtiver a resposta correta, “*Certo!*” deve ser exibido e o `JTextField` utilizado para entrada deve ser alterado para não-editável. Deve-se fornecer um `JButton` para permitir que o usuário jogue de novo. Quando se clicar no `JButton`, um novo número aleatório deverá ser gerado e a entrada `JTextField` deve ser alterada para o estado editável.

12.22 Muitas vezes é útil exibir os eventos que ocorrem durante a execução de um programa para ajudar a entender quando ocorrem os eventos e como eles são gerados. Escreva um programa que permite gerar e processar cada evento discutido neste capítulo. O programa deve fornecer os métodos das interfaces `ActionListener`, `ItemListener`, `ListSelectionListener`, `MouseListener`, `MouseMotionListener` e `KeyListener` para exibir as mensagens quando ocorrem os eventos. Utilize o método `toString` para converter os objetos de evento recebidos em cada tratador de evento para um `String` que possa ser exibido. O método `toString` cria um `String` que contém todas as informações no objeto de evento.

12.23 Modifique a solução para o Exercício 12.17 para permitir que o usuário selecione uma fonte e um tamanho de fonte e depois digite o texto em um `JTextField`. Quando o usuário pressionar *Enter*, o texto deve ser exibido no fundo com a fonte e o tamanho escolhidos. Modifique o programa ainda mais para permitir que o usuário especifique a posição exata em que o texto deve ser exibido.

12.24 Escreva um programa que permita selecionar uma forma de um `JComboBox`, depois desenhar essa forma 20 vezes em posições e com dimensões aleatórias no método `paint`. O primeiro item na `JComboBox` deve ser a forma *default* que é exibida na primeira vez que `paint` é chamado.

12.25 Modifique o Exercício 12.24 para desenhar cada uma das 20 formas dimensionadas aleatoriamente em uma cor aleatoriamente selecionada. Utilize todos os 13 objetos `Color` predefinidos em um *array* de `Colors`.

12.26 Modifique o Exercício 12.25 para permitir que o usuário selecione a cor em que as formas devem ser desenhadas a partir de um diálogo `JColorChooser`.

12.27 Escreva um programa que utiliza os métodos da interface `MouseListener` que permitem ao usuário pressionar o botão do mouse, arrastar o mouse e soltar o botão. Quando o botão do mouse for solto, desenhe um retângulo com o canto superior esquerdo, a largura e a altura apropriados. (*Dica:* o método `mousePressed` deve capturar o conjunto de coordenadas em que o usuário pressiona e segura inicialmente o botão do mouse, e o método `mouseReleased` deve capturar o conjunto de coordenadas em que o usuário solta o botão do mouse. Ambos os métodos devem armazenar os valores apropriados de coordenadas. Todos os cálculos da largura, altura e canto superior esquerdo devem ser realizados pelo método `paint` antes de a forma ser desenhada.)

12.28 Modifique o Exercício 12.27 para produzir um efeito de “elástico”. Enquanto o usuário arrasta o mouse, o usuário deve ser capaz de ver o tamanho do retângulo atual para saber a aparência exata que o retângulo terá quando o botão do mouse for liberado. (*Dica:* o método `mouseDragged` deve realizar as mesmas tarefas que `mouseReleased`.)

12.29 Modifique o Exercício 12.28 para permitir que o usuário selecione qual forma desenhar. A `JComboBox` deve fornecer opções que incluem pelo menos retângulo, elipse, linha e retângulo arredondado.

12.30 Modifique o Exercício 12.29 para permitir que o usuário selecione a cor de desenho de uma caixa de diálogo `JColorChooser`.

12.31 Modifique o Exercício 12.30 para permitir que o usuário especifique se uma forma deve aparecer preenchida ou vazada quando for desenhada. O usuário deve clicar em um `JCheckBox` para indicar preenchido ou vazio.

12.32 (*Programa de pintura*) Utilizando as técnicas dos Exercícios 9.28, 9.29, e 12.27 a 12.30 e as técnicas gráficas do Capítulo 11, rescreva o Exercício 12.31 para permitir que o usuário desenhe múltiplas formas e armazene cada uma em um *array* de formas (se você se sentir ambicioso, investigue os recursos da classe `Vector` no Capítulo 20). Para esse programa, crie suas próprias classes (como aquelas na hierarquia de classes descrita nos Exercícios 9.28 e 9.29) a partir das quais os objetos serão criados, para armazenar cada forma que o usuário desenha. As classes devem armazenar a posição, as dimensões e a cor de cada forma e devem indicar se a forma é preenchida ou não. Todas as suas classes devem se derivar de uma classe chamada `MyShape` que tem todos os recursos comuns de cada tipo de forma. Cada subclasse de `MyShape` deve ter seu próprio método `draw`, que retorna `void` e recebe um objeto `Graphics` como argumento. Quando o método `paint` da janela do aplicativo for chamado, ele deverá percorrer o *array* de formas e exibir cada forma chamando de maneira polimórfica o método `draw` da forma (passando o objeto `Graphics` como argumento). Cada método `draw` da forma deve saber desenhar a forma. No mínimo, o programa deve fornecer as seguintes classes: `MinhaLine`, `MinhaElipse`, `MeuRet`, `MeuRetRedon`. Projete a hierarquia de classes para maximizar a reutilização de *software* e coloque todas as suas classes no pacote `shapes`. Importe esse pacote para seu programa.

12.33 Modifique o Exercício 12.32 para fornecer um botão `Desfazer` que pode ser utilizado repetidamente para desfazer a última operação de pintura. Se não houver formas no *array* de formas, o botão `Desfazer` deve ser desativado.

Componentes da interface gráfica com o usuário: parte 2

Objetivos

- Criar e manipular áreas de texto, controles deslizantes, menus, menus *pop-up* e janelas.
- Ser capaz de criar objetos `JPanel` personalizados.
- Ser capaz de criar um programa que pode ser executado como *applet* ou como aplicativo.
- Ser capaz de alterar a aparência e o comportamento de uma GUI utilizando a aparência e o comportamento plugável do Swing (PLAF).
- Ser capaz de criar uma interface com múltiplos documentos com `JDesktopPane` e `JInternalFrame`.
- Ser capaz de utilizar gerenciadores avançados de layoute.

Não afirmo ter controlado os eventos, mas simplesmente confesso que os eventos me controlaram.

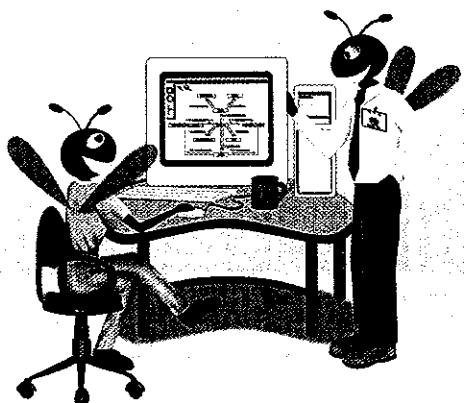
Abraham Lincoln

Um bom símbolo é o melhor argumento e um missionário para convencer milhares.

Ralph Waldo Emerson

Capture sua realidade na pintura!

Paul Cézanne



Sumário do capítulo

- 13.1 Introdução**
- 13.2 JTextArea**
- 13.3 Criando uma subclasse personalizada de JPanel**
- 13.4 Criando uma subclasse autocontida de JPanel**
- 13.5 JButton**
- 13.6 Janelas**
- 13.7 Projetando programas que podem ser executados como applets ou aplicativos**
- 13.8 Utilizando menus com frames**
- 13.9 Utilizando JPopupMenu**
- 13.10 Aparência e comportamento plugável**
- 13.11 Utilizando JDesktopPane e JInternalFrame**
- 13.12 Gerenciadores de layout**
 - 13.13 O gerenciador de layout BoxLayout**
 - 13.14 O gerenciador de layout CardLayout**
 - 13.15 O gerenciador de layout GridBagLayout**
 - 13.16 As constantes RELATIVE e REMAINDER de GridBagConstraints**
 - 13.17 (Estudo de caso opcional) Pensando em objetos: Model-View-Controller**
- 13.18 (Opcional) Descobrindo padrões de projeto: padrões de projeto usados nos pacotes java.awt e javax.swing**
 - 13.18.1 Padrões de criação de projeto**
 - 13.18.2 Padrões estruturais de projeto**
 - 13.18.3 Padrões comportamentais de projeto**
 - 13.18.4 Conclusão**

Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão • Exercícios

13.1 Introdução

Neste capítulo, continuamos nosso estudo de GUIs. Discutimos os componentes mais avançados e os gerenciadores de layout e projetamos a base para construir GUIs complexas.

Iniciamos nossa discussão com outro componente GUI baseado em texto – *JTextArea* – que permite que múltiplas linhas de texto sejam exibidas ou lidas. Continuamos com dois exemplos de *personalização da classe JPanel* em que discutimos questões que se relacionam com a pintura em componentes GUI do Swing. A seguir, ilustramos como projetar um programa Java que pode ser executado como um *applet* e um aplicativo. Um aspecto importante de qualquer GUI completa é um sistema de *menus* que permite ao usuário realizar efetivamente tarefas no programa. Os próximos dois exemplos mostram como criar e utilizar menus. A aparência e o comportamento de uma GUI do Swing podem ser uniformes em todas as plataformas em que o programa Java é executado ou a GUI pode ser personalizada com *aparência e o comportamento plugável (PLAF – pluggable look-and-feel)* do Swing. O próximo exemplo ilustra como alternar entre a *aparência e o comportamento de metal* padrão do Swing, uma aparência e comportamento que simula o *Motif* (aparência e comportamento popular no UNIX) e uma que simula a aparência e o comportamento do Microsoft Windows.

Muitos aplicativos atuais utilizam uma *interface de múltiplos documentos (MDI – multiple document interface)*, [isto é, uma janela principal (freqüentemente chamada de *janela-pai*) que contém outras janelas (freqüentemente chamadas de *janelas-filhas*)] para gerenciar vários *documentos* abertos que estão sendo processados paralelamente.

te. Por exemplo, muitos programas de correio eletrônico permitem ter várias janelas de correio eletrônico abertas ao mesmo tempo para permitir compor e/ou ler múltiplas mensagens de correio eletrônico. O próximo exemplo discute classes do Swing que fornecem suporte para criar interfaces de múltiplos documentos. Por fim, o capítulo termina com uma série de exemplos que discute vários gerenciadores de layout avançados para organização da interface gráfica com o usuário.

O Swing é um assunto extenso e complexo. Há muito mais componentes e recursos GUI do que podem ser apresentados aqui. Muito outros componentes GUI do Swing são apresentados nos demais capítulos deste livro, à medida que forem necessários. Nossa livro *Advanced Java How to Program* discutirá outros componentes e recursos mais avançados do Swing.

13.2 JTextArea

As *JTextAreas* fornecem uma área para manipulação de múltiplas linhas de texto. Semelhante à classe *JTextField*, a classe *JTextArea* herda de *JTextComponent*, que define métodos comuns para *JTextFields*, *JTextAreas* e vários outros componentes GUI baseados em texto.

O aplicativo da Fig. 13.1 demonstra *JTextAreas*. A *JTextArea* exibe texto que o usuário pode selecionar. A outra *JTextArea* não é editável. Seu propósito é exibir o texto que o usuário selecionou na primeira *JTextArea*. As *JTextAreas* não têm eventos de ação como *JTextFields*. Freqüentemente, um *evento externo* – evento gerado por um componente GUI diferente – indica quando o texto em uma *JTextArea* deve ser processado. Por exemplo, ao digitar uma mensagem de correio eletrônico, você freqüentemente clica no botão **Enviar** para selecionar o texto da mensagem e enviá-lo para o destinatário. De maneira semelhante, ao editar um documento em um processador de texto, você normalmente salva o arquivo selecionando um item de menu chamado **Salvar** ou **Salvar como....**. Nesse programa, o botão **Copy >>> (Copiar)** gera o evento externo que faz com que o texto selecionado na *JTextArea* esquerda seja copiado e exibido na *JTextArea* direita.

```

1 // Fig. 13.1: TextAreaDemo.java
2 // Copiando texto selecionado de uma área de texto para outra.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class TextAreaDemo extends JFrame {
12     private JTextArea textArea1, textArea2;
13     private JButton copyButton;
14
15     // configura a GUI
16     public TextAreaDemo()
17     {
18         super( "TextArea Demo" );
19
20         Box box = Box.createHorizontalBox();
21
22         String string = "This is a demo string to\n" +
23             "illustrate copying text\n" +
24             "from one TextArea to \n" +
25             "another TextArea using an\n" + "external event\n";
26
27         // configura textArea1
28         textArea1 = new JTextArea( string, 10, 15 );
29         box.add( new JScrollPane( textArea1 ) );
30

```

Fig. 13.1 Copiando o texto selecionado de uma área de texto para outra (parte 1 de 3).

```

31     // configura copyButton
32     copyButton = new JButton( "Copy >>>" );
33     copyButton.addActionListener(
34
35         // classe interna anônima para tratar eventos de copyButton
36         new ActionListener() {
37
38             // configura o texto na textArea2 para o
39             // texto que está selecionado na textArea1
40             public void actionPerformed( ActionEvent event )
41             {
42                 textArea2.setText( textArea1.getSelectedText() );
43             }
44
45         } // fim da classe interna anônima
46
47     ); // fim da chamada para addActionListener
48
49     box.add( copyButton );
50
51     // configura textArea2
52     textArea2 = new JTextArea( 10, 15 );
53     textArea2.setEditable( false );
54     box.add( new JScrollPane( textArea2 ) );
55
56     // adiciona uma caixa ao painel de conteúdo
57     Container container = getContentPane();
58     container.add( box ); // posiciona em BorderLayout.CENTER
59
60     setSize( 425, 200 );
61     setVisible( true );
62 }
63
64 // executa o applicativo
65 public static void main( String args[] )
66 {
67     TextAreaDemo application = new TextAreaDemo();
68
69     application.setDefaultCloseOperation(
70         JFrame.EXIT_ON_CLOSE );
71 }
72
73 } // fim da classe TextAreaDemo

```

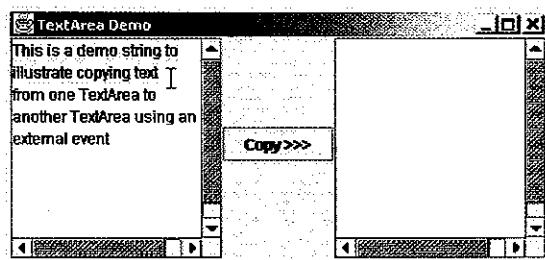


Fig. 13.1 Copiando o texto selecionado de uma área de texto para outra (parte 2 de 3).

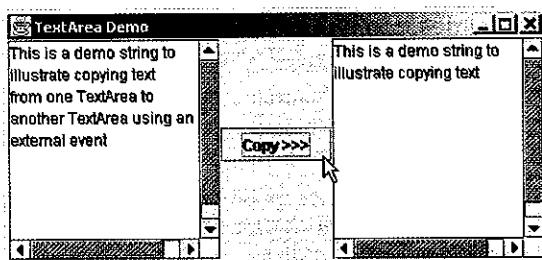


Fig. 13.1 Copiando o texto selecionado de uma área de texto para outra (parte 3 de 3).



Observação de aparência e comportamento 13.1

Freqüentemente, um evento externo determina quando o texto em uma `JTextArea` deve ser processado.

No método construtor (linhas 16 a 62), a linha 20 cria um *contêiner Box* (pacote `javax.swing`) ao qual os componentes GUI serão anexados. A classe `Box` é uma subclasse de `Container` que utiliza um gerenciador de layout `BoxLayout` para organizar os componentes GUI horizontal ou verticalmente. A Seção 13.13 discute `BoxLayout` detalhes. A classe `Box` fornece o método estático `createHorizontalBox` para criar uma `Box` que organiza os componentes da esquerda para direita, na ordem em que os componentes são anexados.

O aplicativo instancia os objetos `JTextArea textArea1` (linha 28) e `textArea2` (linha 52). Cada `JTextArea` tem 10 linhas visíveis e 15 colunas visíveis. A linha 28 especifica que o *string* deve ser exibido como o conteúdo *default* da `JTextArea`. A `JTextArea` não fornece barras de rolagem se ela não consegue exibir seu conteúdo completo. Por essa razão, a linha 29 cria um objeto `JScrollPane` que é inicializado com `textArea1` e anexa o objeto ao contêiner `box`. Por *default*, as barras de rolagem horizontais e verticais aparecerão se for necessário.

As linhas 32 a 49 instanciam o objeto `JButton copyButton` com o rótulo “*Copy >>>*”, criam uma classe interna anônima para tratar `ActionEvents` de `copyButton` e adicionam `copyButton` ao contêiner `box`. Esse botão fornece o evento externo que determina quando o texto selecionado em `textArea1` deve ser copiado para `textArea2`. Quando o usuário clica em `copyButton`, a linha 42 em `actionPerformed` indica que o método `getSelectedText` (herdado por `JTextArea` de `JTextComponent`) deve devolver o *texto selecionado* de `textArea1`. O usuário seleciona o texto arrastando o mouse sobre o texto desejado para destacá-lo. O método `setText` altera o texto em `textArea2` para o *String* que o método `getSelectedText` devolve.

As linhas 52 a 54 criam a `textArea2` e a adicionam ao contêiner `box`. As linhas 57 e 58 obtêm o painel de conteúdo para a janela e adicionam `box` ao painel de conteúdo. Lembre-se de que o layout *default* do painel de conteúdo é um `BorderLayout` e que o método `add` anexa seu argumento ao `CENTER` do `BorderLayout` se o método `add` não especificar a região.

Algumas vezes é preferível fazer o texto mudar para a próxima linha quando ele atingir o lado direito de uma `JTextArea`. Isso se chama *mudança automática de linha*.



Observação de aparência e comportamento 13.2

Para fornecer a funcionalidade de mudança automática de linha para uma `JTextArea`, invoque o método `setLineWrap` de `JTextArea` com um argumento `true`.

Este exemplo usa um JSP para fornecer a funcionalidade de rolagem para uma `JTextArea`. Por *default*, `JScrollPane` fornece barras de rolagem somente se elas forem necessárias. Você pode configurar a qualquer momento as *regras para as barras de rolagem* horizontal e vertical para o `JScrollPane` quando um `JScrollPane` é construído ou com os métodos `setHorizontalScrollBarPolicy` e `setVerticalScrollBarPolicy` da classe `JScrollPane`. A classe `JScrollPane` fornece as constantes

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

para indicar que uma barra de rolagem sempre deve aparecer, as constantes

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

para indicar que uma barra de rolagem deve aparecer somente se necessário e as constantes

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

para indicar que uma barra de rolagem nunca deve aparecer. Se a regra para a barra de rolagem horizontal for configurada como `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`, a `JTextArea` anexada ao `JScrollPane` exibirá o comportamento de mudança automática de linha.

13.3 Criando uma subclasse personalizada de JPanel

No Capítulo 12, vimos que um `JPanel` pode agregar um conjunto de componentes GUI para fins de layout. `JPanels` são bastante flexíveis. Alguns de seus muitos usos incluem a criação de *áreas dedicadas de desenho* e de áreas que recebem eventos de mouse. Os programas freqüentemente estendem a classe `JPanel` para criar novos componentes. Nossa próximo exemplo usa um `JPanel` para criar uma área dedicada de desenho. As áreas dedicadas de desenho ajudam a separar o desenho do resto de sua interface gráfica com o usuário. Isso traz benefícios para interfaces gráficas como o usuário Swing. Se os gráficos e componentes GUI do Swing não são exibidos na ordem correta, é possível que os componentes GUI não sejam exibidos corretamente. Por exemplo, para assegurar que gráficos e componentes GUI sejam ambos exibidos corretamente, podemos separar a GUI e as imagens gráficas criando áreas dedicadas de desenho como subclasses de `JPanel`.



Observação de aparência e comportamento 13.3

Combinar imagens gráficas e componentes GUI do Swing pode resultar na exibição incorreta das imagens gráficas, dos componentes GUI ou de ambos. Utilizar `JPanels` para desenhar pode eliminar esse problema fornecendo uma área dedicada para gráficos.

Os componentes do Swing que herdam da classe `JComponent` contêm o método `paintComponent` que os ajuda a desenhar adequadamente no contexto de um GUI do Swing. Ao personalizar um `JPanel` para utilizar como área dedicada de desenho, a subclasse deve sobreescrivê-lo para chamar a versão de `paintComponent` da superclasse como primeira instrução no corpo do método sobreescrito. Isso assegura que a pintura ocorre na ordem adequada e que o mecanismo de pintura do Swing permanece intato. Uma parte importante desse mecanismo é que as subclasses de `JComponent` suportam *transparência*, que pode ser configurada com o método `setOpaque` (um argumento `false` indica que o componente é transparente). Para pintar um componente corretamente, o programa deve determinar se o componente é transparente. O código que faz esta verificação está na versão de `paintComponent` da superclasse. Quando um componente é *transparente*, `paintComponent` não apaga o fundo do componente quando o programa pinta o componente. Quando o componente é *opaco*, `paintComponent` limpa o fundo antes de continuar a operação de pintura. Se a versão de `paintComponent` da superclasse não for chamada, geralmente um componente GUI opaco não será exibido corretamente na interface com o usuário. Além disso, se a versão da superclasse for chamada depois de executar as instruções personalizadas de desenho, geralmente os resultados serão apagados.



Observação de aparência e comportamento 13.4

Ao sobreescrivê-lo para chamar a versão original do método da superclasse.



Erro comum de programação 13.1

Não chamar a versão original de `paintComponent` da superclasse ao sobreescrivê-lo para chamar a versão original do método da superclasse.



Erro comum de programação 13.2

Ao sobreescrivê-lo para chamar a versão original do método da superclasse, chamar a versão original do método da superclasse depois de outro desenho ser executado apaga os outros desenhos.

As classes **JFrame** e **JApplet** não são subclasses de **JComponent**; portanto, elas não contêm o método **paintComponent**. Para desenhar diretamente em subclasses de **JFrame** e **JApplet**, sobrescreva o método **paint**.

Observação de aparência e comportamento 13.5



Chamar **repaint** para um componente GUI do Swing indica que o componente deve ser pintado assim que possível. O fundo do componente GUI é limpo somente se o componente for opaco. A maioria dos componentes do Swing é transparente por default. O método **setOpaque** de **JComponent** pode receber um argumento boolean indicando se o componente é opaco (**true**) ou transparente (**false**). Os componentes GUI do pacote **java.awt** são diferentes dos componentes do Swing porque **repaint** resulta em uma chamada do método **update** de **Component** (que limpa o fundo do componente) e o método **update** chama **paint** (em vez de **paintComponent**).

O programa das Figs. 13.2 e 13.3 demonstra uma subclasse personalizada de **JPanel**. A classe **CustomPanel** (Fig. 13.2) tem seu próprio método **paintComponent** que desenha um círculo ou um quadrado dependendo do valor passado para o método **draw** de **CustomPanel**. Com esse propósito, a linha 11 de **CustomPanel** define constantes que são utilizadas para permitir que o programa especifique a forma que um **CustomPanel** desenha sobre si próprio em cada chamada para seu método **paintComponent**. A classe **CustomPanelTest** (Fig. 13.3) cria um **CustomPanel** e uma GUI que permite ao usuário escolher a forma a desenhar.

A classe **CustomPanel** contém uma variável de instância **shape** que armazena um inteiro que representa a forma a ser desenhada. O método **paintComponent** (linhas 15 a 23) desenha uma forma no painel. Se **shape** for **CIRCLE**, o método **fillOval** de **Graphics** desenha um círculo sólido. Se **shape** for **SQUARE**, o método **fillRect** de **Graphics** desenha um quadrado sólido. O método **draw** (linhas 26 a 30) configura a variável de instância **shape** e chama **repaint** para atualizar o objeto **CustomPanel**. Observe que chamar **repaint** (que é, na verdade, **this.repaint()**) para o **CustomPanel** agenda uma operação de desenho para o **CustomPanel**. O método **paintComponent** será chamado para repintar o **CustomPanel** e desenhar a forma adequada.

```

1 // Fig. 13.2: CustomPanel.java
2 // Uma classe personalizada JPanel.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class CustomPanel extends JPanel {
11     public final static int CIRCLE = 1, SQUARE = 2;
12     private int shape;
13
14     // usa a forma para desenhar uma elipse ou um retângulo
15     public void paintComponent( Graphics g )
16     {
17         super.paintComponent( g );
18
19         if ( shape == CIRCLE )
20             g.fillOval( 50, 10, 60, 60 );
21         else if ( shape == SQUARE )
22             g.fillRect( 50, 10, 60, 60 );
23     }
24
25     // configura o valor da forma e pinta novamente o CustomPanel
26     public void draw( int shapeToDraw )
27     {
28         shape = shapeToDraw;
29         repaint();
30     }
31
32 } // fim da classe CustomPanel

```

Fig. 13.2 Definindo uma área de desenho personalizada com uma subclasse de **JPanel**.

A classe **CustomPanelTest** (Fig. 13.3) instancia um objeto **CustomPanel** (linha 22 de seu construtor) e configura a cor de fundo como verde, de modo que a área **CustomPanel** seja visível no aplicativo. Em seguida, o construtor instancia os objetos **JButton squareButton** e **circleButton**. As linhas 27 a 40 registram um tratador de eventos para **ActionEvents** de **squareButton**. As linhas 43 a 56 registram um tratador de eventos para **ActionEvents** de **circleButton**. Tanto a linha 35 como a 51 chamam o método **draw** de **CustomPanel**. Em cada caso, a constante apropriada (**CustomPanel.SQUARE** ou **CustomPanel.CIRCLE**) é passada como argumento para indicar qual forma deve ser desenhada.

```

1 // Fig. 13.3: CustomPanelTest.java
2 // Usando um objeto personalizado Panel.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class CustomPanelTest extends JFrame {
12     private JPanel buttonPanel;
13     private CustomPanel myPanel;
14     private JButton circleButton, squareButton;
15
16     // configura a GUI
17     public CustomPanelTest()
18     {
19         super( "CustomPanel Test" );
20
21         // cria área de desenho personalizada
22         myPanel = new CustomPanel();
23         myPanel.setBackground( Color.green );
24
25         // configura squareButton
26         squareButton = new JButton( "Square" );
27         squareButton.addActionListener(
28
29             // classe interna anônima para tratar de eventos de squareButton
30             new ActionListener() {
31
32                 // desenha um quadrado
33                 public void actionPerformed( ActionEvent event )
34                 {
35                     myPanel.draw( CustomPanel.SQUARE );
36                 }
37
38             } // fim da classe interna anônima
39
40         ); // fim da chamada para addActionListener
41
42         circleButton = new JButton( "Circle" );
43         circleButton.addActionListener(
44
45             // classe interna anônima para tratar eventos de circleButton
46             new ActionListener() {
47
48                 // desenha um círculo
49                 public void actionPerformed( ActionEvent event )
50                 {

```

Fig. 13.3 Desenhando em uma subclasse personalizada da classe **JPanel** (parte 1 de 2).

```

51             myPanel.draw( CustomPanel.CIRCLE );
52         }
53
54     } // fim da classe interna anônima
55
56 ); // fim da chamada para addActionListener
57
58 // configura painel contendo botões
59 buttonPanel = new JPanel();
60 buttonPanel.setLayout( new GridLayout( 1, 2 ) );
61 buttonPanel.add( circleButton );
62 buttonPanel.add( squareButton );
63
64 // anexa botão de painel e área de desenho personalizada ao painel de conteúdo
65 Container container = getContentPane();
66 container.add( myPanel, BorderLayout.CENTER );
67 container.add( buttonPanel, BorderLayout.SOUTH );
68
69 setSize( 300, 150 );
70 setVisible( true );
71 }
72
73 // executa o aplicativo
74 public static void main( String args[] )
75 {
76     CustomPanelTest application = new CustomPanelTest();
77
78     application.setDefaultCloseOperation(
79         JFrame.EXIT_ON_CLOSE );
80 }
81
82 } // fim da classe CustomPanelTest

```

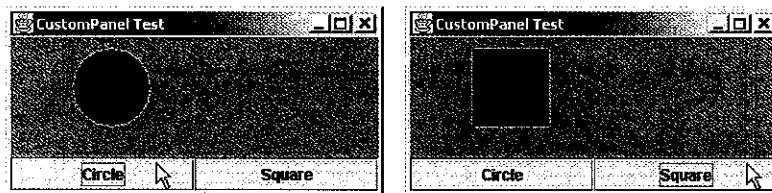


Fig. 13.3 Desenhando em uma subclasse personalizada da classe `JPanel` (parte 2 de 2).

Para dispor os botões, `CustomPanelTest` cria o `JPanel` `buttonPanel` com um `GridLayout` de uma linha e duas colunas (linhas 59 e 60) e a seguir adiciona os botões ao `buttonPanel` (linhas 61 e 62). Por fim, `CustomPanelTest` adiciona `myPanel` à região `CENTER` do painel de conteúdo e `buttonPanel` é adicionado à região `SUL` do painel de conteúdo. Observe que `BorderLayout` expande `myPanel` para preencher a região central.

13.4 Criando uma subclasse autocontida de `JPanel`

`JPanels` não suportam eventos convencionais suportados por outros componentes GUI, como botões, campos de texto e janelas. No entanto, os `JPanels` são capazes de reconhecer eventos de nível mais baixo, como eventos de mouse e de teclas. O programa das Figs. 13.4 e 13.5 permite que o usuário desenhe uma elipse em uma subclasse de `JPanel` arrastando o mouse através do painel. A classe `SelfContainedPanel` (Fig. 13.4) espera seus próprios eventos de mouse e desenha uma elipse em si mesma em resposta a esses eventos. A posição e o tamanho da elipse são determinados a partir das coordenadas dos eventos do mouse. As coordenadas nas quais o usuário pressiona o botão do mouse especificam o ponto de partida da caixa delimitadora da elipse. À medida que o usuário arrasta o

mouse, as coordenadas do ponteiro especificam outro ponto. O programa utiliza estes pontos para calcular as coordenadas x-y do canto superior esquerdo, a largura e a altura da caixa delimitadora da elipse. O tamanho da elipse muda continuamente enquanto o usuário arrasta o mouse. Quando o usuário libera o botão do mouse, o programa calcula a caixa delimitadora final para a elipse e desenha a elipse. A linha 4 da Fig. 13.4 indica que a classe `SelfContainedPanel` está no pacote `com.deitel.jhttp4.ch13` para reutilização futura. A classe `SelfContainedPanelTest` importa `SelfContainedPanel` na linha 13 da Fig. 13.5.

```

1 // Fig. 13.4: SelfContainedPanel.java
2 // Uma classe autocontida JPanel que
3 // trata seus próprios eventos de mouse.
4 package com.deitel.jhttp4.ch13;
5
6 // Pacotes do núcleo de Java
7 import java.awt.*;
8 import java.awt.event.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 public class SelfContainedPanel extends JPanel {
14     private int x1, y1, x2, y2;
15
16     // configura tratamento de eventos do mouse para o SelfContainedPanel
17     public SelfContainedPanel()
18     {
19         // configura o ouvinte para o mouse
20         addMouseListener(
21
22             // classe interna anônima para tratamento de eventos
23             // de pressionamento do botão e liberação do botão do mouse
24             new MouseAdapter() {
25
26                 // trata eventos de pressionamento do botão do mouse
27                 public void mousePressed( MouseEvent event )
28                 {
29                     x1 = event.getX();
30                     y1 = event.getY();
31                 }
32
33                 // trata eventos de liberação dos botões do mouse
34                 public void mouseReleased( MouseEvent event )
35                 {
36                     x2 = event.getX();
37                     y2 = event.getY();
38                     repaint();
39                 }
40
41             } // fim da classe interna anônima
42
43         ); // fim da chamada para addMouseListener
44
45         // configura o ouvinte para os movimentos do mouse
46         addMouseMotionListener(
47
48             // classe interna anônima para tratar de eventos de arrasto do mouse
49             new MouseMotionAdapter() {
50

```

Fig. 13.4 Subclasse personalizada de JPanel que processa eventos do mouse (parte 1 de 2).

```

51         // trata eventos de arrasto do mouse
52         public void mouseDragged( MouseEvent event )
53         {
54             x2 = event.getX();
55             y2 = event.getY();
56             repaint();
57         }
58     } // fim da classe interna anônima
59 } // fim da chamada para addMouseMotionListener
60
61 } // fim do construtor
62
63 // devolve a largura e altura preferidas do SelfContainedPanel
64 public Dimension getPreferredSize()
65 {
66     return new Dimension( 150, 100 );
67 }
68
69 // pinta uma elipse nas coordenadas especificadas
70 public void paintComponent( Graphics g )
71 {
72     super.paintComponent( g );
73
74     g.drawOval( Math.min( x1, x2 ), Math.min( y1, y2 ),
75                 Math.abs( x1 - x2 ), Math.abs( y1 - y2 ) );
76 }
77
78 }
79
80 } // fim da classe SelfContainedPanel

```

Fig. 13.4 Subclasse personalizada de JPanel que processa eventos do mouse (parte 2 de 2).

A classe **SelfContainedPanel** (Fig. 13.4) estende a classe **JPanel**. As variáveis de instância **x1** e **y1** armazenam as coordenadas iniciais nas quais o evento **mousePressed** ocorre no **SelfContainedPanel**. As variáveis de instância **x2** e **y2** armazenam as coordenadas para as quais o usuário arrasta o mouse ou libera o botão do mouse. Todas as coordenadas são relativas ao canto superior esquerdo do **SelfContainedPanel**.



Observação de aparência e comportamento 13.6

O desenho em qualquer componente GUI é realizado com coordenadas que são medidas a partir do canto superior esquerdo (0, 0) desse componente GUI.

O construtor **SelfContainedPanel** (linhas 17 a 63) utiliza os métodos **addMouseListener** e **addMouseMotionListener** para registrar objetos da classe interna anônima para tratar os eventos do mouse e os eventos de movimento do mouse para o **SelfContainedPanel**. Somente **mousePressed** (linhas 27 a 31), **mouseReleased** (linhas 34 a 39) e **mouseDragged** (linhas 52 a 57) são sobreescritos para realizar tarefas. Os outros métodos de tratamento dos eventos do mouse são herdados pelas classes internas anônimas das classes adaptadoras **MouseAdapter** e **MouseMotionAdapter**.

Estendendo a classe **JPanel**, estamos, na realidade, criando um novo componente GUI. Os gerenciadores de layout utilizam freqüentemente um método **getPreferredSize** do componente GUI (herdado da classe **java.awt.Component**) para determinar a largura e a uma altura preferidas de um componente ao dispor esse componente como parte de uma GUI. Se um novo componente tiver uma largura e uma altura preferidas, ele deve sobreescriver o método **getPreferredSize** (linhas 66 a 69) para devolver essa largura e altura como objeto da classe **Dimension** (pacote **java.awt**).



Observação de aparência e comportamento 13.7

O tamanho default de um objeto JPanel é 10 pixels de largura e 10 pixels de altura.



Observação de aparência e comportamento 13.8

Ao derivar de `JPanel` (ou qualquer outro `JComponent`), sobrescreva o método `getPreferredSize` se o novo componente precisar ter largura e altura preferidas específicas.

O método `paintComponent` (linhas 72 a 78) desenha uma elipse que usa os valores atuais das variáveis de instância `x1` e `y1`, `x2` e `y2`. A largura, a altura e o canto superior esquerdo são determinados pelo usuário pressionando-se e mantendo-se pressionado o botão do mouse, arrastando o mouse e liberando o botão do mouse na área de desenho do `SelfContainedPanel`.

As coordenadas iniciais `x1` e `y1` na área de desenho do `SelfContainedPanel` são capturadas no método `mousePressed` (linhas 27 a 31). Enquanto o usuário arrasta o mouse depois da operação `mousePressed` inicial, o programa gera uma série de chamadas para `mouseDragged` (linhas 52 a 57), enquanto o usuário continua a segurar o botão do mouse e a mover o mouse. Cada chamada captura nas variáveis `x2` e `y2` a localização atual do mouse com relação ao canto superior esquerdo do `SelfContainedPanel` e chama `repaint` para desenhar a versão atual da elipse. O desenho é estritamente confinado ao `SelfContainedPanel`, mesmo se o usuário arrastar para fora da área de desenho do `SelfContainedPanel`. Qualquer coisa desenhada fora do `SelfContainedPanel` é cortada – os pixels não são exibidos fora dos limites do `SelfContainedPanel`.

Os cálculos fornecidos no método `paintComponent` determinam o canto superior esquerdo adequado utilizando o método `Math.min` duas vezes para localizar as menores coordenadas `x` e `y`. A largura e a altura da elipse devem ser valores positivos ou a elipse não será exibida. O método `Math.abs` obtém o valor absoluto das subtrações `x1 - x2` e `y1 - y2` que determinam a largura e a altura do retângulo delimitador da elipse, respectivamente. Quando os cálculos estiverem completos, `paintComponent` desenha a elipse. A chamada à versão da superclasse de `paintComponent`, no início do método, garante que a elipse previamente exibida no `SelfContainedPanel` seja apagada antes de a nova ser exibida.



Observação de aparência e comportamento 13.9

A maioria dos componentes GUI do Swing pode ser transparente ou opaco. Se um componente GUI do Swing é opaco, quando seu método `paintComponent` for chamado, seu fundo será limpo. Caso contrário, o fundo não será limpo. Somente componentes opacos podem exibir uma cor de fundo personalizada.



Observação de aparência e comportamento 13.10

Objetos `JPanel` são opacos por default.

Quando o usuário libera o botão do mouse, o método `mouseReleased` (linhas 34 a 39) captura nas variáveis `x2` e `y2` a localização final do mouse e invoca `repaint` para desenhar a versão final da elipse.

O construtor da classe `SelfContainedPanelTest` (linhas 21 a 57 da Fig. 13.5) cria uma instância da classe `SelfContainedPanel` (linha 24) e configura a cor de fundo (linha 25) do `SelfContainedPanel` como amarelo, de modo que a área seja visível contra o fundo da janela do aplicativo.

```

1 // Fig. 13.5: SelfContainedPanelTest.java
2 // Criando uma subclasse autocontida de JPanel
3 // que processa seus próprios eventos de mouse.
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11

```

Fig. 13.5 Capturando eventos do mouse com um `JPanel` (parte 1 de 3).

```

12 // pacotes Deitel
13 import com.deitel.jhtp4.ch13.SelfContainedPanel;
14
15 public class SelfContainedPanelTest extends JFrame {
16     private SelfContainedPanel myPanel;
17
18
19     // configura a GUI e os tratadores de eventos de
20     // movimentação do mouse para a janela do aplicativo
21     public SelfContainedPanelTest()
22     {
23         // configura um SelfContainedPanel
24         myPanel = new SelfContainedPanel();
25         myPanel.setBackground( Color.yellow );
26
27         Container container = getContentPane();
28         container.setLayout( new FlowLayout() );
29         container.add( myPanel );
30
31         // configura o tratamento de eventos do mouse
32         addMouseMotionListener(
33
34             // classe interna anônima para tratamento de eventos do mouse
35             new MouseMotionListener() {
36
37                 // trata eventos de arrasto do mouse
38                 public void mouseDragged( MouseEvent event )
39                 {
40                     setTitle( "Dragging: x=" + event.getX() +
41                             "; y=" + event.getY() );
42                 }
43
44                 // trata eventos de movimentação do mouse
45                 public void mouseMoved( MouseEvent event )
46                 {
47                     setTitle( "Moving: x=" + event.getX() +
48                             "; y=" + event.getY() );
49                 }
50
51             } // fim da classe interna anônima
52
53         ); // fim da chamada para addMouseMotionListener
54
55         setSize( 300, 200 );
56         setVisible( true );
57     }
58
59     // executa o aplicativo
60     public static void main( String args[] )
61     {
62         SelfContainedPanelTest application =
63             new SelfContainedPanelTest();
64
65         application.setDefaultCloseOperation(
66             JFrame.EXIT_ON_CLOSE );
67     }
68
69 } // fim da classe SelfContainedPanelTest

```

Fig. 13.5 Capturando eventos do mouse com um JPanel (parte 2 de 3).

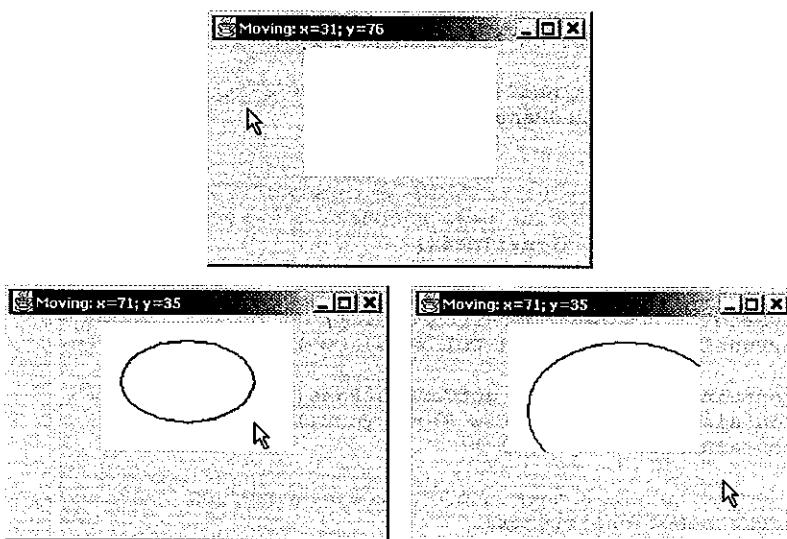


Fig. 13.5 Capturando eventos do mouse com um JPanel (parte 3 de 3).

Gostaríamos que este programa fizesse a distinção entre eventos de movimento do mouse no `SelfContainedPanel` e eventos de movimento do mouse na janela do aplicativo e, por isso, as linhas 32 a 53 registram um objeto de uma classe interna anônima para tratar os eventos de movimento do mouse do aplicativo. Os tratadores de eventos `mouseDragged` (linhas 38 a 42) e `mouseMoved` (linhas 45 a 49) utilizam o método `setTitle` (herdado da classe `java.awt.Frame`) para exibir um `String` na barra de título da janela indicando as coordenadas `x` e `y` nas quais o evento de movimento do mouse ocorreu.

Ao executar esse programa, tente arrastar a partir do fundo da janela do aplicativo para dentro da área `SelfContainedPanel` para ver que os eventos de arrasto são enviados para a janela do aplicativo, e não para o `SelfContainedPanel`. Depois, inicie uma nova operação de arrasto dentro da área `SelfContainedPanel` e arraste para fora sobre o fundo da janela do aplicativo para ver que os eventos de arrasto são enviados para o `SelfContainedPanel`, e não para a janela do aplicativo.



Observação de aparência e comportamento 13.11

Uma operação de arrasto do mouse inicia com um evento de pressionamento do mouse. Todos os eventos subsequentes de arrasto do mouse (até que o usuário libere o botão) são enviados para o componente GUI que recebeu o evento original de pressionamento do mouse.

13.5 JSlider

`JSliders` permitem ao usuário fazer uma seleção a partir de um intervalo de valores inteiros. A classe `JSlider` herda de `JComponent`. A Fig. 13.6 mostra um `JSlider` horizontal com *marcas de medida* (*tick marks*) e um *marcador* (*thumb*) que permitem selecionar um valor. Os `JSliders` são altamente personalizáveis, no sentido de que podem exibir marcas de medida principais, marcas de medida secundárias e rótulos para as marcas de medida. Eles também suportam a *aderência às marcas* (*snap-to ticks*), modo de operação em que o ato de posicionar o marcador entre duas marcas de medida faz o marcador *aderir* à marca de medida mais próxima.



Fig. 13.6 Um componente `JSlider` horizontal.

A maioria dos componentes GUI do Swing suporta as interações do usuário feitas pelo mouse e pelo teclado. Por exemplo, se um **JSlider** tem o *foco* (isto é, ele é o componente GUI atualmente selecionado na interface com o usuário), a *tecla de seta para esquerda* e a *tecla da seta para direita* fazem com que o marcador do **JSlider** diminua ou aumente uma unidade, respectivamente. A *tecla da seta para baixo* e *tecla da seta para cima* também fazem com que o marcador do **JSlider** diminua ou aumente uma unidade, respectivamente. A tecla *PgDn* (página para baixo) e a tecla *PgUp* (página para cima) fazem com que o marcador do **JSlider** diminua ou aumente por *incrementos de bloco* de um décimo da faixa de valores, respectivamente. A *tecla Home* move o marcador para o valor mínimo do **JSlider** e a *tecla End* move o marcador para o valor máximo do **JSlider**.



Observação de aparência e comportamento 13.12

A maioria dos componentes Swing suporta as interações do usuário feitas pelo mouse e pelo teclado.

JSliders têm uma *orientação horizontal* ou uma *orientação vertical*. Para um **JSlider** horizontal, o valor mínimo está na extrema esquerda e o valor máximo está na extrema direita do **JSlider**. Para um **JSlider** vertical, o valor mínimo está na extremidade inferior e o valor máximo está na extremidade superior do **JSlider**. A posição relativa do marcador indica o valor atual do **JSlider**.



Observação de aparência e comportamento 13.13

As posições de valor mínimo e máximo em um JSlider podem ser invertidas chamando-se o método setInverted de JSlider com o argumento booleano true.

O programa das Figs. 13.7 e 13.8 permite dimensionar um círculo desenhado em uma subclasse de **JPanel** denominada **OvalPanel** (Fig. 13.7). O usuário especifica o diâmetro do círculo com um **JSlider** horizontal. A classe de aplicativo **SliderDemo** (Fig. 13.8) cria o **JSlider** que controla o diâmetro do círculo. A classe **OvalPanel** é uma subclasse de **JPanel** que sabe desenhar um círculo por si própria utilizando sua própria variável de instância **diameter** para determinar o diâmetro do círculo – **diameter** é utilizada como a largura e a altura do quadrado delimitador em que o círculo é exibido. O valor de **diameter** é ajustado quando o usuário interage com o **JSlider**. O tratador de eventos chama o método **setDiameter** da classe **OvalPanel** para ajustar **diameter** e chama **repaint** para desenhar o novo círculo. A chamada a **repaint** resulta em uma chamada ao método **paintComponent** de **OvalPanel**.

A classe **OvalPanel** (Fig. 13.7) contém um método **paintComponent** (linhas 14 a 19) que desenha uma elipse preenchida (um círculo nesse exemplo), um método **setDiameter** (linhas 22 a 28) que altera o **diameter** do círculo e redesenha (**repaint**) o **OvalPanel**, um método **getPreferredSize** (linhas 31 a 34) que define a largura e a altura preferidas de um **OvalPanel** e um método **getMinimumSize** (linhas 37 a 40) que define a largura e a altura mínimas de um **OvalPanel**.



Observação de aparência e comportamento 13.14

Se um novo componente GUI tiver largura e altura mínimas (isto é, dimensões menores resultariam em um componente ineficiente na tela), sobrescreva o método getMinimumSize para devolver a largura e a altura mínimas como uma instância da classe Dimension.



Observação de aparência e comportamento 13.15

Para muitos componentes GUI, o método getMinimumSize é definido de modo a devolver o resultado de uma chamada ao método getPreferredSize desse componente.

```

1 // Fig. 13.7: OvalPanel.java
2 // Uma classe personalizada JPanel.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6

```

Fig. 13.7 Subclasse personalizada de **JPanel** para desenhar círculos de um diâmetro especificado (parte 1 de 2).

```

7 // Pacotes de extensão de Java
8 import javax.swing.*;
9
10 public class OvalPanel extends JPanel {
11     private int diameter = 10;
12
13     // desenha uma elipse com o diâmetro especificado
14     public void paintComponent( Graphics g )
15     {
16         super.paintComponent( g );
17
18         g.fillOval( 10, 10, diameter, diameter );
19     }
20
21     // valida e configura o diâmetro e depois pinta novamente
22     public void setDiameter( int newDiameter )
23     {
24         // se o diâmetro é inválido, usa 10 por default
25         diameter = ( newDiameter >= 0 ? newDiameter : 10 );
26
27         repaint();
28     }
29
30     // usado pelo gerenciador de layout para determinar o tamanho preferido
31     public Dimension getPreferredSize()
32     {
33         return new Dimension( 200, 200 );
34     }
35
36     // usado pelo gerenciador de layout para determinar o tamanho mínimo
37     public Dimension getMinimumSize()
38     {
39         return getPreferredSize();
40     }
41
42 } // fim da classe OvalPanel

```

Fig. 13.7 Subclasse personalizada de `JPanel` para desenhar círculos de um diâmetro especificado (parte 2 de 2).

O construtor da classe `SliderDemo` (linhas 17 a 54 da Fig. 13.8) instancia o objeto `OvalPanel myPanel` e configura sua cor de fundo (linhas 22 e 23). As linhas 26 a 27 instanciam o objeto `JSlider diameterSlider` para controlar o diâmetro do círculo desenhado no `OvalPanel`. A orientação de `diameterSlider` é `HORIZONTAL` (uma constante da interface `SwingConstants`). O segundo e o terceiro argumentos do construtor de `JSlider` indicam os valores inteiros mínimo e máximo no intervalo de valores para este `JSlider`. O último argumento do construtor indica que o valor inicial do `JSlider` (isto é, onde o marcador é exibido) deve ser 10.

```

1 // Fig. 13.8: SliderDemo.java
2 // Usando um JSlider para dimensionar uma elipse.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7

```

Fig. 13.8 Usando um `JSlider` para determinar o diâmetro de um círculo (parte 1 de 3).

```

8 // Pacotes de extensão de Java
9 import javax.swing.*;
10 import javax.swing.event.*;
11
12 public class SliderDemo extends JFrame {
13     private JSlider diameterSlider;
14     private OvalPanel myPanel;
15
16     // configura a GUI
17     public SliderDemo()
18     {
19         super( "Slider Demo" );
20
21         // configura OvalPanel
22         myPanel = new OvalPanel();
23         myPanel.setBackground( Color.yellow );
24
25         // configura o JSlider para controlar o valor do diâmetro
26         diameterSlider =
27             new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
28         diameterSlider.setMajorTickSpacing( 10 );
29         diameterSlider.setPaintTicks( true );
30
31         // registra o ouvinte de eventos do JSlider
32         diameterSlider.addChangeListener(
33
34             // classe interna anônima para tratar de eventos do JSlider
35             new ChangeListener() {
36
37                 // trata mudança de valor do controle deslizante
38                 public void stateChanged( ChangeEvent e )
39                 {
40                     myPanel.setDiameter( diameterSlider.getValue() );
41                 }
42
43             } // fim da classe interna anônima
44
45         ); // fim da chamada para addChangeListener
46
47         // anexa componentes ao painel de conteúdo
48         Container container = getContentPane();
49         container.add( diameterSlider, BorderLayout.SOUTH );
50         container.add( myPanel, BorderLayout.CENTER );
51
52         setSize( 220, 270 );
53         setVisible( true );
54     }
55
56     // executa o aplicativo
57     public static void main( String args[] )
58     {
59         SliderDemo application = new SliderDemo();
60
61         application.setDefaultCloseOperation(
62             JFrame.EXIT_ON_CLOSE );
63     }
64 }
65 // fim da classe SliderDemo

```

Fig. 13.8 Usando um `JSlider` para determinar o diâmetro de um círculo (parte 2 de 3).

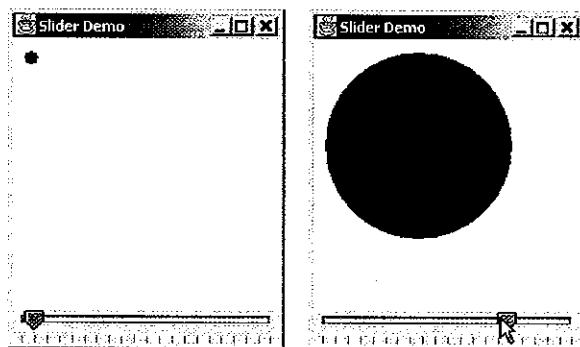


Fig. 13.8 Usando um `JSlider` para determinar o diâmetro de um círculo (parte 3 de 3).

As linhas 28 e 29 personalizam a aparência do `JSlider`. O método `setMajorTickSpacing` indica que cada marca de medida representa 10 valores no intervalo de valores suportados pelo `JSlider`. O método `setPaintTicks` com um argumento `true` indica que as marcas de medida devem ser exibidas (por *default* elas não são exibidas). Veja a documentação *on-line* de `JSlider` para obter mais informações sobre os métodos que são utilizados para personalizar a aparência de um `JSlider`.

`JSliders` geram `ChangeEvent`s (pacote `javax.swing.event`) quando o usuário interage com um `JSlider`. O objeto de uma classe que implementa a interface `ChangeListener` (pacote `javax.swing.event`) e define o método `stateChanged` pode responder para `ChangeEvent`s. As linhas 32 a 45 registram um objeto de uma classe interna anônima que implementa `ChangeListener` para tratar os eventos de `diameter`. Quando o método `stateChanged` é chamado em resposta a uma interação do usuário, ele chama o método `setDiameter` de `myPanel` e passa o valor atual do `JSlider` como argumento. O método `getValue` da classe `JSlider` devolve a posição atual do marcador.

13.6 Janelas

Do Capítulo 9 até este, a maioria dos aplicativos utilizou uma instância de uma subclasse de `JFrame` como a janela do aplicativo. Nesta seção, discutimos várias questões importantes concernentes a `JFrames`.

O `JFrame` é uma *janela* com uma *barra de título* e uma *borda*. A classe `JFrame` é uma subclasse de `java.awt.Frame` (que é uma subclasse de `java.awt.Window`). Como tal, o `JFrame` é um dos poucos componentes GUI do Swing que não é considerado um componente GUI peso-leve. Ao contrário da maioria dos componentes do Swing, o `JFrame` não é escrito completamente em Java. Na verdade, quando você exibe uma janela de um programa Java, a janela faz parte do conjunto dos componentes GUI da plataforma local — a janela será semelhante a todas as outras janelas exibidas nessa plataforma. Quando um programa Java roda em um Macintosh e exibe uma janela, a barra de título e as bordas da janela serão semelhantes às de outros aplicativos do Macintosh. Quando um programa Java roda no Microsoft Windows e exibe uma janela, a barra de título e as bordas da janela serão semelhantes às de outros aplicativos do Microsoft Windows. E quando um programa Java rodar em uma plataforma Unix e exibir uma janela, a barra de título e as bordas da janela serão semelhantes às de outros aplicativos Unix nessa plataforma.

A classe `JFrame` suporta três operações quando o usuário fecha a janela. Por *default*, oculta-se uma janela (isto é, removida da tela) quando o usuário fecha uma janela. Isso pode ser controlado com o método `setDefaultCloseOperation` de `JFrame`. A interface `WindowConstants` (pacote `javax.swing`) define três constantes para utilizar com esse método — `DISPOSE_ON_CLOSE`, `DO NOTHING_ON_CLOSE` e `HIDE_ON_CLOSE` (*o default*). A maioria das plataformas limita o número total de janelas que podem ser exibidas na tela. Como tal, a janela é um recurso valioso que deve ser devolvido ao sistema quando não for mais necessário. A classe `Window` (uma superclasse indireta de `JFrame`) define o método `dispose` para esse propósito. Quando uma `Window` não é mais necessária em um aplicativo, você deve explicitamente descartá-la (`dispose`). Isso pode ser feito chamando-se explicitamente o método `dispose` da `Window` ou chamando-se o método `setDefaultCloseOperation` com o argumento `WindowConstants.DISPOSE_ON_CLOSE`. Além disso, terminar um aplicativo devol-

verá os recursos da janela ao sistema. Configurar a operação de fechamento *default* como `DO NOTHING_ON_CLOSE` indica que você determinará o que fazer quando o usuário indicar que a janela deve ser fechada.



Observação de engenharia de software 13.1

As janelas são um recurso valioso do sistema que deve ser devolvido ao sistema quando não são mais necessárias.

Por *default*, a janela não é exibida na tela até que o programa invoque o método `setVisible` da janela (herdado da classe `java.awt.Component`) com `true` como argumento ou invoque o método `show` da janela, que não possui argumentos. Além disso, o tamanho de uma janela deve ser configurado com uma chamada ao método `setSize` (herdado da classe `java.awt.Component`). A posição de uma janela quando aparece na tela é especificada com o método `setLocation` (herdado da classe `java.awt.Component`).



Erro comum de programação 13.3

Esquecer de chamar o método `show` ou o método `setVisible` sobre uma janela é um erro de lógica em tempo de execução; a janela não é exibida.



Erro comum de programação 13.4

Esquecer de chamar o método `setSize` sobre uma janela é um erro de lógica em tempo de execução – somente a barra de título aparece.

Todas as janelas geram *eventos de janela* quando o usuário manipula a janela. Os ouvintes (*listeners*) de eventos são registrados para eventos de janela com o método `addWindowListener` da classe `Window`. A interface `WindowListener` (implementada por ouvintes de eventos de janela) fornece sete métodos para tratar eventos de janela – `windowActivated` (chamado quando o usuário torna uma janela ativa), `windowClosed` (chamado depois que a janela é fechada), `windowClosing` (chamado quando o usuário inicia o fechamento da janela), `windowDeactivated` (chamado quando o usuário torna outra janela a janela ativa), `windowIconified` (chamado quando o usuário minimiza uma janela), `windowDeiconified` (chamado quando o usuário restaura uma janela que estava minimizada) e `windowOpened` (chamado quando uma janela é exibida pela primeira vez na tela).

A maioria das janelas tem um ícone no canto superior esquerdo ou no canto superior direito que permite ao usuário fechar a janela e terminar um programa. A maioria das janelas também tem um ícone no canto superior esquerdo da janela que exibe um menu quando o usuário clica no ícone. Esse menu normalmente contém uma opção **Close** para fechar a janela e várias outras opções para manipular a janela.

13.7 Projetando programas que podem ser executados como applets ou aplicativos

Às vezes é desejável projetar um programa Java que possa ser executado tanto como aplicativo independente (*stand-alone*) quanto como *applet* em um navegador da Web. Pode-se utilizar um programa assim para fornecer a mesma funcionalidade para usuários em todo o mundo tornando o *applet* disponível para *download* pela Web e pode ser instalado em um computador como aplicativo independente. O exemplo a seguir discute como criar um pequeno programa que pode ser executado como *applet* e como aplicativo. [Nota: nos Capítulos 16 e 17, discutimos diversas questões que tornam os *applets* diferentes dos aplicativos e impedem que certos aspectos dos aplicativos funcionem em um *applet* devido às restrições de segurança que freqüentemente são impostas aos *applets*.]

JFrames são freqüentemente utilizados para criar *aplicativos baseados em GUI*. O **JFrame** fornece o espaço em que a GUI do aplicativo é construída. Quando o **JFrame** é fechado, o aplicativo é finalizado. Nesta seção, demonstramos como converter um *applet* em um aplicativo baseado em GUI. O programa da Fig. 13.9 apresenta um *applet* que também pode ser executado como aplicativo.



Observação de engenharia de software 13.2

Ao projetar um programa para executar como applet e como aplicativo, comece definindo-o como applet porque eles têm limitações devido às restrições de segurança impostas a eles pelos navegadores da Web. Se o programa rodar adequadamente como applet, pode-se fazê-lo funcionar adequadamente como aplicativo. Entretanto, o inverso nem sempre é verdadeiro.

Nossa classe de *applet* **DrawShapes** apresenta ao usuário três botões que, ao serem pressionados, fazem com que uma instância da classe **DrawPanel** (linhas 121 a 126) desenhe aleatoriamente uma linha, um retângulo ou uma elipse (dependendo de qual botão é pressionado). O *applet* não contém nenhum recurso novo no que diz respeito aos componentes GUI, leiautes ou desenhos. O único recurso novo é que a classe **DrawShapes** agora também contém um método **main** (linhas 60 a 99) que pode ser utilizado para executar o programa como um aplicativo. Discutimos esse método em detalhes abaixo.

O documento HTML que carrega o *applet* no **appletviewer** ou em um navegador da Web especifica a largura e altura do *applet* como 300 e 200, respectivamente. Quando o programa roda como aplicativo com o interpretador **java**, você pode fornecer argumentos para o programa (chamados de *argumentos de linha de comando*) que especificam a largura e a altura da janela do aplicativo. Por exemplo, o comando

```
java DrawShapes 600 400
```

especifica dois argumentos de linha de comando – 600 e 400 – que são utilizados como a largura e a altura da janela do aplicativo. Java passa os argumentos de linha de comando para **main** como o *array* de **Strings** chamado **args** que declaramos na lista de parâmetros de cada método **main** de aplicativo, mas não utilizamos até esse ponto. O primeiro argumento depois do nome da classe do aplicativo é o primeiro **String** no *array* **args** e o comprimento do *array* é o número total de argumentos de linha de comando. A linha 60 inicia a definição de **main** e declara o *array* **args** como *array* de **Strings** que permite ao aplicativo acessar os argumentos de linha de comando. A linha 62 define as variáveis **width** e **height** que são utilizadas para especificar o tamanho da janela do aplicativo.

```

1 // Fig. 13.9: DrawShapes.java
2 // Desenha linhas, retângulos e elipses aleatoriamente.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class DrawShapes extends JApplet {
12     private JButton choices[];
13     private String names[] = { "Line", "Rectangle", "Oval" };
14     private JPanel buttonPanel;
15     private DrawPanel drawingPanel;
16     private int width = 300, height = 200;
17
18     // inicializa o applet; configura a GUI
19     public void init()
20     {
21         // configura DrawPanel
22         drawingPanel = new DrawPanel( width, height );
23
24         // cria um novo array de botões
25         choices = new JButton[ names.length ];
26
27         // configura painel para os botões
28         buttonPanel = new JPanel();
29         buttonPanel.setLayout(
30             new GridLayout( 1, choices.length ) );
31
32         // configura botões e registra seus ouvintes
33         ButtonHandler handler = new ButtonHandler();
34
35         for ( int count = 0; count < choices.length; count++ ) {

```

Fig. 13.9 Criando um aplicativo baseado em GUI a partir de um *applet* (parte 1 de 4).

```

36     choices[ count ] = new JButton( names[ count ] );
37     buttonPanel.add( choices[ count ] );
38     choices[ count ].addActionListener( handler );
39 }
40
41 // anexa componentes ao painel de conteúdo
42 Container container = getContentPane();
43 container.add( buttonPanel, BorderLayout.NORTH );
44 container.add( drawingPanel, BorderLayout.CENTER );
45 }
46
47 // permite que o aplicativo especifique a largura da área de desenho
48 public void setWidth( int newWidth )
49 {
50     width = ( newWidth >= 0 ? newWidth : 300 );
51 }
52
53 // permite que o aplicativo especifique a altura da área de desenho
54 public void setHeight( int newHeight )
55 {
56     height = ( newHeight >= 0 ? newHeight : 200 );
57 }
58
59 // executa o applet como aplicativo
60 public static void main( String args[] )
61 {
62     int width, height;
63
64     // verifica se há argumentos da linha de comando
65     if ( args.length != 2 ) {
66         width = 300;
67         height = 200;
68     }
69     else {
70         width = Integer.parseInt( args[ 0 ] );
71         height = Integer.parseInt( args[ 1 ] );
72     }
73
74     // cria a janela na qual o applet será executado
75     JFrame applicationWindow =
76         new JFrame( "An applet running as an application" );
77
78     applicationWindow.setDefaultCloseOperation(
79         JFrame.EXIT_ON_CLOSE );
80
81     // cria uma instância do applet
82     DrawShapes appletObject = new DrawShapes();
83     appletObject.setWidth( width );
84     appletObject.setHeight( height );
85
86     // chama os métodos init e start do applet
87     appletObject.init();
88     appletObject.start();
89
90     // anexa o applet ao centro da janela
91     applicationWindow.getContentPane().add( appletObject );
92
93     // configura o tamanho da janela
94     applicationWindow.setSize( width, height );

```

Fig. 13.9 Criando um aplicativo baseado em GUI a partir de um *applet* (parte 2 de 4).

```

95      // mostrar a janela faz com que todos os componentes
96      // GUI anexados à janela sejam pintados
97      applicationWindow.setVisible( true );
98  }
99
100
101     // classe interna anônima privativa para tratar de eventos de botão
102     private class ButtonHandler implements ActionListener {
103
104         // determina qual botão o usuário pressionou e
105         // configura a opção atual da área de desenho
106         public void actionPerformed( ActionEvent event ) {
107
108             for ( int count = 0; count < choices.length; count++ ) {
109
110                 if ( event.getSource() == choices[ count ] ) {
111                     drawingPanel.setCurrentChoice( count );
112                     break;
113                 }
114             }
115
116         } // fim da classe interna anônima ButtonHandler
117     } // fim da classe DrawShapes
118
119
120     // subclasse de JPanel para permitir que se desenhe em uma área separada
121     class DrawPanel extends JPanel {
122         private int currentChoice = -1; // don't draw first time
123         private int width = 100, height = 100;
124
125         // inicializa largura e altura de DrawPanel
126         public DrawPanel( int newWidth, int newHeight ) {
127
128             width = ( newWidth >= 0 ? newWidth : 100 );
129             height = ( newHeight >= 0 ? newHeight : 100 );
130         }
131
132         // desenha linha, retângulo ou elipse, com base na opção do usuário
133         public void paintComponent( Graphics g ) {
134
135             super.paintComponent( g );
136
137             switch( currentChoice ) {
138
139                 case 0:
140                     g.drawLine( randomX(), randomY(),
141                             randomX(), randomY() );
142                     break;
143
144                 case 1:
145                     g.drawRect( randomX(), randomY(),
146                             randomX(), randomY() );
147                     break;
148
149                 case 2:
150                     g.drawOval( randomX(), randomY(),
151                             randomX(), randomY() );
152                     break;
153             }
154

```

Fig. 13.9 Criando um aplicativo baseado em GUI a partir de um applet (parte 3 de 4).

```

155 } // fim do método paintComponent
156
157 // especifica a forma escolhida atual e pinta novamente
158 public void setCurrentChoice( int choice )
159 {
160     currentChoice = choice;
161     repaint();
162 }
163
164 // obtém uma coordenada x aleatória
165 private int randomX()
166 {
167     return ( int ) ( Math.random() * width );
168 }
169
170 // obtém uma coordenada y aleatória
171 private int randomY()
172 {
173     return ( int ) ( Math.random() * height );
174 }
175
176 } // fim da classe DrawPanel

```

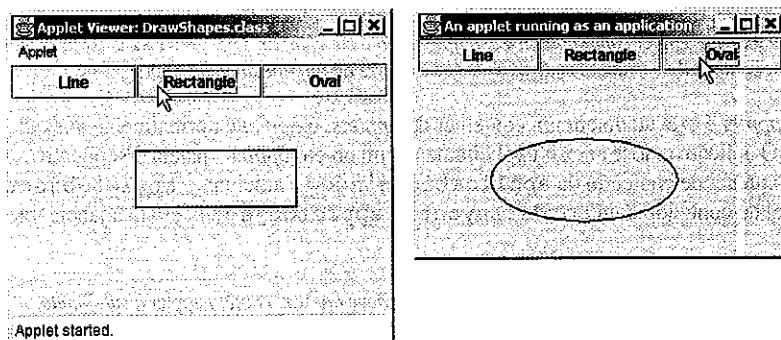


Fig. 13.9 Criando um aplicativo baseado em GUI a partir de um *applet* (parte 4 de 4).

As linhas 65 a 72 determinam a largura e altura iniciais da janela do aplicativo. A condição `if` determina o tamanho do array `args`. Se o número de elementos não for 2, `width` e `height` são configuradas como 300 e 200 por *default*. Caso contrário, as linhas 70 e 71 convertem os argumentos de linha de comando de `Strings` em valores `int` com `parseInt` e os utilizam como `width` e `height`. [Nota: esse programa pressupõe que o usuário digita valores de números inteiros para os argumentos de linha de comando; se não digitar, ocorrerá uma exceção. No Capítulo 14, discutimos como tornar nossos programas mais robustos, lidando com valores impróprios quando eles ocorrem.]

Quando um *applet* é executado, a janela em que ele é executado é fornecida pelo contêiner de *applets* (isto é, o `appletviewer` ou o navegador). Quando o programa é executado como aplicativo, o aplicativo deve criar sua própria janela (se ela se fizer necessária). As linhas 75 e 76 criam o `JFrame` ao qual o programa irá anexar o *applet*. Como ocorre com qualquer `JFrame` que é utilizado como janela principal do aplicativo, você deve fornecer um mecanismo para terminar o aplicativo. As linhas 78 e 79 especificam que o aplicativo deve terminar quando o usuário fecha a janela.

Observação de engenharia de software 13.3



Para executar um *applet* como aplicativo, o aplicativo deve fornecer uma janela em que o *applet* possa ser exibido.

Quando um *applet* é executado em um contêiner de *applets*, o contêiner cria um objeto da classe do *applet* para executar as tarefas do *applet*. Em um aplicativo, os objetos não são criados, a menos que o aplicativo contenha explicitamente instruções para criar objetos. A linha 82 define uma instância da classe de *applet* **DrawShapes**. Repare na chamada ao construtor sem argumentos. Não definimos um construtor na classe **DrawShapes** (classes de *applet* normalmente não definem construtores). Lembre-se de que o compilador gera um construtor *default* para uma classe que não define nenhum construtor. As linhas 83 e 84 chamam os métodos **setWidth** e **setHeight** de **DrawShapes** para validar os valores para **width** e **height** (valores impróprios são configurados como 300 e 200, respectivamente).



Observação de engenharia de software 13.4

Para executar um applet como aplicativo, o aplicativo deve criar uma instância da classe do applet para executar.

Quando um *applet* é executado em um contêiner de *applets*, o contêiner garante que os métodos **init**, **start** e **paint** serão chamados para iniciar a execução do *applet*. Entretanto, esses métodos não são específicos para um aplicativo. Os métodos **init** e **start** não são invocados automaticamente ou exigidos por um aplicativo (o método **paint** faz parte de qualquer janela e será chamado quando necessário para repintar a janela). As linhas 87 e 88 invocam o método **init** do **appletObject** para inicializar o *applet* e configurar sua GUI e depois invocam o método **start**. [Nota: em nosso exemplo, não sobrescrevemos o método **start**. Ele é chamado aqui para simular a seqüência inicial normalmente seguida por um *applet*.]



Observação de engenharia de software 13.5

Ao executar um applet como aplicativo, o aplicativo deve chamar **init** e **start** explicitamente para simular a seqüência normal de chamadas de métodos que ocorre na inicialização do applet.

Quando um *applet* é executado em um contêiner de *applets*, o *applet* é normalmente anexado à janela do contêiner de *applets*. O aplicativo deve anexar explicitamente um objeto *applet* à janela do aplicativo. A linha 91 obtém uma referência ao painel de conteúdo da **applicationWindow** e adiciona o **appletObject** à região *default CENTER* do painel de conteúdo com **BorderLayout**. O **appletObject** ocupará a janela inteira.



Observação de engenharia de software 13.6

Quando se executa um applet como aplicativo, este deve anexar o objeto applet à sua janela.

Por fim, a janela do aplicativo deve ser dimensionada e exibida na tela. A linha 94 configura o tamanho da janela do aplicativo e a linha 98 exibe a janela. Quando um programa Java exibe qualquer janela, todos os componentes anexados à janela recebem chamadas para seus métodos **paint** (se eles forem componentes peso-pesado) ou a seus métodos **paintComponent** (se eles forem componentes peso-leve). Portanto, exibir a janela do aplicativo resulta em uma chamada ao método **paint** do *applet* para completar a seqüência de inicialização normal para o *applet*.

Tente executar esse programa como *applet* e como aplicativo para ver que ele tem a mesma funcionalidade quando executado.

13.8 Utilizando menus com frames

Os *menus* são uma parte integrante das GUIs. Os menus permitem ao usuário realizar ações sem “poluir” desnecessariamente uma interface gráfica com o usuário com componentes GUI em excesso. Em GUIs do Swing, os menus podem ser anexados somente aos objetos das classes que fornecem o método **setJMenuBar**. Duas dessas classes são **JFrame** e **JApplet**. As classes utilizadas para definir menus são **JMenuBar**, **JMenuItem**, **JMenu**, **JCheckBoxMenuItem** e a classe **JRadioButtonMenuItem**.



Observação de aparência e comportamento 13.16

Os menus simplificam as GUIs reduzindo o número de componentes que o usuário visualiza.

A classe **JMenuBar** (uma subclasse de **JComponent**) contém os métodos necessários para gerenciar uma *barra de menus*, que é um contêiner para menus.

A classe **JMenuItem** (uma subclasse de **javax.swing.AbstractButton**) contém os métodos necessários para gerenciar *itens de menu*. O item de menu é um componente GUI dentro de um menu que, quando selecionado, faz com que uma ação seja realizada. Pode-se usar um item de menu para iniciar uma ação ou ser um *submenu* que fornece mais itens de menu a partir dos quais o usuário pode selecionar. Os submenus são úteis para agrupar itens de menu relacionados em um menu.

A classe **JMenu** (uma subclasse de **javax.swing.JMenuItem**) contém os métodos necessários para gerenciar *menus*. Os menus contêm itens de menu e são adicionados a barras de menus ou a outros menus como submenus. Quando se clica num menu, ele se expande para mostrar sua lista de itens. Clicar num item de menu gera um evento de ação.

A classe **JCheckBoxMenuItem** (uma subclasse de **javax.swing.JMenuItem**) contém os métodos necessários para gerenciar itens de menu que podem ser ativados ou desativados. Quando um **JCheckBoxMenuItem** é selecionado, uma marca de verificação aparece à esquerda do item de menu. Quando o **JCheckBoxMenuItem** é selecionado novamente, a marca de verificação à esquerda do item de menu é removida.

A classe **JRadioButtonMenuItem** (uma subclasse de **javax.swing.JMenuItem**) contém os métodos necessários para gerenciar itens de menu que podem ser ativados ou desativados como os **JCheckBoxMenuItem**s. Quando vários **JRadioButtonMenuItem**s são mantidos como parte de um **ButtonGroup**, apenas um item no grupo pode ser selecionado de cada vez. Quando se seleciona um **JRadioButtonMenuItem**, um círculo preenchido aparece à esquerda do item de menu. Quando outro **JRadioButtonMenuItem** é selecionado, o círculo preenchido à esquerda do item do menu previamente selecionado é removido.

O aplicativo da Fig. 13.10 demonstra vários tipos de itens de menu. O programa também demonstra como especificar caracteres especiais, chamados mnemônicos, que podem fornecer acesso rápido a um menu ou a um item de menu pelo teclado. Os mnemônicos podem ser utilizados com objetos de todas as classes que sejam subclasses de **javax.swing.AbstractButton**.

```

1 // Fig. 13.10: MenuTest.java
2 // Demonstrando menus
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class MenuTest extends JFrame {
12     private Color colorValues[] =
13         { Color.black, Color.blue, Color.red, Color.green };
14
15     private JRadioButtonMenuItem colorItems[], fonts[];
16     private JCheckBoxMenuItem styleItems[];
17     private JLabel displayLabel;
18     private ButtonGroup fontGroup, colorGroup;
19     private int style;
20
21     // configura a GUI
22     public MenuTest()
23     {
24         super( "Using JMenus" );
25
26         // configura o menu File e seus itens de menu
27         JMenu fileMenu = new JMenu( "File" );
28         fileMenu.setMnemonic( 'F' );
29

```

Fig. 13.10 Usando JMenus e mnemônicos (parte 1 de 5).

```

30     // configura o item de menu About...
31     JMenuItem aboutItem = new JMenuItem( "About..." );
32     aboutItem.setMnemonic( 'A' );
33
34     aboutItem.addActionListener(
35
36         // classe interna anônima para tratar eventos de item de menu
37         new ActionListener() {
38
39             // exibe diálogo de mensagem quando o usuário seleciona About...
40             public void actionPerformed( ActionEvent event )
41             {
42                 JOptionPane.showMessageDialog( MenuTest.this,
43                     "This is an example\nof using menus",
44                     "About", JOptionPane.PLAIN_MESSAGE );
45             }
46
47         } // fim da classe interna anônima
48
49     ); // fim da chamada para addActionListener
50
51     fileMenu.add( aboutItem );
52
53     // configura o item de menu Exit
54     JMenuItem exitItem = new JMenuItem( "Exit" );
55     exitItem.setMnemonic( 'x' );
56
57     exitItem.addActionListener(
58
59         // classe interna anônima para tratar evento exitItem
60         new ActionListener() {
61
62             // termina o aplicativo quando o usuário clica em exitItem
63             public void actionPerformed( ActionEvent event )
64             {
65                 System.exit( 0 );
66             }
67
68         } // fim da classe interna anônima
69
70     ); // fim da chamada para addActionListener
71
72     fileMenu.add( exitItem );
73
74     // cria barra de menus e a anexa à janela MenuTest
75     JMenuBar bar = new JMenuBar();
76     setJMenuBar( bar );
77     bar.add( fileMenu );
78
79     // cria o menu Format, seus submenus e seus itens de menu
80     JMenu formatMenu = new JMenu( "Format" );
81     formatMenu.setMnemonic( 'r' );
82
83     // cria o submenu Color
84     String colors[] = { "Black", "Blue", "Red", "Green" };
85
86     JMenu colorMenu = new JMenu( "Color" );
87     colorMenu.setMnemonic( 'C' );
88
89     colorItems = new JRadioButtonMenuItem[ colors.length ];

```

Fig. 13.10 Usando JMenus e mnemônicos (parte 2 de 5).

```

90     colorGroup = new ButtonGroup();
91     ItemHandler itemHandler = new ItemHandler();
92
93     // cria itens do menu Color com botões de opção
94     for ( int count = 0; count < colors.length; count++ ) {
95         colorItems[ count ] =
96             new JRadioButtonMenuItem( colors[ count ] );
97
98         colorMenu.add( colorItems[ count ] );
99         colorGroup.add( colorItems[ count ] );
100
101        colorItems[ count ].addActionListener( itemHandler );
102    }
103
104    // seleciona o primeiro item do menu Color
105    colorItems[ 0 ].setSelected( true );
106
107    // adiciona o menu Format à barra de menus
108    formatMenu.add( colorMenu );
109    formatMenu.addSeparator();
110
111    // cria o submenu Font
112    String fontNames[] = { "Serif", "Monospaced", "SansSerif" };
113
114    JMenu fontMenu = new JMenu( "Font" );
115    fontMenu.setMnemonic( 'n' );
116
117    fonts = new JRadioButtonMenuItem[ fontNames.length ];
118    fontGroup = new ButtonGroup();
119
120    // cria itens do menu Font com botões de opção
121    for ( int count = 0; count < fonts.length; count++ ) {
122        fonts[ count ] =
123            new JRadioButtonMenuItem( fontNames[ count ] );
124
125        fontMenu.add( fonts[ count ] );
126        fontGroup.add( fonts[ count ] );
127
128        fonts[ count ].addActionListener( itemHandler );
129    }
130
131    // seleciona o primeiro item do menu Font
132    fonts[ 0 ].setSelected( true );
133
134    fontMenu.addSeparator();
135
136    // configura os itens de estilo do menu
137    String styleNames[] = { "Bold", "Italic" };
138
139    styleItems = new JCheckBoxMenuItem[ styleNames.length ];
140    StyleHandler styleHandler = new StyleHandler();
141
142    // cria os itens de estilo do menu com caixas de marcação
143    for ( int count = 0; count < styleNames.length; count++ ) {
144        styleItems[ count ] =
145            new JCheckBoxMenuItem( styleNames[ count ] );
146
147        fontMenu.add( styleItems[ count ] );
148
149        styleItems[ count ].addItemListener( styleHandler );

```

Fig. 13.10 Usando JMenus e mnemônicos (parte 3 de 5).

```

150     }
151
152     // coloca o menu Font no menu Format
153     formatMenu.add( fontMenu );
154
155     // adiciona o menu Format à barra de menus
156     bar.add( formatMenu );
157
158     // configura rótulo para exibir texto
159     displayLabel = new JLabel(
160         "Sample Text", SwingConstants.CENTER );
161     displayLabel.setForeground( colorValues[ 0 ] );
162     displayLabel.setFont(
163         new Font( "TimesRoman", Font.PLAIN, 72 ) );
164
165     getContentPane().setBackground( Color.cyan );
166     getContentPane().add( displayLabel, BorderLayout.CENTER );
167
168     setSize( 500, 200 );
169     setVisible( true );
170
171 } // fim do construtor
172
173 // executa o aplicativo
174 public static void main( String args[] )
175 {
176     MenuTest application = new MenuTest();
177
178     application.setDefaultCloseOperation(
179         JFrame.EXIT_ON_CLOSE );
180 }
181
182 // classe interna anônima para tratar eventos de ação dos itens de menu
183 private class ItemHandler implements ActionListener {
184
185     // processa as seleções de cor e fonte
186     public void actionPerformed( ActionEvent event )
187     {
188         // processa a seleção de cor
189         for ( int count = 0; count < colorItems.length; count++ )
190
191             if ( colorItems[ count ].isSelected() ) {
192                 displayLabel.setForeground( colorValues[ count ] );
193                 break;
194             }
195
196             // processa a seleção de fonte
197             for ( int count = 0; count < fonts.length; count++ )
198
199                 if ( event.getSource() == fonts[ count ] ) {
200                     displayLabel.setFont( new Font(
201                         fonts[ count ].getText(), style, 72 ) );
202                     break;
203                 }
204
205             repaint();
206         }
207
208     } // fim da classe ItemHandler
209

```

Fig. 13.10 Usando JMenus e mnemônicos (parte 4 de 5).

```

210     // classe interna para tratar eventos dos itens de menu com caixa de marcação
211     private class StyleHandler implements ItemListener {
212
213         // processa seleções de estilo da fonte
214         public void itemStateChanged( ItemEvent e )
215         {
216             style = 0;
217
218             // verifica se negrito foi selecionado
219             if ( styleItems[ 0 ].isSelected() )
220                 style += Font.BOLD;
221
222             // verifica se itálico foi selecionado
223             if ( styleItems[ 1 ].isSelected() )
224                 style += Font.ITALIC;
225
226             displayLabel.setFont( new Font(
227                 displayLabel.getFont().getName(), style, 72 ) );
228
229             repaint();
230         }
231
232     } // fim da classe StyleHandler
233
234 } // fim da classe MenuTest

```

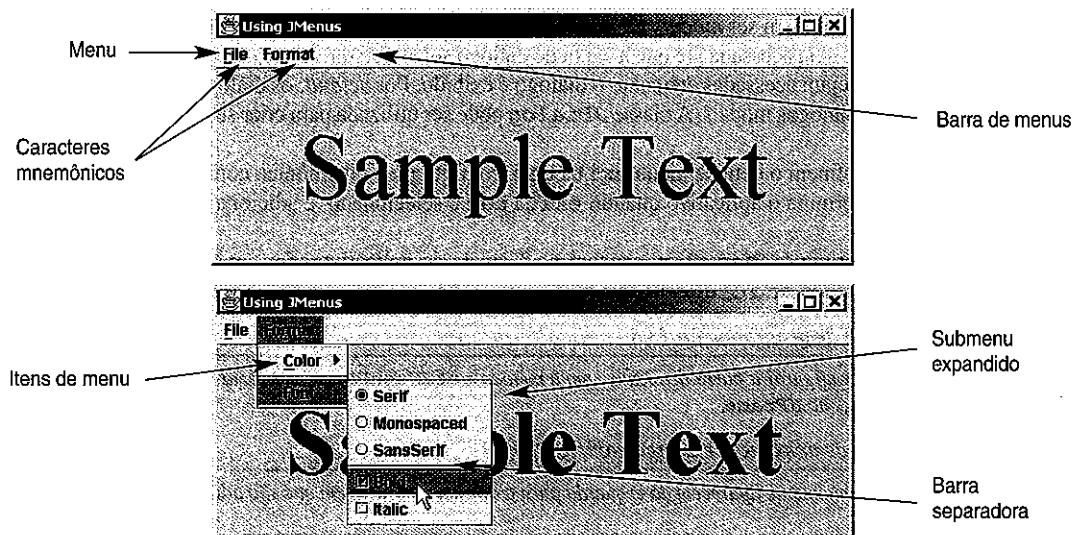


Fig. 13.10 Usando JMenus e mnemônicos (parte 5 de 5).

A classe **MenuTest** (linha 11) é uma classe completamente autocontida – define todos os componentes GUI e tratamento de eventos para os itens de menu. A maior parte do código para esse aplicativo aparece no construtor da classe (linhas 22 a 171).

As linhas 27 a 72 configuram o menu **File** e o anexam à barra de menus. O menu **File** contém um item de menu **About...** que exibe um diálogo de mensagem quando o item de menu é selecionado e um item de menu **Exit** que pode ser selecionado para terminar o aplicativo.

A linha 27 cria **fileMenu** e passa para o construtor o string “**File**” como nome do menu. A linha 28 utiliza o método **setMnemonic** de **AbstractButton** (herdado pela classe **JMenu**) para indicar que **F** é o *mnemônico* para esse menu. Pressionar a tecla **Alt** e a letra **F** abre o menu da mesma maneira que ao se clicar no nome do menu com o mouse. Na GUI, o caractere mnemônico no nome do menu é exibido sublinhado (veja as capturas de tela).



Observação de aparência e comportamento 13.18

Os mnemônicos fornecem acesso rápido pelo teclado para os comandos de menu e de botão.



Observação de aparência e comportamento 13.19

Devem-se utilizar diferentes mnemônicos para cada botão ou item de menu. Normalmente, a primeira letra do rótulo no item de menu ou botão é utilizada como mnemônico. Se vários botões ou itens de menu iniciam com a mesma letra, escolha a próxima letra mais significativa no nome (por exemplo, x comumente é escolhido para um botão ou item de menu chamado **Exit**).

As linhas 31 e 32 definem o **JMenuItem aboutItem** com o nome “**About . . .**” e configuram seu mnemônico como a letra **A**. Esse item de menu é adicionado a **fileMenu** na linha 51. Para acessar o item **About...** pelo teclado, pressione a tecla **Alt** e a letra **F** para abrir o menu **File**, depois pressione **A** para selecionar o item de menu **About....** As linhas 34 a 49 criam um **ActionListener** para processar eventos de ação de **aboutItem**. As linhas 42 a 44 exibem uma caixa de diálogo de mensagem. Na maioria das utilizações anteriores de **showMessageDialog**, o primeiro argumento têm sido **null**. O propósito do primeiro argumento é especificar a janela-pai para a caixa de diálogo. A janela-pai ajuda a determinar onde a caixa de diálogo será exibida. Se a janela-pai for especificada como **null**, a caixa de diálogo normalmente é exibida no centro da tela. Se a janela-pai não for nula, a caixa de diálogo normalmente será exibida centrada sobre a janela-pai especificada. Neste exemplo, o programa especifica a janela-pai com **MenuTest.this** – a referência **this** da classe **MenuTest**. Ao usar a referência **this** em uma classe interna, especificar **this** sozinha faz referência ao objeto da classe interna. Para representar objetos da classe externa com uma referência **this**, qualifique **this** com o nome da classe externa e um operador ponto **(.)**.

As caixas de diálogo podem ser *modais* ou *não-modais*. A caixa de diálogo *modal* não permite que qualquer outra janela do aplicativo seja acessada até que a caixa de diálogo seja liberada. A caixa de diálogo *não-modal* permite que outras janelas sejam acessadas enquanto o diálogo é exibido. Por *default*, os diálogos exibidos com a classe **JOptionPane** são diálogos modais. A classe **JDialog** pode ser utilizada para criar seus próprios diálogos não modais ou modais.

As linhas 54 a 72 definem o item de menu **exitItem**, configuram o mnemônico como **x** e registram um **ActionListener** que termina o aplicativo quando **exitItem** é selecionado, e adicionam **exitItem** ao menu **File**.

As linhas 75 a 77 criam o **JMenuBar**, anexam-no à janela do aplicativo com o método **setMenuBar** de **JFrame** e utilizam o método **add** de **JMenuBar** para anexar o **fileMenu** à barra de menus.



Erro comum de programação 13.5

Esquecer de configurar a barra de menus com o método **setJMenuBar** de **JFrame** resulta na não-exibição da barra de menus no **JFrame**.



Observação de aparência e comportamento 13.19

Os menus normalmente aparecem da esquerda para a direita, na ordem em que são adicionados à **JMenuBar**.

As linhas 80 e 81 criam o menu **formatMenu** e configuram o mnemônico como **r** (**F** não é utilizado porque esse é o mnemônico do menu **File**).

As linhas 86 e 87 criam o menu **colorMenu** (que será um submenu no menu **Format**) e configuram seu mnemônico como **C**. A linha 89 cria o array de **JRadioButtonMenuItem**s **colorItems** que fará referência aos os itens de menu em **colorMenu**. A linha 90 cria o **ButtonGroup** **colorGroup** que assegurará que somente um dos itens de menu do submenu **Color** seja selecionado por vez. A linha 91 define uma instância da classe interna **ItemHandler** (definida nas linhas 183 a 208) que será utilizada para responder às seleções do submenu **Color** e do submenu **Font** (discutidos em breve). A estrutura **for** nas linhas 94 a 102 cria cada **JRadioButtonMenuItem** no array **colorItems**, adiciona cada item de menu ao **colorMenu**, depois ao **colorGroup** e registra o **ActionListener** para cada item de menu.

A linha 105 utiliza o método **setSelected** de **AbstractButton** para selecionar o primeiro elemento do array **colorItems**. A linha 108 adiciona o **colorMenu** como um submenu do **formatMenu**.

Observação de aparência e comportamento 13.20



Adicionar um menu como um item de menu em outro menu torna automaticamente o menu adicionado um submenu. Quando o mouse é posicionado sobre um submenu (ou o mnemônico do submenu é pressionado), o submenu expande para mostrar seus itens de menu.

A linha 109 adiciona uma linha *separadora* para o menu. O separador aparece como uma linha horizontal no menu.

Observação de aparência e comportamento 13.21



Podem-se adicionar separadores a um menu para agrupar logicamente itens de menu.

Observação de aparência e comportamento 13.22



Qualquer componente GUI peso-leve (isto é, um componente que é uma subclasse de `JComponent`) pode ser adicionado a um `JMenu` ou a um `JMenuBar`.

As linhas 114 a 132 criam o submenu `Font` e vários `JRadioButtonMenuItem`s e selecionam o primeiro elemento do array de `JRadioButtonMenuItem`s `fonts`. A linha 139 cria um array `JCheckBoxMenuItem` para representar os itens de menu para especificar os estilos negrito e itálico para as fontes. A linha 140 define uma instância da classe interna `StyleHandler` (definida nas linhas 211 a 232) para responder aos eventos de `JCheckBoxMenuItem`. A estrutura `for` nas linhas 143 a 150 cria cada `JCheckBoxMenuItem`, adiciona cada item de menu ao `fontMenu` e registra o `ItemListener` para cada item de menu. A linha 153 adiciona `fontMenu` como um submenu do `formatMenu`. A linha 156 adiciona o `formatMenu` a `bar`.

As linhas 159 a 163 criam um `JLabel` para o qual a fonte, a cor de fonte e o estilo de fonte são controlados pelo menu `Format`. A cor inicial de primeiro plano é configurada como o primeiro elemento do array `colorValues` (`Color.black`) e a fonte inicial é configurada como `TimesRoman` com o estilo `PLAIN` e tamanho de 72 pontos. A linha 165 configura a cor de fundo do painel de conteúdo da janela para `Color.cyan` e a linha 166 anexa o `JLabel` ao `CENTER` do painel de conteúdo `BorderLayout`.

O método `actionPerformed` da classe `ItemHandler` (linhas 186 a 206) utiliza duas estruturas `for` para determinar qual item de menu de fonte ou de cor gerou o evento e configura a fonte ou a cor do `JLabel display`, respectivamente. A condição `if` na linha 191 utiliza o método `isSelected` de `AbstractButton` para determinar o `JRadioButtonMenuItem` selecionado. A condição `if` na linha 199 utiliza o método `getSource` de `EventSource` para obter uma referência ao `JRadioButtonMenuItem` que gerou o evento. A linha 201 utiliza o método `getText` de `AbstractButton` para obter o nome da fonte a partir do item de menu.

O programa chama o método `itemStateChanged` da classe `StyleHandler` (linhas 214 a 230) se o usuário selecionar um `JCheckBoxMenuItem` no `fontMenu`. As linhas 219 e 223 determinam se um ou ambos os `JCheckBoxMenuItem`s estão selecionados e utilizam seu estado combinado para determinar o novo estilo da fonte.

13.9 Utilizando JPopupMenu

Muitos dos aplicativos atuais de computador usam os chamados *menus pop-up sensíveis ao contexto*. No Swing, esses menus são criados com a classe `JPopupMenu` (uma subclasse de `JComponent`). Esses menus fornecem opções que são específicas para o componente pelo qual o *evento de disparo do pop-up* foi gerado. Na maioria dos sistemas, o evento de disparo do *pop-up* ocorre quando o usuário pressiona e libera o botão direito do mouse.

Observação de aparência e comportamento 13.23



O evento de disparo do pop-up é específico para cada plataforma. Na maioria das plataformas que utilizam um mouse com vários botões, o evento de disparo do pop-up é gerado quando o usuário clica com o botão direito do mouse.

A Fig. 13.8 cria um `JPopupMenu` que permite ao usuário selecionar uma de três cores e alterar a cor de fundo da janela. Quando o usuário clica com o botão direito do mouse sobre o fundo da janela `PopupTest`, é exibido um `JPopupMenu` de cores. Se o usuário clicar em um dos `JRadioButtonMenuItem`s que representa uma cor, o método `actionPerformed` da classe `ItemHandler` altera a cor de fundo do painel de conteúdo da janela.

```

1 // Fig. 13.11: PopupTest.java
2 // Demonstrando JPopupMenu
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class PopupTest extends JFrame {
12
13     private JRadioButtonMenuItem items[];
14     private Color colorValues[] =
15         { Color.blue, Color.yellow, Color.red },
16
17     private JPopupMenu popupMenu;
18
19     // configura a GUI
20     public PopupTest()
21     {
22         super( "Using JPopupMenu" );
23
24         ItemHandler handler = new ItemHandler();
25         String colors[] = { "Blue", "Yellow", "Red" };
26
27         // configura o menu pop-up e seus itens
28         ButtonGroup colorGroup = new ButtonGroup();
29         popupMenu = new JPopupMenu();
30         items = new JRadioButtonMenuItem[ 3 ];
31
32         // constrói cada item de menu e o adiciona ao menu pop-up;
33         // também permite o tratamento de eventos para cada item do menu
34         for ( int count = 0; count < items.length; count++ ) {
35             items[ count ] =
36                 new JRadioButtonMenuItem( colors[ count ] );
37
38             popupMenu.add( items[ count ] );
39             colorGroup.add( items[ count ] );
40
41             items[ count ].addActionListener( handler );
42         }
43
44         getContentPane().setBackground( Color.white );
45
46         // define uma MouseListener para a janela que exibe um JPopupMenu
47         // quando ocorre o evento de acionamento do menu pop-up
48         addMouseListener(
49
50             // classe interna anônima para tratar eventos do mouse
51             new MouseAdapter() {
52
53                 // trata eventos de pressionamento de botões do mouse
54                 public void mousePressed( MouseEvent event )
55                 {
56                     checkForTriggerEvent( event );
57                 }
58
59                 // trata eventos de liberação de botões do mouse
60                 public void mouseReleased( MouseEvent event )

```

Fig. 13.11 Usando um objeto JPopupMenu (parte 1 de 2).

```

61         {
62             checkForTriggerEvent( event );
63         }
64
65         // determina se o evento deve acionar o menu pop-up
66         private void checkForTriggerEvent( MouseEvent event )
67         {
68             if ( event.isPopupTrigger() )
69                 popupMenu.show( event.getComponent(),
70                               event.getX(), event.getY() );
71         }
72
73     } // fim da classe interna anônima
74
75 }; // fim da chamada para addMouseListener
76
77 setSize( 300, 200 );
78 setVisible( true );
79 }
80
81 // executa o aplicativo
82 public static void main( String args[] )
83 {
84     PopupTest application = new PopupTest();
85
86     application.setDefaultCloseOperation(
87         JFrame.EXIT_ON_CLOSE );
88 }
89
90 // classe interna privativa para tratar eventos de itens do menu
91 private class ItemHandler implements ActionListener {
92
93     // processa as seleções de itens do menu
94     public void actionPerformed( ActionEvent event )
95     {
96         // determina qual item do menu foi selecionado
97         for ( int i = 0; i < items.length; i++ )
98             if ( event.getSource() == items[ i ] ) {
99                 getContentPane().setBackground(
100                     colorValues[ i ] );
101                 repaint();
102                 return;
103             }
104         }
105     }
106 } // fim da classe interna privativa ItemHandler
107
108 } // fim da classe PopupTest

```

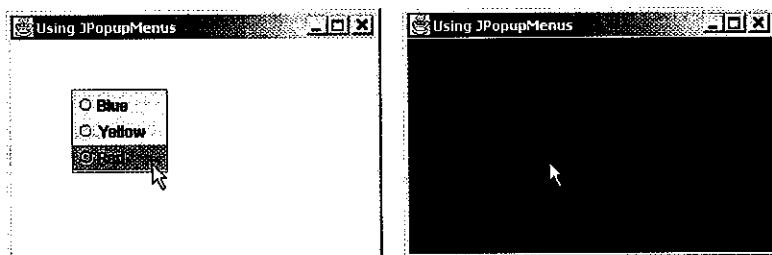


Fig. 13.11 Usando um objeto `JPopupMenu` (parte 2 de 2).

O construtor para a classe `PopupTest` (linhas 20 a 79) define o `JPopupMenu` na linha 29. A estrutura `for` nas linhas 34 a 42 cria `JRadioButtonMenuItem`s para adicionar ao `JPopupMenu`, adiciona-os ao `JPopupMenu` (linha 38), depois ao `ButtonGroup colorGroup` (para manter um `JRadioButtonMenuItem` selecionado por vez) e registra um `ActionListener` para cada item de menu.

As linhas 48 a 75 registram uma instância de uma classe interna anônima que estende `MouseAdapter` para tratar os eventos de mouse da janela de aplicativo. Os métodos `mousePressed` (linhas 54 a 57) e `mouseReleased` (linhas 60 a 63) verificam o evento de disparo de *pop-up*. Cada método chama o método utilitário `private checkForTriggerEvent` (linhas 66 a 71) para determinar se o evento de disparo de *pop-up* ocorreu. O método `isPopupTrigger` de `MouseEvent` devolve `true` se o evento de disparo de *pop-up* ocorreu. Se ocorreu, o método `show` da classe `JPopupMenu` exibe o `JPopupMenu`. O primeiro argumento para o método `show` especifica o *componente de origem*, cuja posição ajuda a determinar onde o `JPopupMenu` aparecerá na tela. Os últimos dois argumentos são as coordenadas *x* e *y* do canto superior esquerdo do componente de origem em que o `JPopupMenu` deve aparecer.



Observação de aparência e comportamento 13.24

Exibir um JPopupMenu para o evento de disparo de pop-up de múltiplos componentes GUI diferentes exige o registro de tratadores de eventos de mouse para verificar o evento de disparo de pop-up para cada um desses componentes GUI.

Quando um item de menu é selecionado do menu *pop-up*, o método `actionPerformed` (linhas 94 a 104) da classe `ItemHandler` (linhas 91 a 106) determina qual `JRadioButtonMenuItem` foi selecionado pelo usuário e depois configura a cor de fundo do painel de conteúdo da janela.

13.10 Aparência e comportamento plugável

O programa que utiliza componentes GUI do Abstract Windowing Toolkit de Java (pacote `java.awt`) assume a aparência e o comportamento da plataforma em que o programa é executado. O programa Java que está sendo executado em um Macintosh se parece com outros programas que são executados em um Macintosh. Um programa Java que está sendo executado no Microsoft Windows se parece com outros programas que são executados no Microsoft Windows. O que está sendo executado em uma plataforma UNIX se parece com outros programas que são executados nessa plataforma UNIX. Isso pode ser desejável, uma vez que permite aos usuários do programa em cada plataforma utilizar os componentes GUI com que eles já estão familiarizados. Entretanto, isso também introduz questões interessantes de portabilidade.



Dica de portabilidade 13.1

Os programas que utilizam componentes GUI do Abstract Windowing Toolkit de Java (pacote `java.awt`) assumem a aparência e o comportamento da plataforma em que eles são executados.



Dica de portabilidade 13.2

Os componentes GUI em cada plataforma têm aparências diferentes que podem exigir quantidades diferentes de espaço para serem exibidos. Isso pode alterar o layout e o alinhamento dos componentes GUI.



Dica de portabilidade 13.3

Os componentes GUI em cada plataforma têm funcionalidade default diferente (por exemplo, algumas plataformas permitem que um botão com foco seja “pressionado” com a barra de espaço e outras não).

Os componentes GUI peso-leve do Swing eliminam muitas dessas questões fornecendo funcionalidade uniforme de uma plataforma para outra e definindo aparência e comportamento uniformes para diversas plataformas (que é conhecida como aparência de metal – *metal look-and-feel*). O Swing também fornece a flexibilidade para personalizar a aparência e o comportamento com o estilo do Microsoft Windows ou com o estilo do Motif (UNIX).

O programa da Fig. 13.12 demonstra como alterar a aparência e o comportamento de uma GUI do Swing. O programa cria vários componentes GUI para que você possa ver a mudança na aparência e no comportamento dos diversos componentes GUI ao mesmo tempo. A primeira janela de saída mostra a aparência e o padrão de metal, a segunda janela de saída mostra a aparência e o comportamento do Motif e a terceira janela de saída mostra a aparência e o comportamento do Windows.

```

1 // Fig. 13.12: LookAndFeelDemo.java
2 // Mudando a aparência e o comportamento.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class LookAndFeelDemo extends JFrame {
12
13     private String strings[] = { "Metal", "Motif", "Windows" };
14     private UIManager.LookAndFeelInfo looks[];
15     private JRadioButton radio[];
16     private ButtonGroup group;
17     private JButton button;
18     private JLabel label;
19     private JComboBox comboBox;
20
21     // configura a GUI
22     public LookAndFeelDemo()
23     {
24         super( "Look and Feel Demo" );
25
26         Container container = getContentPane();
27
28         // configura o painel para a região NORTH de BorderLayout
29         JPanel northPanel = new JPanel();
30         northPanel.setLayout( new GridLayout( 3, 1, 0, 5 ) );
31
32         // configura o rótulo para o painel NORTH
33         label = new JLabel( "This is a Metal look-and-feel",
34             SwingConstants.CENTER );
35         northPanel.add( label );
36
37         // configura o botão para o painel NORTH
38         button = new JButton( "JButton" );
39         northPanel.add( button );
40
41         // configura a caixa de combinação para o painel NORTH
42         comboBox = new JComboBox( strings );
43         northPanel.add( comboBox );
44
45         // anexa o painel NORTH ao painel de conteúdo
46         container.add( northPanel, BorderLayout.NORTH );
47
48         // cria o array para os botões de opção
49         radio = new JRadioButton[ strings.length ];
50
51         // configura o painel para a região SOUTH de BorderLayout
52         JPanel southPanel = new JPanel();
53         southPanel.setLayout(
54             new GridLayout( 1, radio.length ) );
55
56         // configura os botões de opção para o painel SOUTH
57         group = new ButtonGroup();
58         ItemHandler handler = new ItemHandler();
59
60         for ( int count = 0; count < radio.length; count++ ) {

```

Fig. 13.12 Mudando a aparência e o comportamento de uma GUI baseada em Swing (parte 1 de 3).

```

61     radio[ count ] = new JRadioButton( strings[ count ] );
62     radio[ count ].addItemListener( handler );
63     group.add( radio[ count ] );
64     southPanel.add( radio[ count ] );
65   }
66
67   // anexa o painel SOUTH ao painel de conteúdo
68   container.add( southPanel, BorderLayout.SOUTH );
69
70   // obtém as informações sobre aparência e comportamento instaladas
71   looks = UIManager.getInstalledLookAndFeels();
72
73   setSize( 300, 200 );
74   setVisible( true );
75
76   radio[ 0 ].setSelected( true );
77 }
78
79 // usa UIManager para mudar aparência e comportamento da GUI
80 private void changeTheLookAndFeel( int value )
81 {
82   // muda aparência e comportamento
83   try {
84     UIManager.setLookAndFeel(
85       looks[ value ].getClassName() );
86     SwingUtilities.updateComponentTreeUI( this );
87   }
88
89   // processa problemas com a mudança de aparência e comportamento
90   catch ( Exception exception ) {
91     exception.printStackTrace();
92   }
93 }
94
95 // executa o aplicativo
96 public static void main( String args[] )
97 {
98   LookAndFeelDemo application = new LookAndFeelDemo();
99
100  application.setDefaultCloseOperation(
101    JFrame.EXIT_ON_CLOSE );
102 }
103
104 // classe interna privativa para tratar eventos de botões de opção
105 private class ItemHandler implements ItemListener {
106
107   // processa a seleção de aparência e comportamento pelo usuário
108   public void itemStateChanged( ItemEvent event )
109   {
110     for ( int count = 0; count < radio.length; count++ )
111
112       if ( radio[ count ].isSelected() ) {
113         label.setText( "This is a " +
114           strings[ count ] + " look-and-feel" );
115         comboBox.setSelectedIndex( count );
116
117         changeTheLookAndFeel( count );
118       }
119   }
120 }
```

Fig. 13.12 Mudando a aparência e o comportamento de uma GUI baseada em Swing (parte 2 de 3).

```

121     } // fim da classe interna privada ItemHandler
122
123 } // fim da classe LookAndFeelDemo

```

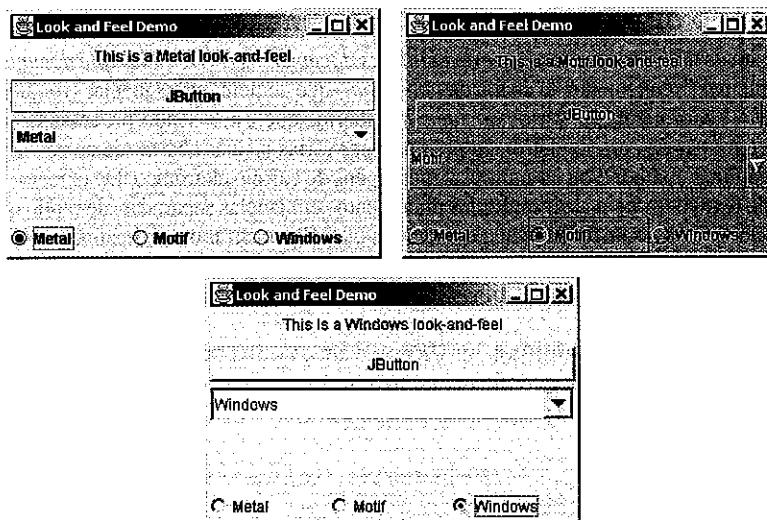


Fig. 13.12 Mudando a aparência e o comportamento de uma GUI baseada em Swing (parte 3 de 3).

Todos os componentes GUI e o tratamento de eventos deste exemplo foram abordados anteriormente, de modo que nos concentraremos nesse exemplo apenas no mecanismo para alterar a aparência e o comportamento.

A classe **UIManager** (pacote **javax.swing**) contém a classe interna **public static LookAndFeelInfo** que é utilizada para manter as informações sobre aparência e comportamento. A linha 14 declara um *array* do tipo **UIManager.LookAndFeelInfo** (repare na sintaxe utilizada para acessar a classe interna **LookAndFeelInfo**). A linha 71 utiliza o método **static getInstalledLookAndFeels** da classe **UIManager** para obter o *array* de objetos **UIManager.LookAndFeelInfo** que descrevem a aparência e o comportamento instalados.

Dica de desempenho 13.1



Cada aparência e comportamento é representado por uma classe Java. O método **getInstalledLookAndFeels** de **UIManager** não carrega cada classe. Em vez disso, fornece acesso aos nomes de aparência e comportamento de modo que uma escolha de aparência e comportamento possa ser feita (presumivelmente uma vez na inicialização do programa). Isso reduz a sobrecarga de se carregar classes adicionais que o programa não vai usar.

O método utilitário **changeTheLookAndFeel** (linhas 80 a 93) é chamado pelo tratador de eventos (definido na classe interna **private ItemHandler** nas linhas 105 a 121) para os **JRadioButtons** na parte inferior da interface com o usuário. O tratador de eventos passa um inteiro que representa o elemento no *array* **looks** que deve ser utilizado para alterar a aparência e o comportamento. As linhas 84 e 85 utilizam o método **static setLookAndFeel** da classe **UIManager** para alterar a aparência e o comportamento. O método **getClassName** da classe **UIManager.LookAndFeelInfo** determina o nome da classe de aparência e comportamento que corresponde ao **UIManager.LookAndFeelInfo**. Se a classe de aparência e comportamento ainda não estiver carregada, ela será carregada como parte da chamada a **setLookAndFeel**. A linha 86 utiliza o método **static updateComponentTreeUI** da classe **SwingUtilities** (pacote **javax.swing**) para alterar a aparência e o comportamento de cada componente anexado a seu argumento (essa instância da classe **LookAndFeelDemo**) para a nova aparência e o novo comportamento.

As duas instruções precedentes aparecem em um bloco especial de código denominado *bloco try*. Esse código faz parte do *mecanismo de tratamento de exceções* discutido em detalhes no próximo capítulo. Esse código é exigido no caso de as linhas 84 e 85 tentarem alterar a aparência e o comportamento para uma aparência e um comportamento que não existe. As linhas 90 a 92 completam o mecanismo de tratamento de exceções com um *tratador*

`catch` que simplesmente processa esse problema (se ocorrer) imprimindo uma mensagem de erro na linha de comando.

13.11 Utilizando `JDesktopPane` e `JInternalFrame`

Muitos aplicativos atuais utilizam uma *interface de múltiplos documentos* (*MDI – multiple document interface*) [isto é, uma janela principal (freqüentemente chamada *janela-pai*) que contém outras janelas (freqüentemente chamadas de *janelas-filhas*)] para gerenciar vários *documentos* abertos que estão sendo processados em paralelo. Por exemplo, muitos programas de correio eletrônico permitem ter várias janelas abertas de correio eletrônico ao mesmo tempo para possibilitar compor e/ou ler múltiplas mensagens de correio eletrônico. De maneira semelhante, muitos processadores de texto permitem abrir múltiplos documentos em janelas separadas, de modo que o usuário possa alternar entre os documentos sem ter de fechar o documento atual para abrir outro documento. O programa da Fig. 13.13 demonstra as classes `JDesktopPane` e `JInternalFrame` do Swing, que fornecem suporte para criar interfaces de múltiplos documentos. As janelas-filhas simplesmente exibem uma imagem da capa original deste livro em inglês.

As linhas 20 a 27 definem um `JMenuBar`, um `JMenu` e um `JMenuItem`, adicionam o `JMenuItem` a `JMenu`, adicionam o `JMenu` a `JMenuBar` e configuram o `JMenuBar` para a janela do aplicativo. Quando o usuário seleciona o `JMenuItem newFrame`, o programa cria e exibe um novo `JInternalFrame`.

```

1 // Fig. 13.13: DesktopTest.java
2 // Demonstrando JDesktopPane.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class DesktopTest extends JFrame {
12     private JDesktopPane theDesktop;
13
14     // configura a GUI
15     public DesktopTest()
16     {
17         super( "Using a JDesktopPane" );
18
19         // cria uma barra de menus, um menu e itens de menu
20         JMenuBar bar = new JMenuBar();
21         JMenu addMenu = new JMenu( "Add" );
22         JMenuItem newFrame = new JMenuItem( "Internal Frame" );
23
24         addMenu.add( newFrame );
25         bar.add( addMenu );
26
27         setJMenuBar( bar );
28
29         // configura a área de trabalho
30         theDesktop = new JDesktopPane();
31         getContentPane().add( theDesktop );
32
33         // configura ouvinte para o item de menu newFrame
34         newFrame.addActionListener(
35
36             // classe interna anônima para tratar eventos de item de menu
37             new ActionListener() {

```

Fig. 13.13 Criando uma interface com múltiplos documentos (parte 1 de 3).

```

38
39         // exibe nova janela interna
40         public void actionPerformed( ActionEvent event ) {
41
42             // cria frame interna
43             JInternalFrame frame = new JInternalFrame(
44                 "Internal Frame", true, true, true, true );
45
46             // anexa painel ao painel de conteúdo da frame interna
47             Container container = frame.getContentPane();
48             MyJPanel panel = new MyJPanel();
49             container.add( panel, BorderLayout.CENTER );
50
51             // configura o tamanho da frame interna com o tamanho de seu conteúdo
52             frame.pack();
53
54             // anexa a frame interna à área de trabalho e a exibe
55             theDesktop.add( frame );
56             frame.setVisible( true );
57         }
58
59     } // fim da classe interna anônima
60
61 }; // fim da chamada para addActionListener
62
63 setSize( 600, 440 );
64 setVisible( true );
65
66 } // fim do construtor
67
68 // executa o aplicativo
69 public static void main( String args[] )
70 {
71     DesktopTest application = new DesktopTest();
72
73     application.setDefaultCloseOperation(
74         JFrame.EXIT_ON_CLOSE );
75 }
76
77 } // fim da classe DesktopTest
78
79 // classe para exibir um ImageIcon em um painel
80 class MyJPanel extends JPanel {
81     private ImageIcon imageIcon;
82
83     // carrega imagem
84     public MyJPanel()
85     {
86         imageIcon = new ImageIcon( "jhtp4.png" );
87     }
88
89     // exibe o imageIcon no painel
90     public void paintComponent( Graphics g )
91     {
92         // chama o método paintComponent da superclasse
93         super.paintComponent( g );
94
95         // exibe ícone

```

Fig. 13.13 Criando uma interface com múltiplos documentos (parte 2 de 3).

```

96     ImageIcon.paintIcon( this, g, 0, 0 );
97 }
98
99 // devolve dimensões da imagem
100 public Dimension getPreferredSize()
101 {
102     return new Dimension( ImageIcon.getIconWidth(),
103                         ImageIcon.getIconHeight() );
104 }
105
106 } // fim da classe MyJPanel

```

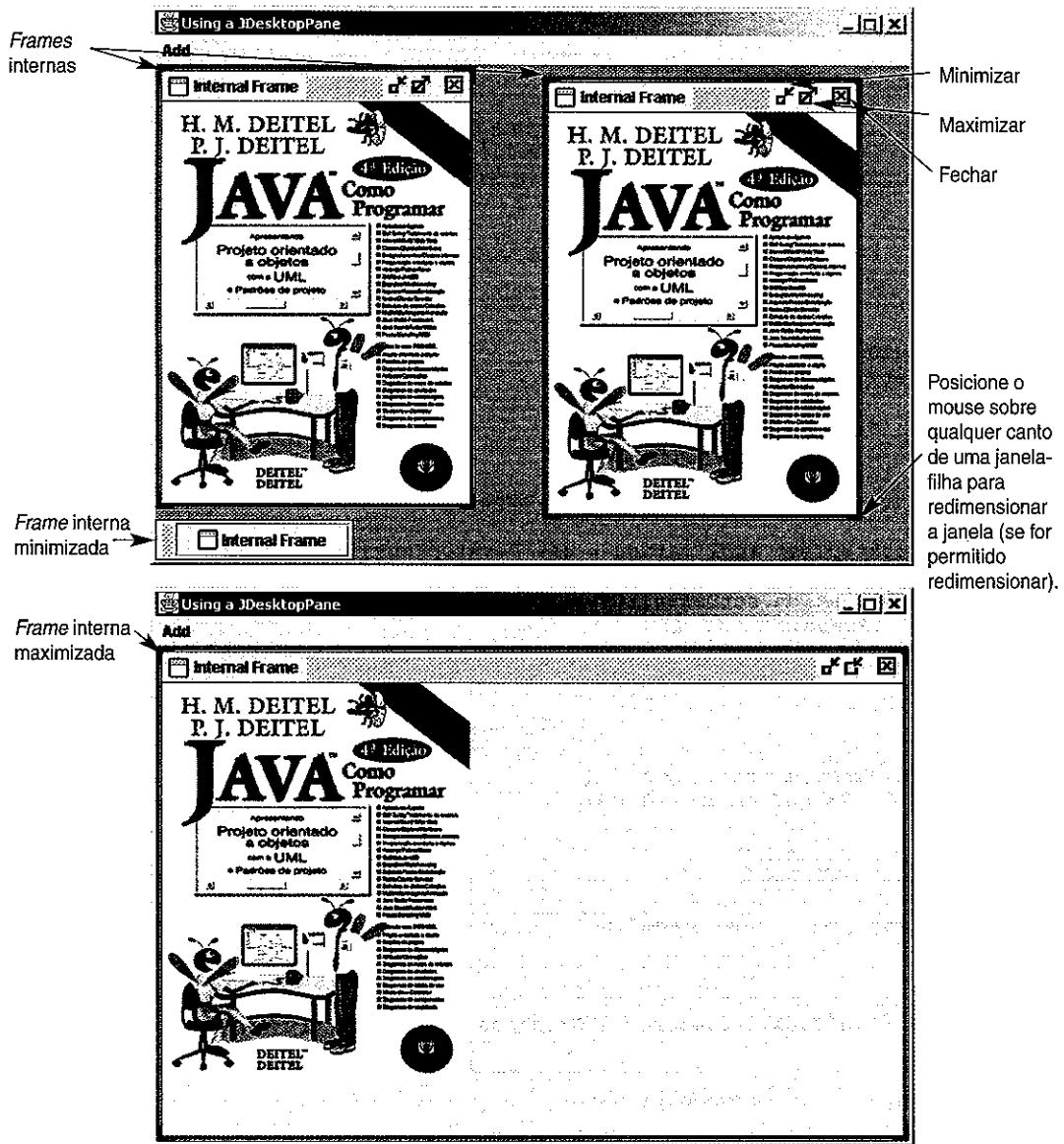


Fig. 13.13 Criando uma interface com múltiplos documentos (parte 3 de 3).

A linha 30 cria a referência `JDesktopPane` (pacote `javax.swing`) `theDesktop` e a atribui a um novo objeto `JDesktopPane`. Utiliza-se o objeto `JDesktopPane` para gerenciar as janelas-filhas `JInternalFrame` que serão exibidas na `JDesktopPane`. A linha 31 adiciona o `JDesktopPane` ao painel de conteúdo da janela do aplicativo.

As linhas 34 a 61 registram uma instância de uma classe interna anônima que implementa `ActionListener` para tratar o evento quando o usuário seleciona o item de menu `newFrame`. Quando ocorre o evento, o método `actionPerformed` (linhas 40 a 57) cria um objeto `JInternalFrame` com as linhas 43 e 44. O construtor `JInternalFrame` utilizado aqui exige cinco argumentos – um `String` para a barra de título da janela interna, um `boolean` para indicar se a *frame* interna pode ser redimensionada pelo usuário, um `boolean` para indicar se a *frame* interna pode ser fechada pelo usuário, um `boolean` para indicar se ela pode ser maximizada pelo usuário e um `boolean` para indicar se pode ser minimizada pelo usuário. Para cada um dos argumentos booleanos um valor `true` indica que a operação deve ser permitida.

Como ocorre com `JFrames` e `JApplets`, um `JInternalFrame` tem um painel de conteúdo ao qual os componentes GUI podem ser anexados. A linha 47 obtém uma referência para o painel de conteúdo do `JInternalFrame`. A linha 48 cria uma instância de nossa classe `My JPanel` (definida nas linhas 80 a 106) que é adicionada ao painel de conteúdo do `JInternalFrame` na linha 49.

A linha 52 usa o método `pack` de `JInternalFrame` para configurar o tamanho da janela-filha. O método `pack` usa os tamanhos preferidos dos componentes no painel de conteúdo para determinar o tamanho da janela. A classe `My JPanel` define o método `getPreferredSize` para especificar o tamanho específico do painel. A linha 55 adiciona o `JInternalFrame` ao `JDesktopPane` e a linha 56 exibe o `JInternalFrame`.

As classes `JInternalFrame` e `JDesktopPane` fornecem muitos métodos para gerenciar janelas-filhas. Consulte a documentação *on-line* da API para ver uma lista completa desses métodos.

13.12 Gerenciadores de leiaute

No capítulo precedente, apresentamos três gerenciadores de leiaute – `FlowLayout`, `BorderLayout` e `GridLayout`. Esta seção apresenta três gerenciadores adicionais de leiaute (resumidos na Fig. 13.14). Discutiremos esses gerenciadores de leiaute nas seções que se seguem.

Gerenciador de leiaute	Descrição
<code>BoxLayout</code>	Gerenciador de leiaute que permite que os componentes GUI sejam organizados da esquerda para a direita ou de cima para baixo em um contêiner. A classe <code>Box</code> define um contêiner com <code>BoxLayout</code> como seu gerenciador <i>default</i> de leiaute e fornece métodos estáticos para criar um <code>Box</code> com um <code>BoxLayout</code> horizontal ou vertical.
<code>CardLayout</code>	Gerenciador de leiaute que empilha componentes como uma pilha de cartas. Se um componente na pilha for um contêiner, você pode utilizar qualquer gerenciador de leiaute. Somente o componente “na parte superior” da pilha é visível.
<code>GridBagLayout</code>	Gerenciador de leiaute semelhante a <code>GridLayout</code> . Diferente de <code>GridLayout</code> , o tamanho de cada componente pode variar e os componentes podem ser adicionados em qualquer ordem.

Fig. 13.14 Gerenciadores de leiaute adicionais.

13.13 O gerenciador de leiaute `BoxLayout`

O gerenciador de leiaute `BoxLayout` organiza componentes GUI horizontalmente ao longo do eixo *x* ou verticalmente ao longo do eixo *y* de um contêiner. O programa da Fig. 13.15 demonstra `BoxLayout` e a classe contêiner `Box` que utiliza `BoxLayout` como seu gerenciador de leiaute *default*.

No construtor para a classe `BoxLayoutDemo`, as linhas 19 e 20 obtêm uma referência ao painel de conteúdo e configuram seu leiaute como `BorderLayout` com espaçamento de 30 pixels na horizontal e de 30 pixels na ver-

tical entre os componentes. O espaço é para ajudar a isolar cada um dos contêineres com **BoxLayout** nesse exemplo.

As linhas 23 a 28 definem um *array* de referências ao contêiner **Box** chamado **boxes** e inicializam cada elemento do *array* com objetos **Box**. Os elementos 0 e 2 do *array* são inicializados com o método **static createHorizontalBox** da classe **Box**, que retorna um contêiner **Box** com um **BoxLayout** horizontal (os componentes GUI são organizados da esquerda para a direita). Os elementos 1 e 3 do *array* são inicializados com o método **static createVerticalBox** da classe **Box**, que retorna um contêiner **Box** com um **BoxLayout** vertical (os componentes GUI são organizados de cima para baixo).

```

1 // Fig. 13.15: BoxLayoutDemo.java
2 // Demonstrando BoxLayout.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class BoxLayoutDemo extends JFrame {
12
13     // configura a GUI
14     public BoxLayoutDemo()
15     {
16         super( "Demonstrating BoxLayout" );
17         final int SIZE = 3;
18
19         Container container = getContentPane();
20         container.setLayout( new BorderLayout( 30, 30 ) );
21
22         // cria contêineres Box com BoxLayout
23         Box boxes[] = new Box[ 4 ];
24
25         boxes[ 0 ] = Box.createHorizontalBox();
26         boxes[ 1 ] = Box.createVerticalBox();
27         boxes[ 2 ] = Box.createHorizontalBox();
28         boxes[ 3 ] = Box.createVerticalBox();
29
30         // adiciona botões a boxes[ 0 ]
31         for ( int count = 0; count < SIZE; count++ )
32             boxes[ 0 ].add( new JButton( "boxes[0]: " + count ) );
33
34         // cria um suporte e adiciona botões a boxes[ 1 ]
35         for ( int count = 0; count < SIZE; count++ ) {
36             boxes[ 1 ].add( Box.createVerticalStrut( 25 ) );
37             boxes[ 1 ].add( new JButton( "boxes[1]: " + count ) );
38         }
39
40         // cria cola horizontal e adiciona botões a boxes[ 2 ]
41         for ( int count = 0; count < SIZE; count++ ) {
42             boxes[ 2 ].add( Box.createHorizontalGlue() );
43             boxes[ 2 ].add( new JButton( "boxes[2]: " + count ) );
44         }
45

```

Fig. 13.15 Demonstrando o gerenciador de layout **BoxLayout** (parte 1 de 3).

```

46     // cria área rígida e adiciona botões a boxes[ 3 ]
47     for ( int count = 0; count < SIZE; count++ ) {
48         boxes[ 3 ].add(
49             Box.createRigidArea( new Dimension( 12, 8 ) ) );
50         boxes[ 3 ].add( new JButton( "boxes[3]: " + count ) );
51     }
52
53     // cria colas vertical e adiciona botões ao painel
54     JPanel panel = new JPanel();
55     panel.setLayout(
56         new BoxLayout( panel, BoxLayout.Y_AXIS ) );
57
58     for ( int count = 0; count < SIZE; count++ ) {
59         panel.add( Box.createGlue() );
60         panel.add( new JButton( "panel: " + count ) );
61     }
62
63     // coloca os painéis na frame
64     container.add( boxes[ 0 ], BorderLayout.NORTH );
65     container.add( boxes[ 1 ], BorderLayout.EAST );
66     container.add( boxes[ 2 ], BorderLayout.SOUTH );
67     container.add( boxes[ 3 ], BorderLayout.WEST );
68     container.add( panel, BorderLayout.CENTER );
69
70     setSize( 350, 300 );
71     setVisible( true );
72
73 } // fim do construtor
74
75 // executa o aplicativo
76 public static void main( String args[] )
77 {
78     BoxLayoutDemo application = new BoxLayoutDemo();
79
80     application.setDefaultCloseOperation(
81         JFrame.EXIT_ON_CLOSE );
82 }
83
84 } // fim da classe BoxLayoutDemo

```

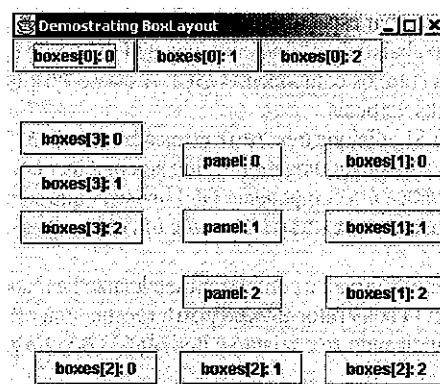


Fig. 13.15 Demonstrando o gerenciador de layout BoxLayout (parte 2 de 3).

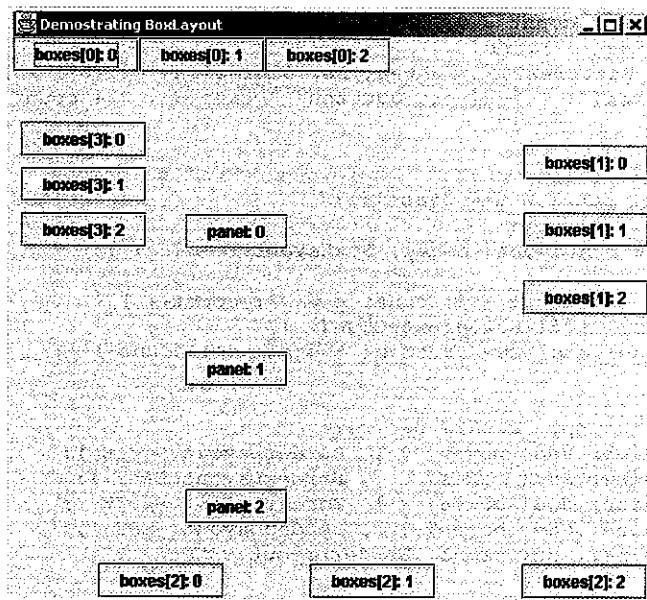


Fig. 13.15 Demonstrando o gerenciador de leiaute `BoxLayout` (parte 3 de 3).

A estrutura `for` nas linhas 31 e 32 adiciona três `JButtons` a `boxes[0]` (uma `Box` horizontal). A estrutura `for` nas linhas 35 a 38 adiciona três `JButtons` a `boxes[1]` (uma `Box` vertical). Antes de adicionar cada botão, a linha 36 adiciona um *suporte vertical* ao contêiner com o método `static createVerticalStrut` da classe `Box`. O suporte vertical é um componente GUI invisível que tem uma altura fixa em *pixels* e é utilizado para garantir uma quantidade fixa de espaço entre os componentes GUI. O argumento para o método `createVerticalStrut` determina a altura do suporte em *pixels*. A classe `Box` também define o método `createHorizontalStrut` para `BoxLayouts` horizontais.

A estrutura `for` nas linhas 41 a 44 adiciona três `JButtons` a `boxes[2]` (uma `Box` horizontal). Antes de adicionar cada botão, a linha 42 adiciona a *cola horizontal* ao contêiner com o método `static createHorizontalGlue` da classe `Box`. A cola horizontal é um componente GUI invisível que pode ser utilizado entre componentes GUI de tamanho fixo para ocupar espaço adicional. Normalmente, o espaço extra aparece à direita do último componente GUI horizontal ou abaixo do último componente GUI vertical em um `BoxLayout`. A cola permite que seja colocado espaço extra entre os componentes GUI. A classe `Box` também define o método `createVerticalGlue` para `BoxLayouts` verticais.

A estrutura `for` nas linhas 47 a 51 adiciona três `JButtons` a `boxes[3]` (um `Box` vertical). Antes de adicionar cada botão, as linhas 48 e 49 adicionam uma *área rígida* ao contêiner com o método `static createRigidArea` da classe `Box`. A área rígida é um componente GUI invisível que sempre tem uma largura e altura fixas em *pixels*. O argumento para o método `createRigidArea` é um objeto `Dimension` que especifica a largura e altura da área rígida.

As linhas 54 a 56 criam um objeto `JPanel` e configuram seu leiaute na maneira convencional utilizando o método `setLayout` de `Container`. O construtor `BoxLayout` recebe uma referência ao contêiner pelo qual ele controla o leiaute e uma constante indicando se o leiaute é horizontal (`BoxLayout.X_AXIS`) ou vertical (`BoxLayout.Y_AXIS`).

A estrutura `for` nas linhas 58 a 61 adiciona três `JButtons` ao `panel`. Antes de adicionar cada botão, a linha 59 adiciona um componente de cola ao contêiner com o método `static createGlue` da classe `Box`. Este componente expande ou contrai de acordo com o tamanho da `Box`.

Os contêineres `Box` e o `JPanel` são anexados ao painel de conteúdo `BorderLayout` nas linhas 64 a 68. Tente executar o aplicativo. Quando a janela aparecer, redimensione a janela para ver como os componentes cola, suporte e área rígida afetam o leiaute em cada contêiner.

13.14 O gerenciador de leiaute CardLayout

O gerenciador de leiaute **CardLayout** organiza componentes como em uma “pilha de cartas” na qual somente a carta superior é visível. Qualquer carta pode ser colocada na parte superior da pilha, a qualquer momento, utilizando métodos da classe **CardLayout**. Cada carta é normalmente um contêiner, como um painel, e cada carta pode utilizar qualquer gerenciador de leiaute. A classe **CardLayout** herda de **Object** e implementa a interface **LayoutManager2**.

O programa da Fig. 13.16 cria cinco painéis. O JPanel **deck** utiliza o gerenciador de leiaute **CardLayout** para controlar a carta que é exibida. Os JPanels **card1**, **card2** e **card3** são utilizados como cartas individuais em **deck**. O JPanel **buttons** contém quatro botões (com rótulos **First card**, **Next card**, **Previous card** e **Last card**) que permitem ao usuário manipular a pilha de cartas. Quando o usuário clicar no botão **First card**, a primeira carta em **deck** (isto é, **card1**) é exibida. Quando o usuário clicar no botão **Last card**, é exibida a última carta (isto é, **card3**) em **deck**. Toda vez que o usuário clicar no botão **Previous card**, a carta anterior em **deck** é exibida. Toda vez que o usuário clicar no botão **Next card**, é exibida a próxima carta em **deck**. Clicar no botão **Previous card** ou no botão **Next card** repetidamente permite ao usuário circular pelo **deck** de cartas. A classe de aplicativo **CardDeck** implementa **ActionListener**, assim os eventos de ação gerados pelos JButtons no JPanel **buttons** são tratados pelo aplicativo em seu método **actionPerformed**.

A classe **CardDeck** declara uma referência do tipo **CardLayout** chamada de **cardManager** (linha 13). Utiliza-se essa referência para invocar os métodos de **CardLayout** que manipulam as cartas na pilha.

```

1 // Fig. 13.16: CardDeck.java
2 // Demonstrando CardLayout.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class CardDeck extends JFrame implements ActionListener {
12
13     private CardLayout cardManager;
14     private JPanel deck;
15     private JButton controls[];
16     private String names[] = { "First card", "Next card",
17         "Previous card", "Last card" };
18
19     // configura a GUI
20     public CardDeck()
21     {
22         super( "CardLayout" );
23
24         Container container = getContentPane();
25
26         // cria o JPanel com CardLayout
27         deck = new JPanel();
28         cardManager = new CardLayout();
29         deck.setLayout( cardManager );
30
31         // configura card1 e a adiciona ao JPanel deck
32         JLabel label1 =
33             new JLabel( "card one", SwingConstants.CENTER );
34         JPanel card1 = new JPanel();
35         card1.add( label1 );
36         deck.add( card1, label1.getText() );    // add card to deck
37

```

Fig. 13.16 Demonstrando o gerenciador de leiaute **CardLayout** (parte 1 de 3).

```

38     // configura card2 e a adiciona ao JPanel deck
39     JLabel label2 =
40         new JLabel( "card two", SwingConstants.CENTER );
41     JPanel card2 = new JPanel();
42     card2.setBackground( Color.yellow );
43     card2.add( label2 );
44     deck.add( card2, label2.getText() );    // add card to deck
45
46     // configura card3 e a adiciona ao JPanel deck
47     JLabel label3 = new JLabel( "card three" );
48     JPanel card3 = new JPanel();
49     card3.setLayout( new BorderLayout() );
50     card3.add( new JButton( "North" ), BorderLayout.NORTH );
51     card3.add( new JButton( "West" ), BorderLayout.WEST );
52     card3.add( new JButton( "East" ), BorderLayout.EAST );
53     card3.add( new JButton( "South" ), BorderLayout.SOUTH );
54     card3.add( label3, BorderLayout.CENTER );
55     deck.add( card3, label3.getText() );    // adiciona carta ao baralho
56
57     // cria e posiciona os botões que vão controlar o baralho
58     JPanel buttons = new JPanel();
59     buttons.setLayout( new GridLayout( 2, 2 ) );
60     controls = new JButton[ names.length ];
61
62     for ( int count = 0; count < controls.length; count++ ) {
63         controls[ count ] = new JButton( names[ count ] );
64         controls[ count ].addActionListener( this );
65         buttons.add( controls[ count ] );
66     }
67
68     // adiciona o JPanel do baralho e os botões JPanel ao applet
69     container.add( buttons, BorderLayout.WEST );
70     container.add( deck, BorderLayout.EAST );
71
72     setSize( 450, 200 );
73     setVisible( true );
74
75 } // fim do construtor
76
77 // trata eventos de botão trocando as cartas
78 public void actionPerformed( ActionEvent event )
79 {
80     // mostra a primeira carta
81     if ( event.getSource() == controls[ 0 ] )
82         cardManager.first( deck );
83
84     // mostra a carta seguinte
85     else if ( event.getSource() == controls[ 1 ] )
86         cardManager.next( deck );
87
88     // mostra a carta anterior
89     else if ( event.getSource() == controls[ 2 ] )
90         cardManager.previous( deck );
91
92     // mostra a última carta
93     else if ( event.getSource() == controls[ 3 ] )
94         cardManager.last( deck );
95 }
96

```

Fig. 13.16 Demonstrando o gerenciador de layout CardLayout (parte 2 de 3).

```

97     // executa o aplicativo
98     public static void main( String args[] )
99     {
100         CardDeck cardDeckDemo = new CardDeck();
101         cardDeckDemo.setDefaultCloseOperation(
102             JFrame.EXIT_ON_CLOSE );
103     }
104 }
105 }
106 } // fim da classe CardDeck

```

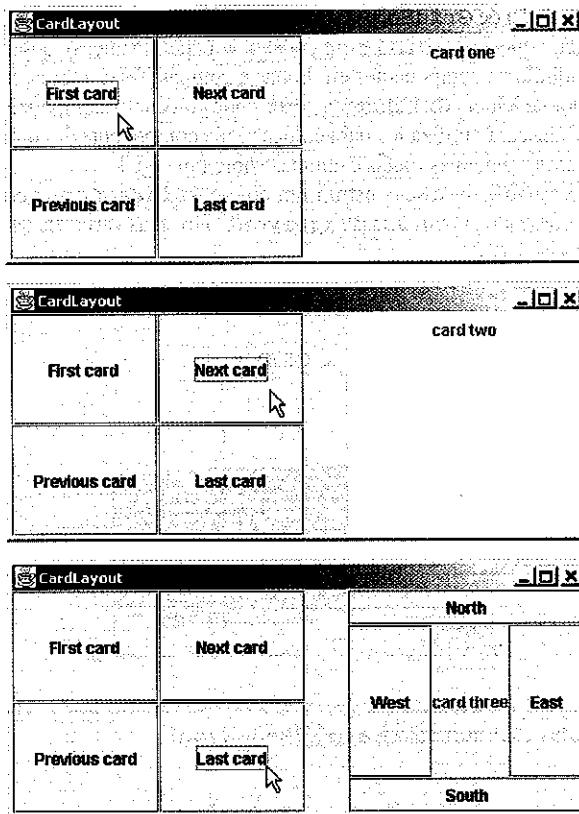


Fig. 13.16 Demonstrando o gerenciador de layout **CardLayout** (parte 3 de 3).

O método construtor (linhas 20 a 75) constrói a GUI. As linhas 27 a 29 criam o **JPanel deck**, criam o objeto **CardLayout cardManager** e configuram o gerenciador de layout de **deck** para **cardManager**. Em seguida, o construtor cria os **JPanels card1**, **card2** e **card3** e seus componentes GUI. À medida que configuramos cada carta, adicionamo-la ao **deck** com o método **add de Container** com dois argumentos – um **Component** e um **String**. O **Component** é o objeto **JPanel** que representa a carta. O argumento **String** identifica a carta. Por exemplo, a linha 36 adiciona o **JPanel card1** à pilha e utiliza o rótulo **JLabel label1** como o identificador **String** para a carta. Os **JPanels card2** e **card3** são adicionados ao **deck** nas linhas 44 a 55. Em seguida, o construtor cria o **JPanel buttons** e seus objetos **JButton** (linhas 58 a 66). A linha 64 registra o **ActionListener** para cada **JButton**. Por fim, os **buttons** e **deck** são adicionados às regiões **WEST** e **EAST** do painel de conteúdo, respectivamente.

O método **actionPerformed** (linhas 78 a 95) determina qual **JButton** gerou o evento, usando o método **getSource de EventObject**. Utilizam-se os métodos **first**, **previous**, **next** e **last** de **CardLayout** para exibir uma carta específica com base em qual **JButton** o usuário pressionou. O método **first** exibe a pri-

meira carta adicionada à pilha. O método `previous` exibe a carta anterior na pilha. O método `next` exibe a próxima carta. E o método `last` exibe a última carta da pilha. Observe que `deck` é passado para cada um desses métodos.

13.15 O gerenciador de leiaute `GridBagLayout`

O mais complexo e mais poderoso dos gerenciadores de leiaute predefinidos é `GridBagLayout`. Esse leiaute é semelhante a `GridLayout` porque `GridBagLayout` também dispõe os componentes em uma grade. Entretanto, `GridBagLayout` é mais flexível. Os componentes podem variar em tamanho (isto é, eles podem ocupar múltiplas linhas e colunas) e podem ser adicionados em qualquer ordem.

O primeiro passo na utilização de `GridBagLayout` é determinar a aparência da GUI. Esse passo não envolve nenhuma programação; tudo que é necessário é um pedaço de papel. Primeiro, desenhe a GUI. Depois desenhe uma grade sobre a GUI, dividindo os componentes em linhas e colunas. Os números iniciais de linha e coluna devem ser 0, assim o gerenciador de leiaute `GridBagLayout` pode colocar os componentes na grade adequadamente. Os números de linha e coluna serão utilizados para colocar cada componente em uma posição exata na grade. A Fig. 13.17 demonstra como se desenham as linhas e colunas sobre uma GUI.

Para utilizar `GridBagLayout`, deve-se construir um objeto `GridBagConstraints`. Esse objeto especifica como um componente é colocado em um `GridBagLayout`. Diversas variáveis de instância `GridBagConstraints` são resumidas na Fig. 13.18.

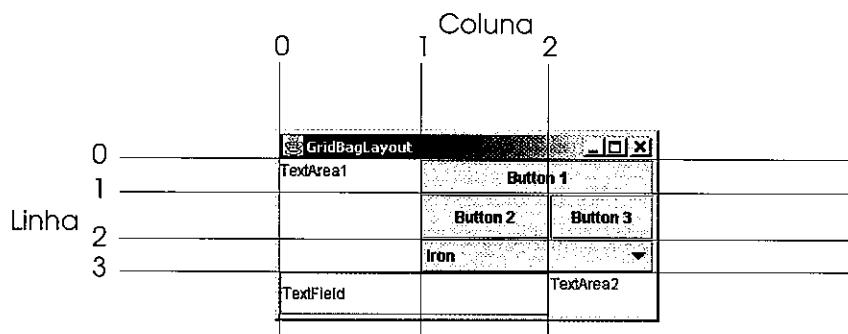


Fig. 13.17 Projetando uma GUI que utilizará `GridBagLayout`.

Variável de instância	Descrição
<code>gridx</code>	A coluna em que o componente será colocado.
<code>gridy</code>	A linha em que o componente será colocado.
<code>gridwidth</code>	O número de colunas que o componente ocupa.
<code>gridheight</code>	O número de linhas que o componente ocupa.
<code>weightx</code>	A parte de espaço extra a alocar horizontalmente. Os componentes podem se tornar mais largos quando há espaço extra disponível.
<code>weighty</code>	A parte de espaço extra a alocar verticalmente. Os componentes podem se tornar mais altos quando há espaço extra disponível.

Fig. 13.18 Variáveis de instância de `GridBagConstraints`.

As variáveis `gridx` e `gridy` especificam a linha e a coluna em que o canto superior esquerdo do componente é posicionado na grade. A variável `gridx` corresponde à coluna e a variável `gridy` corresponde à linha. Na Fig. 13.17, a `JComboBox` (exibindo “Iron”) tem um valor `gridx` de 1 e um valor `gridy` de 2.

A variável `gridwidth` especifica o número de colunas que um componente ocupa. Na Fig. 13.17, o botão `JComboBox` ocupa duas colunas. A variável `gridheight` especifica o número de linhas que um componente ocupa. Na Fig. 13.17, a `JTextArea` no lado esquerdo da janela ocupa três linhas.

A variável `weightx` especifica como distribuir o espaço horizontal extra para os componentes em um `GridBagLayout` quando o contêiner é redimensionado. Um valor zero indica que o componente não cresce horizontalmente por conta própria. Entretanto, se o componente abrange uma coluna que contém um componente com valor `weightx` diferente de zero, o componente com o valor `weightx` igual a zero crescerá horizontalmente na mesma proporção que o(s) outro(s) componente(s) na mesma coluna. Isso ocorre porque cada componente deve ser mantido na mesma linha e coluna em que foi originalmente posicionado.

A variável `weighty` especifica como distribuir o espaço vertical extra para os componentes em um `GridBagLayout` quando o contêiner é redimensionado. Um valor zero indica que o componente não cresce sozinho verticalmente. Entretanto, se o componente abrange uma linha que contém um componente com valor `weighty` diferente de zero, o componente com o valor `weighty` igual a zero cresce verticalmente na mesma proporção que o(s) outro(s) componente(s) na mesma linha.

Na Fig. 13.17, os efeitos de `weighty` e `weightx` não podem ser vistos facilmente até que o contêiner seja redimensionado e o espaço adicional se torne disponível. Os componentes com valores de peso maiores ocupam mais do espaço adicional que os componentes com valores de peso menores. Os exercícios exploram os efeitos de se variar `weightx` e `weighty`.

Os componentes devem receber valores de peso positivos diferentes de zero – caso contrário, os componentes “se amontoam” no meio do contêiner. A Fig. 13.19 mostra a GUI da Fig. 13.17 – em que todos os pesos foram configurados como zero.



Erro comum de programação 13.6

Utilizar um valor negativo para weightx ou weighty é um erro de lógica.

A variável de instância `fill` de `GridBagConstraints` especifica quanto da área do componente (o número de linhas e colunas que o componente ocupa na grade) é ocupada. Atribui-se à variável `fill` uma das seguintes constantes de `GridBagConstraints`: `NONE`, `VERTICAL`, `HORIZONTAL` ou `BOTH`. O valor `default` é `NONE`, o que indica que o componente não crescerá em nenhuma direção. `VERTICAL` indica que o componente crescerá verticalmente. `HORIZONTAL` indica que o componente crescerá horizontalmente. `BOTH` indica que o componente crescerá em ambas as direções.

A variável de instância `anchor` de `GridBagConstraints` especifica a localização do componente na área quando o componente não preenche a área inteira. Atribui-se à variável `anchor` uma das seguintes constantes de `GridBagConstraints`: `NORTH`, `NORTHEAST`, `EAST`, `SOUTHEAST`, `SOUTH`, `SOUTHWEST`, `WEST`, `NORTHWEST` ou `CENTER`. O valor `default` é `CENTER`.

O programa da Fig. 13.20 utiliza o gerenciador de layout `GridBagLayout` para organizar os componentes na GUI da Fig. 13.17. O programa não faz nada além de demonstrar como utilizar `GridBagLayout`.

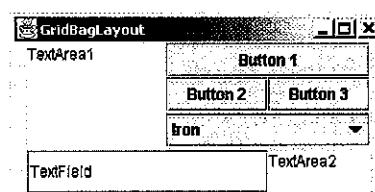


Fig. 13.19 GridBagLayout com `weightx` e `weighty` configurados como zero.

```

1 // Fig. 13.20: GridBagDemo.java
2 // Demonstrando GridBagLayout.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class GridBagDemo extends JFrame {
12     private Container container;
13     private GridBagLayout layout;
14     private GridBagConstraints constraints;
15
16     // configura a GUI
17     public GridBagDemo()
18     {
19         super( "GridBagLayout" );
20
21         container = getContentPane();
22         layout = new GridBagLayout();
23         container.setLayout( layout );
24
25         // instancia as restrições de Gridbag
26         constraints = new GridBagConstraints();
27
28         // cria componentes da GUI
29         JTextArea textArea1 = new JTextArea( "TextArea1", 5, 10 );
30         JTextArea textArea2 = new JTextArea( "TextArea2", 2, 2 );
31
32         String names[] = { "Iron", "Steel", "Brass" };
33         JComboBox comboBox = new JComboBox( names );
34
35         JTextField textField = new JTextField( "TextField" );
36         JButton button1 = new JButton( "Button 1" );
37         JButton button2 = new JButton( "Button 2" );
38         JButton button3 = new JButton( "Button 3" );
39
40         // textArea1
41         // weightx e weighty são ambas 0: o default
42         // âncora para todos os componentes é CENTER: o default
43         constraints.fill = GridBagConstraints.BOTH;
44         addComponent( textArea1, 0, 0, 1, 3 );
45
46         // button1
47         // weightx e weighty são ambas 0: o default
48         constraints.fill = GridBagConstraints.HORIZONTAL;
49         addComponent( button1, 0, 1, 2, 1 );
50
51         // comboBox
52         // weightx e weighty são ambas 0: o default
53         // preenchimento é HORIZONTAL
54         addComponent( comboBox, 2, 1, 2, 1 );
55
56         // button2
57         constraints.weightx = 1000;      // pode ficar mais larga
58         constraints.weighty = 1;        // pode ficar mais alta
59         constraints.fill = GridBagConstraints.BOTH;
60         addComponent( button2, 1, 1, 1, 1 );
61

```

Fig. 13.20 Demonstrando o gerenciador de layout GridBagLayout (parte 1 de 3).

```

62     // button3
63     // preenchimento é BOTH
64     constraints.weightx = 0;
65     constraints.weighty = 0;
66     addComponent( button3, 1, 2, 1, 1 );
67
68     // textField
69     // weightx e weighty são ambas 0, preenchimento é BOTH
70     addComponent( textField, 3, 0, 2, 1 );
71
72     // textArea2
73     // weightx e weighty são ambas 0, preenchimento é BOTH
74     addComponent( textArea2, 3, 2, 1, 1 );
75
76     setSize( 300, 150 );
77     setVisible( true );
78 }
79
80 // método para ativar as restrições
81 private void addComponent( Component component,
82     int row, int column, int width, int height )
83 {
84     // configura gridx e gridy
85     constraints.gridx = column;
86     constraints.gridy = row;
87
88     // configura gridwidth e gridheight
89     constraints.gridwidth = width;
90     constraints.gridheight = height;
91
92     // configura restrições e adiciona componente
93     layout.setConstraints( component, constraints );
94     container.add( component );
95 }
96
97 // executa o aplicativo
98 public static void main( String args[] )
99 {
100     GridBagDemo application = new GridBagDemo();
101
102     application.setDefaultCloseOperation(
103         JFrame.EXIT_ON_CLOSE );
104 }
105
106 } // fim da classe GridBagDemo

```

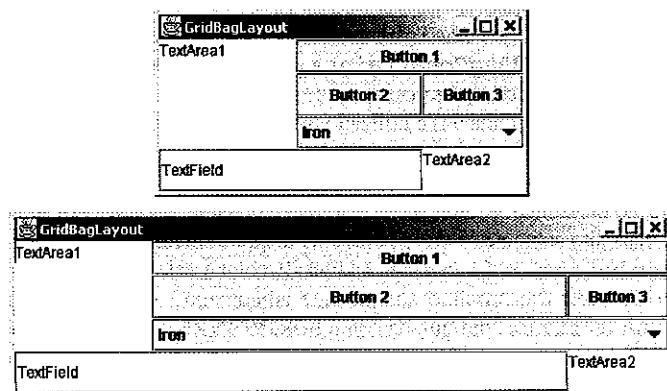


Fig. 13.20 Demonstrando o gerenciador de layout `GridBagLayout` (parte 2 de 3).

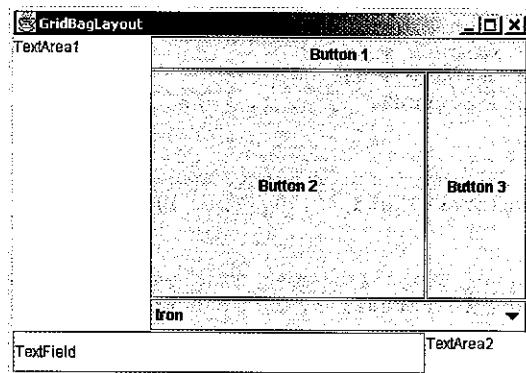


Fig. 13.20 Demonstrando o gerenciador de layout `GridLayout` (parte 3 de 3).

A GUI consiste em três `JButtons`, duas `JTextAreas`, um `JComboBox` e um `JTextField`. O gerenciador de layout para o painel de conteúdo é `GridLayout`. As linhas 22 e 23 instanciam o objeto `GridLayout` e configuram o gerenciador de layout para o painel de conteúdo como `layout`. O objeto `GridBagConstraints` utilizado para determinar a posição e o tamanho de cada componente na grade é instanciado com a linha 26. As linhas 29 a 38 instanciam cada um dos componentes GUI que será adicionado ao painel de conteúdo.

A `JTextArea` `textArea1` é o primeiro componente adicionado ao `GridLayout` (linha 44). Os valores para `weightx` e `weighty` não são especificados em `constraints`, então cada um tem o valor zero por *default*. Portanto, a `JTextArea` não redimensionará a si própria mesmo que haja espaço disponível. Entretanto, a `JTextArea` abrange múltiplas linhas, assim o tamanho vertical está sujeito aos valores `weighty` dos `JButtons` `button2` e `button3`. Quando `button2` ou `button3` é redimensionado verticalmente com base em seu valor `weighty`, a `JTextArea` também é redimensionada.

A linha 43 configura a variável `fill` em `constraints` como `GridBagConstraints.BOTH`, fazendo com que a `JTextArea` sempre ocupe toda a área alocada para ela na grade. Um valor `anchor` não é especificado em `constraints`, assim o *default* `CENTER` é utilizado. Não utilizamos a variável `anchor` nesse programa, então todos os componentes utilizarão `CENTER` por *default*. A linha 44 chama nosso método utilitário `addComponent` (definido nas linhas 81 a 95). O objeto `JTextArea`, a linha, a coluna, o número de colunas e o número de linhas a ocupar são passados como argumentos.

Os parâmetros do método `addComponent` são uma referência `component` a um `Component` e os inteiros `row`, `column`, `width` e `height`. As linhas 85 e 86 configuram as variáveis `GridBagConstraints gridx` e `gridy`. À variável `gridx` é atribuída a coluna em que o `Component` será posicionado e à variável `gridy` é atribuída a linha em que o `Component` será posicionado. As linhas 89 e 90 configuram as variáveis `GridBagConstraints gridwidth` e `gridheight`. A variável `gridwidth` especifica o número de colunas que o `Component` irá ocupar na grade, e a variável `gridheight` especifica o número de linhas que o `Component` irá ocupar na grade. A linha 93 configura as `GridBagConstraints` para um componente no `GridLayout`. O método `setConstraints` da classe `GridLayout` recebe um argumento `Component` e um argumento `GridBagConstraints`. O método `add` (linha 94) é utilizado para adicionar o componente ao painel de conteúdo.

O objeto `JButton` `button1` é o próximo componente adicionado (linhas 48 e 49). Os valores de `weightx` e `weighty` ainda são zero. A variável `fill` é configurada como `HORIZONTAL` – o componente sempre preencherá toda a sua área na direção horizontal. A direção vertical não é preenchida. Uma vez que o valor de `weighty` é zero, o botão se tornará mais alto somente se outro componente na mesma linha tiver um valor de `weighty` diferente de zero. O `JButton` `button1` está localizado na linha 0, coluna 1. Uma linha e duas colunas são ocupadas.

A `JComboBox` `comboBox` é o próximo componente adicionado (linha 54). Os valores de `weightx` e `weighty` são zero e a variável `fill` é configurada como `HORIZONTAL`. O botão `JComboBox` crescerá somente na direção horizontal. Observe que as variáveis `weightx`, `weighty` e `fill` permanecem configuradas em `constraints` até serem alteradas. O botão `JComboBox` é colocado na linha 2, coluna 1. Uma linha e duas colunas são ocupadas.

O objeto **JButton button2** é o próximo componente adicionado (linhas 57 a 60). Atribui-se a **JButton button2** um valor **weightx** de 1000 e um valor **weighty** de 1. A área ocupada pelo botão é capaz de crescer nas direções horizontal e vertical. A variável **fill** é configurada como **BOTH**, o que especifica que o botão sempre ocupará a área inteira. Quando a janela for redimensionada, **button2** crescerá. O botão é colocado na linha 1, coluna 1. Uma linha e uma coluna são ocupadas.

O **JButton button3** é adicionado a seguir (linhas 64 a 66). O valor de **weightx** e de **weighty** é configurado como zero e o valor de **fill** é **BOTH**. O **JButton button3** crescerá se a janela for redimensionada; ele é afetado pelos valores dos pesos de **button2**. Observe que o valor de **weightx** para **button2** é muito maior do que para **button3**. Quando ocorrer o redimensionamento, **button2** ocupará uma porcentagem maior do novo espaço. O botão é colocado na linha 1, coluna 2. Uma linha e uma coluna são ocupadas.

O **JTextField** (linha 70) e o **JTextArea textArea2** (linha 74) têm um valor de **weightx** de 0 e um valor de **weighty** de 0. O valor de **fill** é **BOTH**. O **JTextField** é colocado na linha 3, coluna 0 e a **JTextArea** é colocada na linha 3, coluna 2. O **JTextField** ocupa uma linha e duas colunas. A **JTextArea** ocupa uma linha e uma coluna.

Quando executar esse aplicativo, tente redimensionar a janela para ver como as limitações para cada componente GUI afetam sua posição e seu tamanho na janela.

13.16 As constantes Relative e Remainder de GridBagConstraints

Uma variação de **GridBagLayout** não utiliza **gridx** e **gridy**. Em seu lugar, são utilizadas as constantes **GridBagConstraints RELATIVE** e **REMAINDER**. **RELATIVE** especifica que o componente é o penúltimo componente nesta linha particular e que ele deve ser colocado à direita do componente anterior nessa linha. **REMAINDER** especifica que o componente é o último componente em uma linha. Qualquer componente que não seja o penúltimo ou o último componente em uma linha deve especificar os valores para as variáveis **gridwidth** e **gridheight** de **GridbagConstraints**. A classe **GridBagDemo2** na Fig. 13.21 dispõe os componentes em um **GridBagLayout** que utiliza essas constantes.

```

1 // Fig. 13.21: GridBagDemo2.java
2 // Demonstrando as constantes de GridBagConstraints.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class GridBagDemo2 extends JFrame {
12     private GridBagLayout layout;
13     private GridBagConstraints constraints;
14     private Container container;
15
16     // configura a GUI
17     public GridBagDemo2()
18     {
19         super( "GridBagLayout" );
20
21         container = getContentPane();
22         layout = new GridBagLayout();
23         container.setLayout( layout );
24
25         // instancia as restrições de Gridbag
26         constraints = new GridBagConstraints();
27

```

Fig. 13.21 Demonstrando as constantes **RELATIVE** e **REMAINDER** de **GridBagConstraints** (parte 1 de 3).

```

28     // cria componentes da GUI
29     String metals[] = { "Copper", "Aluminum", "Silver" };
30     JComboBox comboBox = new JComboBox( metals );
31
32     JTextField textField = new JTextField( "TextField" );
33
34     String fonts[] = { "Serif", "Monospaced" };
35     JList list = new JList( fonts );
36
37     String names[] =
38         { "zero", "one", "two", "three", "four" };
39     JButton buttons[] = new JButton[ names.length ];
40
41     for ( int count = 0; count < buttons.length; count++ )
42         buttons[ count ] = new JButton( names[ count ] );
43
44     // define as restrições para os componentes da GUI
45     // textField
46     constraints.weightx = 1;
47     constraints.weighty = 1;
48     constraints.fill = GridBagConstraints.BOTH;
49     constraints.gridx = GridBagConstraints.REMAINDER;
50     addComponent( textField );
51
52     // buttons[0] -- weightx e weighty são 1: preenchimento é BOTH
53     constraints.gridx = 1;
54     addComponent( buttons[ 0 ] );
55
56     // buttons[1] -- weightx e weighty são 1: preenchimento é BOTH
57     constraints.gridx = GridBagConstraints.RELATIVE;
58     addComponent( buttons[ 1 ] );
59
60     // buttons[2] -- weightx e weighty são 1: preenchimento é BOTH
61     constraints.gridx = GridBagConstraints.REMAINDER;
62     addComponent( buttons[ 2 ] );
63
64     // comboBox -- weightx é 1: preenchimento é BOTH
65     constraints.weighty = 0;
66     constraints.gridx = GridBagConstraints.REMAINDER;
67     addComponent( comboBox );
68
69     // buttons[3] -- weightx é 1: preenchimento é BOTH
70     constraints.weighty = 1;
71     constraints.gridx = GridBagConstraints.REMAINDER;
72     addComponent( buttons[ 3 ] );
73
74     // buttons[4] -- weightx e weighty são 1: preenchimento é BOTH
75     constraints.gridx = GridBagConstraints.RELATIVE;
76     addComponent( buttons[ 4 ] );
77
78     // list -- weightx e weighty são 1: preenchimento é BOTH
79     constraints.gridx = GridBagConstraints.REMAINDER;
80     addComponent( list );
81
82     setSize( 300, 200 );
83     setVisible( true );
84
85 } // fim do construtor
86

```

Fig. 13.21 Demonstrando as constantes RELATIVE e REMAINDER de GridBagConstraints (parte 2 de 3).

```

87     // addComponent é definido pelo programador
88     private void addComponent( Component component )
89     {
90         layout.setConstraints( component, constraints );
91         container.add( component );    // adiciona componente
92     }
93
94     // executa o aplicativo
95     public static void main( String args[] )
96     {
97         GridBagDemo2 application = new GridBagDemo2();
98
99         application.setDefaultCloseOperation(
100             JFrame.EXIT_ON_CLOSE );
101     }
102
103 } // fim da classe GridBagDemo2

```

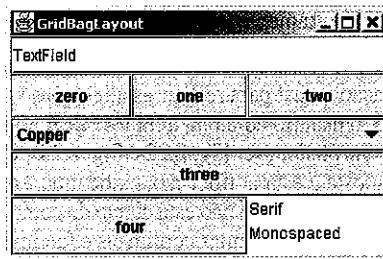


Fig. 13.21 Demonstrando as constantes `RELATIVE` e `REMAINDER` de `GridBagConstraints` (parte 3 de 3).

As linhas 22 e 23 constroem um `GridBagLayout` e configuram o gerenciador de layout do painel de conteúdo como `GridBagLayout`. Cada um dos componentes colocados em `GridBagLayout` é então construído (linhas 29 a 42). Os componentes são cinco `JButtons`, um `JTextField`, um `JList` e um `JComboBox`.

O `JTextField` é adicionado primeiro (linhas 46 a 50). Os valores `weightx` e `weighty` são configurados como 1. A variável `fill` é configurada como `BOTH`. A linha 49 especifica que o `JTextField` é o último componente na linha. O `JTextField` é adicionado ao painel de conteúdo com uma chamada para nosso método utilitário `addComponent` (definido nas linhas 88 a 92). O método `addComponent` recebe um argumento `Component` e utiliza o método `setConstraints` de `GridBagLayout` para configurar as limitações para o `Component`. O método `add` anexa o componente ao painel de conteúdo.

O `JButton buttons [0]` (linhas 53 e 54) tem os valores de `weightx` e `weighty` iguais a 1. A variável `fill` é `BOTH`. Uma vez que `buttons [0]` não é um dos dois últimos componentes na linha, é especificada `gridwidth` de 1 e, então, ele ocupa uma coluna. O `JButton` é adicionado ao painel de conteúdo com uma chamada para o método utilitário `addComponent`.

O `JButton buttons [1]` (linhas 57 e 58) tem os valores de `weightx` e `weighty` iguais a 1. A variável `fill` é `BOTH`. A linha 57 especifica que o `JButton` seja colocado à direita em relação ao componente anterior. O `Button` é adicionado à `JFrame` com uma chamada para `addComponent`.

O `JButton buttons [2]` (linhas 61 e 62) tem os valores de `weightx` e `weighty` iguais a 1. A variável `fill` é `BOTH`. Esse `JButton` é o último componente na linha, então `REMAINDER` é utilizado. O `JButton` é adicionado ao painel de conteúdo com uma chamada ao método utilitário `addComponent`.

O botão `JComboBox` (linhas 65 a 67) tem um `weightx` de 1 e um `weighty` de 0. O `JComboBox` não crescerá na direção vertical. O `JComboBox` é o único componente na linha, então `REMAINDER` é utilizado. O `JComboBox` é adicionado ao painel de conteúdo com uma chamada ao método utilitário `addComponent`.

O `JButton buttons [3]` (linhas 70 a 72) tem os valores de `weightx` e `weighty` iguais a 1. A variável `fill` é `BOTH`. Esse `JButton` é o único componente na linha, então `REMAINDER` é utilizado. O `JButton` é adicionado ao painel de conteúdo com uma chamada ao método utilitário `addComponent`.

O JButton buttons[4] (linhas 75 e 76) tem os valores de `weightx` e `weighty` iguais a 1. A variável `fill` é `BOTH`. Esse JButton é o penúltimo componente da linha, então `RELATIVE` é utilizado. O JButton é adicionado ao painel de conteúdo com uma chamada ao método utilitário `addComponent`.

O componente `JList` (linhas 79 e 80) tem os valores de `weightx` e `weighty` iguais a 1. A variável `fill` é `BOTH`. A `JList` é adicionada ao painel de conteúdo com uma chamada ao método utilitário `addComponent`.

13.17 (Estudo de caso opcional) Pensando em objetos: *Model-View-Controller*

Os *padrões de projeto* descrevem estratégias comprovadas para construção de sistemas de *software* orientado a objetos confiáveis. Nossa estudo de caso adota a arquitetura *Model-View-Controller* (MVC), a qual utiliza vários padrões de projeto¹. A MVC divide as responsabilidades do sistema em três partes:

1. o *modelo*, que contém todos os dados e a lógica do programa;
2. a *visão*, que fornece a apresentação visual para o modelo, e
3. o *controlador*, que define o comportamento do sistema enviando a entrada do usuário para o modelo.

Usando o controlador, o usuário altera os dados no modelo. O modelo, então, informa à visão sobre a alteração nos dados. A visão altera sua apresentação visual para refletir as alterações no modelo.

Por exemplo, em nossa simulação, o usuário adiciona uma `Person` ao modelo ao pressionar ou o JButton `First Floor` ou o `Second Floor` no controlador (ver Fig. 2.22 e Fig. 2.24). O modelo, então, notifica a visão sobre a criação da `Person`. A visão, em resposta a esta notificação, exibe uma `Person` em um `Floor`. O modelo não está ciente de como a visão exibe a `Person` e a visão não está ciente de como ou porque o modelo criou a `Person`.

A arquitetura MVC auxilia a construção de sistemas confiáveis e facilmente modificáveis. Se desejamos saída baseada em texto em vez de saída gráfica para a simulação do elevador, podemos criar uma visão alternativa para produzir saída baseada em texto sem alterar o modelo ou o controlador. Também podemos providenciar uma visão tridimensional que usa uma perspectiva de primeira pessoa para permitir que o usuário “participe” da simulação; tais visões são comumente empregadas em sistemas baseados em realidade virtual.

Simulação de elevador MVC

Agora vamos aplicar a arquitetura MVC para nossa simulação de elevador. Cada diagrama UML que providenciamos até este ponto (com exceção do diagrama de casos de uso) se relaciona com o modelo de nosso sistema de elevador. Providenciamos um diagrama de classes de “alto nível” da simulação na Fig. 13.22. A classe `ElevatorSimulation` – uma subclasse de `JFrame` – agrupa uma instância de cada uma das classes `ElevatorModel`, `ElevatorView` e `ElevatorController` para criar o aplicativo `ElevatorSimulation`. Como mencionado na Seção 10.22 de “Pensando em objetos”, o retângulo com o canto superior direito “dobrado” representa uma anotação em UML. Neste caso, cada anotação aponta para uma classe específica (através de uma linha tracejada) para descrever o papel daquela classe no sistema. As classes `ElevatorModel`, `ElevatorView` e `ElevatorController` encapsulam todos os objetos que compreendem o modelo, a visão e o controlador de nossa simulação, respectivamente.

Mostramos na Fig. 9.38 que a classe `ElevatorModel` é uma agregação de várias classes. Para economizar espaço, não vamos representar esta agregação na Fig. 13.22. A classe `ElevatorView` é também uma agregação de várias classes – expandimos o diagrama de classes de `ElevatorView` na Seção 22.9 de “Pensando em objetos” para mostrar essas classes adicionais. A classe `ElevatorController`, como descrita na Seção 12.16, representa o controlador da simulação. Observe que a classe `ElevatorView` implementa a interface `ElevatorModelListener`, que implementa todas as interfaces usadas na nossa simulação, de modo que `ElevatorView` pode receber todos os eventos do modelo.

¹ Para aqueles leitores que desejam se aprofundar no estudo de padrões de projeto e na arquitetura MVC, recomendamos a leitura de nosso material “Descobrindo padrões de projetos” nas Seções 1.16, 9.24, 13.18, 15.13, 17.11 e 21.12.

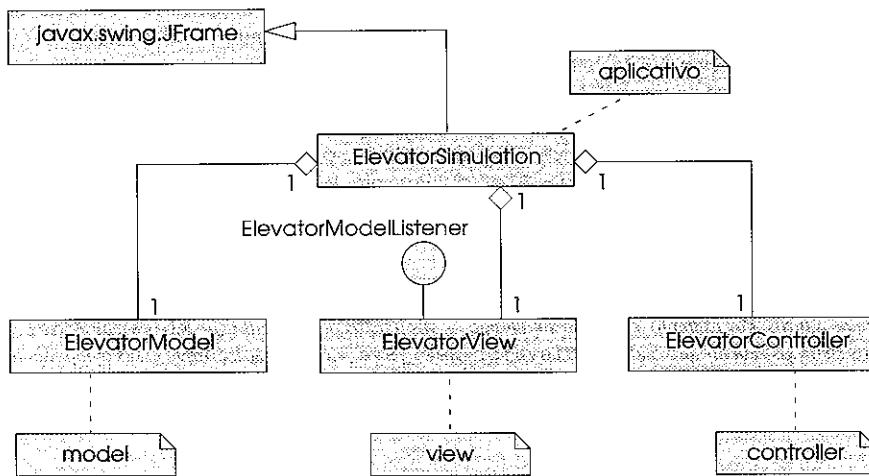


Fig. 13.22 Diagrama de classes da simulação do elevador.



Observação de engenharia de software 13.7

Quando apropriado, faça a partição do diagrama de classes dos sistemas em diversos diagramas de classes menores, de modo que cada diagrama represente um subsistema exclusivo.

A classe `ElevatorSimulation` não contém outros atributos além de suas referências para um objeto `ElevatorModel`, um objeto `ElevatorView` e um objeto `ElevatorController`. O único comportamento para a classe `ElevatorSimulation` é iniciar o programa – portanto, em Java, a classe `ElevatorSimulation` contém um método `static main` que chama o construtor, o qual instancia os objetos `ElevatorModel`, `ElevatorView` e `ElevatorController`. Implementamos a classe `ElevatorSimulation` em Java mais adiante nessa seção.

Diagramas de componentes

A Fig. 13.22 nos auxilia a construir outro diagrama de UML que usamos para projetar nosso sistema – o *diagrama de componentes*. O diagrama de componentes modela as “peças” – denominadas *componentes* – de que o sistema necessita para realizar suas tarefas. Essas peças incluem os executáveis binários, os arquivos `.class` compilados, os arquivos `.java`, imagens, pacotes, recursos, etc. Apresentamos o diagrama de componentes de nossa simulação na Fig. 13.23.

Na Fig. 13.23, cada caixa que contém as duas pequenas caixas brancas sobrepostas em seu lado esquerdo é um *componente*. Desenha-se o componente em UML de forma similar a um *plug* (as duas caixas sobrepostas representam os pinos do *plug*) – pode-se conectar um componente a outros sistemas sem ter que alterar o componente. Nossos sistemas contêm cinco componentes: `ElevatorSimulation.class`, `ElevatorSimulation.java`, `ElevatorModel.java`, `ElevatorView.java` e `ElevatorController.java`.

Na Fig. 13.23, o desenho parecido com pastas (caixas com orelhas no canto superior esquerdo) representam *pacotes* em UML. Podemos agrupar classes, objetos, componentes, casos de uso, etc., em um pacote. Neste diagrama (e no restante de nosso estudo de caso), os pacotes UML se referem a pacotes Java (apresentados na Seção 8.5). Em nossa discussão, usamos o tipo Courier minúsculo em negrito para os nomes de pacotes. Os pacotes no nosso sistema são `model`, `view` e `controller`. O componente `ElevatorSimulation.java` contém uma instância de cada um dos componentes destes pacotes. Atualmente, cada pacote contém somente um componente – um arquivo `.java`. O pacote `model` contém `ElevatorModel.java`, o pacote `view` contém `ElevatorView.java`

e o pacote **controller** contém **ElevatorController.java**. Adicionamos componentes a cada pacote nos apêndices, quando implementamos cada classe, a partir de nossos diagramas de classes, em um componente (um arquivo **.java**).

As setas tracejadas na Fig. 13.23 indicam uma *dependência* entre dois componentes – a direção da seta indica o relacionamento “*depende de*”. A dependência descreve o relacionamento entre os componentes no qual as alterações feitas em um componente afetam outro componente. Por exemplo, o componente **ElevatorSimulation.class** depende do componente **ElevatorSimulation.java** porque uma alteração em **ElevatorSimulation.java** afeta **ElevatorSimulation.class** quando **ElevatorSimulation.java** é compilado. A Seção 12.16 mencionou que o objeto **ElevatorController** contém uma referência para o objeto **ElevatorModel** (para colocar **Persons** nos **Floors**). Portanto, **ElevatorController.java** depende de **ElevatorModel.java**.



Observação de engenharia de software 13.8

O diagrama de dependência de componentes auxilia os projetistas a agrupar componentes para reutilização em sistemas futuros. Por exemplo, em nossa simulação, os projetistas podem reutilizar **ElevatorModel.java** em outros sistemas sem ter que reutilizar **ElevatorView.java** (e vice-versa), porque esses componentes não dependem um do outro. Entretanto, se um projetista quiser reutilizar **ElevatorController.java**, o projetista terá que reutilizar **ElevatorModel.java**.

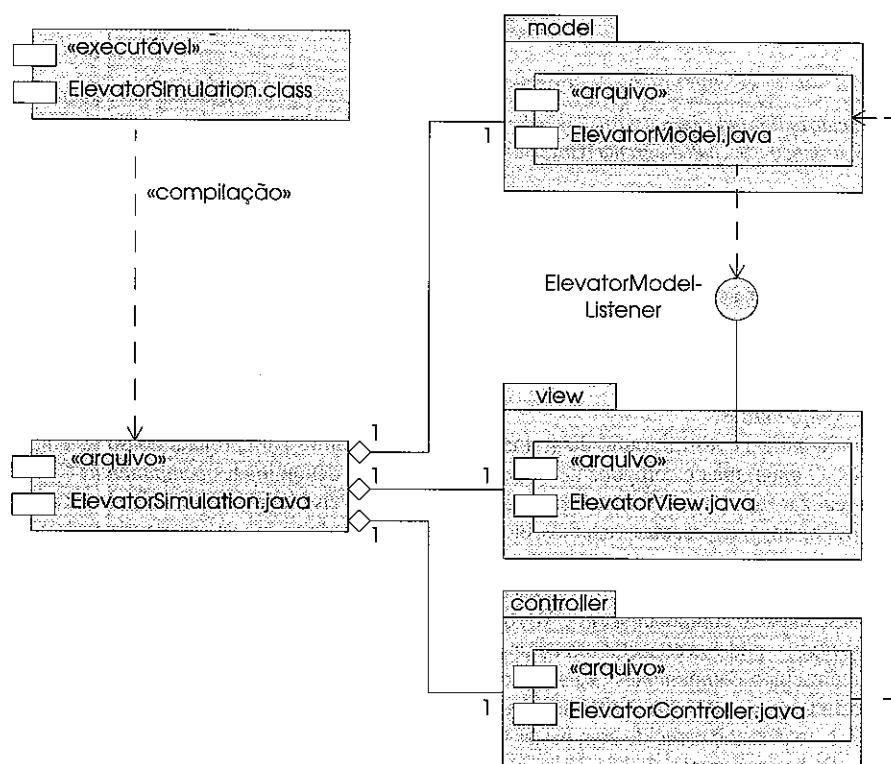


Fig. 13.23 Diagrama de componentes para a simulação do elevador.

De acordo com a Fig. 13.23, `ElevatorModel.java` e `ElevatorView.java` não dependem um do outro – eles se comunicam através da interface `ElevatorModelListener`, que implementa todas as interfaces da simulação. `ElevatorView.java` concretiza a interface `ElevatorModelListener` e `ElevatorModel.java` depende da interface `ElevatorModelListener`.

A Fig. 13.23 contém vários *estereótipos* – palavras colocadas entre aspas francesas (« e ») que indicam o papel de um elemento. Mencionamos o estereótipo «interface» na Seção 11.10 de “Pensando em objetos”. O estereótipo «compilação» descreve a dependência entre `ElevatorSimulation.class` e `ElevatorSimulation.java` – `ElevatorSimulation.java` é compilado para `ElevatorSimulation.class`. O estereótipo «executável» especifica que o componente é um aplicativo, e o estereótipo «arquivo» especifica que o componente é um arquivo que contém o código-fonte para o executável.

Implementação: ElevatorSimulation.java

Usamos o diagrama de componentes da Fig. 13.23, o diagrama de classes da Fig. 13.22 e o diagrama de casos de uso da Fig. 12.28 para implementar `ElevatorSimulation.java` (Fig. 13.24). As linhas 12 a 14 importam os pacotes `model`, `view` e `controller`, como especificados na Fig. 13.23.

```

1 // ElevatorSimulation.java
2 // Aplicativo com Model, View e Controller (MVC) de Elevator
3 package com.deitel.jhtp4.elevator;
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 // pacotes Deitel
12 import com.deitel.jhtp4.elevator.model.*;
13 import com.deitel.jhtp4.elevator.view.*;
14 import com.deitel.jhtp4.elevator.controller.*;
15
16 public class ElevatorSimulation extends JFrame {
17
18     // Model, View e Controller
19     private ElevatorModel model;
20     private ElevatorView view;
21     private ElevatorController controller;
22
23     // construtor instancia Model, View e Controller
24     public ElevatorSimulation()
25     {
26         super( "Deitel Elevator Simulation" );
27
28         // instancia Model, View e Controller
29         model = new ElevatorModel();
30         view = new ElevatorView();
31         controller = new ElevatorController( model );
32
33         // registra View para eventos de Model
34         model.setElevatorModelListener( view );
35
36         // adiciona visão e controlador a ElevatorSimulation
37         getContentPane().add( view, BorderLayout.CENTER );
38         getContentPane().add( controller, BorderLayout.SOUTH );
39
40     } // fim do construtor ElevatorSimulation
41

```

Fig. 13.24 A classe `ElevatorSimulation` é o aplicativo para a simulação do elevador (parte 1 de 2).

```

42  // método main inicia a execução do programa
43  public static void main( String args[] )
44  {
45      // instancia ElevatorSimulation
46      ElevatorSimulation simulation = new ElevatorSimulation();
47      simulation.setDefaultCloseOperation( EXIT_ON_CLOSE );
48      simulation.pack();
49      simulation.setVisible( true );
50  }
51 }
```

Fig. 13.24 A classe `ElevatorSimulation` é o aplicativo para a simulação do elevador (parte 2 de 2).

O diagrama de classes da Fig. 13.22 especifica que a classe `ElevatorSimulation` é uma subclasse de `javax.swing.JFrame` – a linha 16 declara a classe `ElevatorSimulation` como uma classe `public` que estende a classe `JFrame`. As linhas 19 a 21 implementam a agregação, na classe `ElevatorSimulation`, das classes `ElevatorModel`, `ElevatorView` e `ElevatorController` (mostradas na Fig. 13.22) pela declaração de um objeto de cada classe. As linhas 29 a 31 do construtor de `ElevatorSimulation` inicializam estes objetos.

As Figs. 13.22 e 13.23 especificam que `ElevatorView` é um `ElevatorModelListener` para o `ElevatorModel`. A linha 34 registra o `ElevatorView` como um ouvinte para `ElevatorModelEvents`, de modo que `ElevatorView` pode receber eventos do `ElevatorModel` e representar adequadamente o estado do modelo. As linhas 37 e 38 adicionam o `ElevatorView` e o `ElevatorController` ao `ElevatorSimulation`. Conforme os estereótipos da Fig. 13.23, `ElevatorSimulation.java` é compilado para `ElevatorSimulation.class`, o qual é executável – as linhas 43 a 50 fornecem o método `main` que executa o aplicativo.

Completamos o projeto dos componentes de nosso sistema. A Seção 15.12 de “Pensando em objetos” conclui o projeto do modelo, resolvendo os problemas de interação encontrados na Fig. 10.25. Por fim, a Seção 22.9 completa o projeto da visão e descreve em detalhes como o `ElevatorView` recebe eventos de `ElevatorModel`. Estas duas últimas seções preparam-no para percorrer a nossa implementação da simulação do elevador nos Apêndices G, H e I.

13.18 (Opcional) Descobrindo padrões de projeto: padrões de projeto usados nos pacotes `java.awt` e `javax.swing`

Continuamos nossa discussão da Seção 9.24 sobre padrões de projeto. Essa seção apresenta aqueles padrões de projeto associados aos componentes GUI de Java. Após a leitura dessa seção, você deverá entender melhor como esses componentes se beneficiam com padrões de projeto e como os desenvolvedores integram os padrões de projeto com os aplicativos Java GUI.

13.18.1 Padrões de criação de projeto

Agora, continuamos nosso tratamento de padrões de criação de projeto, os quais fornecem maneiras de instanciar objetos em um sistema.

Factory Method

Suponha que estamos projetando um sistema que abre uma imagem de um arquivo especificado. Existem diversos formatos de imagem, como GIF e JPEG. Podemos usar o método `createImage` da classe `java.awt.Component` para criar um objeto `Image`. Por exemplo, para criar uma imagem JPEG e GIF em um objeto de uma subclasse de `Component` – como um objeto `JPanel` – passamos o nome do arquivo de imagem para o método `createImage`, que devolve um objeto `Image` que armazena os dados da imagem. Podemos criar dois objetos `Image`, cada qual contendo dados para duas imagens que possuem estruturas completamente diferentes. Por exemplo, uma imagem JPEG pode usar até 16,7 milhões de cores enquanto que uma imagem `Image` pode usar, no máximo, 256. Além disso, uma imagem GIF pode conter *pixels* transparentes que não são exibidos na tela, enquanto uma imagem JPEG não pode conter *pixels* transparentes.

A classe `Image` é uma classe abstrata que representa uma imagem que podemos exibir na tela. Usando o parâmetro passado pelo programador, o método `createImage` determina a subclasse de `Image` específica através da qual será instanciado o objeto `Image`. Podemos projetar sistemas para permitir que o usuário especifique a imagem a ser criada, e o método `createImage` irá determinar a subclasse a partir da qual instanciar a `Image`. Se o parâmetro passado ao método `createImage` fizer referência a um arquivo JPEG, o método `createImage` instancia e devolve um objeto de uma subclasse de `Image` adequada a imagens JPEG. Se o parâmetro passado ao método `createImage` fizer referência a um arquivo GIF, o método `createImage` instancia e devolve um objeto de uma subclasse de `Image` adequada a imagens GIF.

O método `createImage` é um exemplo do *padrão de projeto Factory Method*. O único objetivo deste *método de fabricação* é criar objetos que permitem ao sistema determinar qual classe instanciar durante a execução. Podemos projetar um sistema que permita a um usuário especificar qual o tipo da imagem a ser criada durante a execução. A classe `Component` poderia não ser capaz de determinar qual subclasse de `Image` a ser instanciada até que o usuário especifique a imagem a ser carregada. Para obter mais informações sobre o método `createImage`, visite

[www.java.sun.com/j2se/1.3/docs/api/java.awt.Component.html#createImage\(java.awt.image.ImageProducer\)](http://www.java.sun.com/j2se/1.3/docs/api/java.awt.Component.html#createImage(java.awt.image.ImageProducer))

13.18.2 Padrões estruturais de projeto

Discutimos agora mais três padrões estruturais de projeto. O padrão de projeto *Adapter* ajuda os objetos que têm interfaces incompatíveis a colaborar entre si. O padrão de projeto *Bridge* auxilia os projetistas a melhorar a independência de plataforma em seus sistemas. O *Composite* fornece uma maneira para os projetistas organizarem e manipularem os objetos.

Adapter

O *padrão de projeto Adapter* fornece um objeto com uma nova interface que se *adapta* à interface de outro objeto, permitindo que ambos os objetos colaborem entre si. O adaptador, neste padrão, é semelhante a um adaptador para um *plug* de um dispositivo elétrico – as tomadas elétricas na Europa são diferentes das americanas, de modo que é necessário um adaptador para conectar um dispositivo americano a uma tomada elétrica europeia e vice-versa.

Java fornece várias classes que usam o padrão de projeto *Adapter*. Os objetos destas classes atuam como adaptadores entre objetos que geram certos eventos e aqueles objetos que tratam estes eventos. Por exemplo, um `MouseAdapter`, que explicamos na Seção 12.12, adapta um objeto que gera `MouseEvents` a um objeto que trata `MouseEvents`.

Bridge

Suponha que estejamos projetando a classe `Button` para os sistemas operacionais Windows e Macintosh. A classe `Button` contém informações específicas sobre botões, como um `ActionListener` e um rótulo `String`. Projetamos as classes `Win32Button` e `MacButton` que estendem a classe `Button`. A classe `Win32Button` contém informações de aparência e comportamento sobre como exibir um `Button` no sistema operacional Windows, e a classe `MacButton` contém informações de aparência e comportamento sobre como exibir um `Button` no sistema operacional Macintosh.

Dois problemas emergem deste enfoque. Primeiro, se criamos novas subclasses de `Button` devemos criar as subclasses correspondentes `Win32Button` e `MacButton`. Por exemplo, se criamos a classe `ImageButton` (um `Button` com uma imagem sobreposta), que estende a classe `Button`, devemos criar as subclasses adicionais `Win32ImageButton` e `MacImageButton`. De fato, devemos criar subclasses `Button` para cada sistema operacional que queiramos suportar, o que aumenta o tempo de desenvolvimento. O segundo problema é que, quando um novo sistema operacional entra no mercado, devemos criar classes `Button` adicionais específicas para aquele sistema operacional.

O padrão de projeto *Bridge* evita esses problemas pela separação de uma abstração, por exemplo, um `Button`, e suas implementações (por exemplo, `Win32Button`, `MacButton`, etc.) em diferentes hierarquias de classes. Por exemplo, as classes Java AWT usam o padrão de projeto *Bridge* para permitir que os projetistas criem subclasses de `Button` AWT sem necessitar criar as subclasses correspondentes para cada sistema operacional. Cada `Button` AWT mantém uma referência para um `ButtonPeer`, que é a superclasse para as implementações específicas para cada plataforma, como `Win32ButtonPeer`, `MacButtonPeer`, etc. Quando o programador cria um objeto

Button, a classe **Button** chama o método de fabricação **createButton**, da classe **Toolkit**, para criar o objeto **ButtonPeer** específico para a plataforma. O objeto **Button** armazena uma referência para seu **ButtonPeer** – esta referência é a “ponte” no padrão de projeto *Bridge*. Quando o programador invoca métodos sobre o objeto **Button**, este invoca os métodos apropriados de seu **ButtonPeer** para atender à requisição. Se um projetista cria uma subclasse de **Button** chamada **ImageButton**, ele não precisa criar um **Win32ImageButton** ou **MacImageButton** correspondente, com recursos de desenho de imagem específicos da plataforma. O **ImageButton** “é um” **Button**. Portanto, quando o **ImageButton** necessita exibir sua imagem, ele usa o objeto **Graphics** de seu **ButtonPeer** para exibir a imagem em cada plataforma. Este padrão de projeto permite que os projetistas criem novos componentes GUI para diferentes plataformas usando uma “ponte” para esconder detalhes específicos da plataforma.



Dica de portabilidade 13.2

Os projetistas em geral usam o padrão de projeto *Bridge* para aumentar a independência de plataforma de seus sistemas. Este padrão de projeto permite que se criem novos componentes para diferentes plataformas com uma “ponte”, usada para esconder detalhes específicos da plataforma.

Composite

Os projetistas em geral organizam os componentes em estruturas hierárquicas (por exemplo, uma hierarquia de diretórios e arquivos em um disco rígido) – nas quais cada nodo na estrutura representa um componente (por exemplo, um arquivo ou um diretório). Cada nodo pode conter referências a outros nodos. O nodo se chama *ramo* se ele contiver uma referência para um ou mais nodos (por exemplo, um diretório que contém arquivos). O nodo se chama *folha* se ele não contiver uma referência a outro nodo (por exemplo, um arquivo). Algumas vezes, uma estrutura contém objetos de várias classes diferentes (por exemplo, um diretório pode conter arquivos e diretórios). Quando um objeto – denominado *cliente* – deseja percorrer a estrutura, o cliente deve determinar a classe particular de cada nodo. Fazer esta determinação pode ser demorado e a estrutura pode tornar-se difícil de manter.

No padrão de projeto *Composite*, cada componente numa estrutura hierárquica implementa a mesma interface ou estende uma superclasse comum. Este polimorfismo (apresentado na Seção 9.10) assegura que os clientes podem percorrer todos os elementos – ramos ou folhas – da estrutura de maneira uniforme. Usando este padrão, o cliente que está percorrendo a estrutura não precisa determinar o tipo de cada componente, porque todos os componentes implementam a mesma interface ou estendem a mesma superclasse.

Os componentes GUI de Java usam o padrão de projeto *Composite*. Considere a classe de componentes Swing **JPanel**, que estende a classe **JComponent**. A classe **JComponent** estende a classe **java.awt.Container**, a qual estende a classe **java.awt.Component** (Fig. 13.25). A classe **Container** fornece o método **add**, que anexa um objeto **Component** (ou um objeto de uma subclasse de **Component**) àquele objeto **Container**. Portanto, pode-se adicionar um objeto **JPanel** a qualquer objeto de uma subclasse de **Component** e qualquer obje-

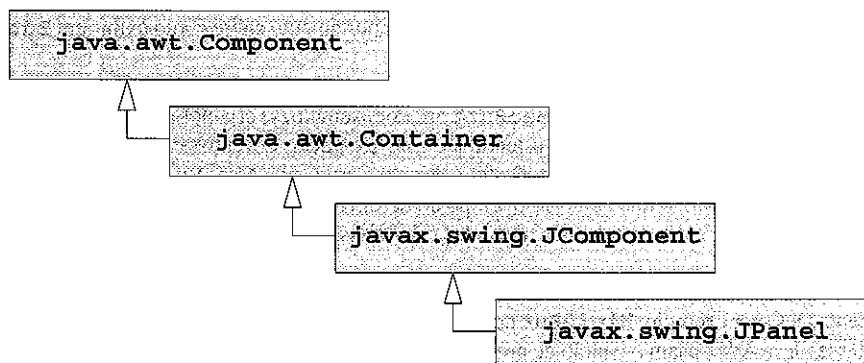


Fig. 13.25 Hierarquia de herança para a classe **JPanel**.

to de uma subclasse de **Component** pode ser adicionado àquele objeto **JPanel**. O objeto **JPanel** pode conter qualquer componente GUI, mesmo sem saber o tipo específico daquele componente.

O cliente, como o objeto **JPanel**, pode percorrer todos os componentes uniformemente na hierarquia. Por exemplo, se o objeto **JPanel** chama o método **repaint** da superclasse **Container**, o método **repaint** exibe o objeto **JPanel** e todos os componentes adicionados ao objeto **JPanel**. O método **repaint** não tem que determinar o tipo de cada componente, porque todos os componentes herdam da superclasse **Container**, a qual contém o método **repaint**.

13.18.3 Padrões comportamentais de projeto

Nesta seção, continuamos nossa discussão sobre padrões de projeto de comportamento. Discutimos os padrões de projeto *Chain-of-Responsibility*, *Command*, *Observer*, *Strategy* e *Template Method*.

Chain-of-Responsibility

Nos sistemas orientados a objetos, os objetos interagem enviando mensagens uns aos outros. Freqüentemente, um sistema precisa determinar durante a execução qual o objeto que irá tratar uma mensagem em particular. Por exemplo, pense no projeto de um sistema telefônico de três linhas para um escritório. Quando uma pessoa chama o escritório, a primeira linha atende à chamada – se a primeira linha está ocupada, a segunda linha atende à chamada e, se a segunda linha está ocupada, a terceira linha atende à chamada. Se todas as linhas do sistema estão ocupadas, uma gravação automática diz àquela pessoa para esperar pela próxima linha disponível – quando uma linha se torna disponível, ela atende a chamada.

O padrão de projeto *Chain-of-Responsibility* permite determinar, durante a execução, o objeto que irá tratar a mensagem. Este padrão permite enviar uma mensagem para diversos objetos em uma *cadeia* de objetos. Cada objeto na cadeia pode tratar da mensagem ou passar a mensagem para o próximo objeto na cadeia. Por exemplo, a primeira linha no sistema telefônico é o primeiro objeto na cadeia de responsabilidade, a segunda linha é o segundo objeto, a terceira linha é o terceiro objeto e a gravação automática é o quarto objeto. Note que esse mecanismo não é o último objeto da cadeia – a próxima linha disponível trata da mensagem e aquela linha é o objeto final na cadeia. A cadeia é criada dinamicamente em resposta à presença ou ausência de tratadores de mensagens específicos.

Diversos componentes de Java AWT usam o padrão de projeto *Chain-of-Responsibility* para tratar de certos eventos. Por exemplo, a classe **java.awt.Button** sobrescreve o método **processEvent** da classe **java.awt.Component** para processar objetos **AWTEvent**. O método **processEvent** tenta tratar o **AWTEvent** quando recebe este evento como argumento. Se o método **processEvent** determina que o **AWTEvent** é um **ActionEvent** (isto é, o **Button** foi pressionado), o método trata do evento chamando o método **processActionEvent**, que informa a qualquer **ActionListener** registrado com o **Button** que o **Button** foi pressionado. Se o método **processEvent** determina que o **AWTEvent** não é um **ActionEvent**, o método é incapaz de tratar o evento e passa o **AWTEvent** para o método **processEvent** da superclasse **Component** (o próximo objeto na cadeia).

Command

Os aplicativos geralmente oferecem ao usuário diversas maneiras de executar uma dada tarefa. Por exemplo, em um processador de texto poderia haver um menu **Edit** com itens de menu para recortar, copiar e colar texto. Também poderia existir uma barra de ferramentas e/ou um menu *pop-up* oferecendo os mesmos itens. A funcionalidade que o aplicativo oferece é a mesma em cada caso – os diferentes componentes da interface para invocar a funcionalidade são oferecidos para conveniência do usuário. Entretanto, a mesma instância de componente GUI (por exemplo, **JButton**) não pode ser usada para menus e barras de ferramentas e menus *pop-up*, de modo que o desenvolvedor precisa codificar a mesma funcionalidade três vezes. Se existissem muitos destes itens de interface, repetir esta funcionalidade se tornaria tedioso e sujeito a erros.

O padrão de projeto *Command* resolve este problema permitindo especificar a funcionalidade desejada (por exemplo, copiar texto) uma só vez, em um objeto reutilizável; aquela funcionalidade pode então ser adicionada a um menu, uma barra de ferramentas, um menu *pop-up* ou outros mecanismos. Este padrão de projeto se chama *Command* porque ele define um comando, ou instrução, do usuário. Este padrão permite encapsular um comando, de modo que o comando possa ser usado em diversos objetos.

Observer

Suponha que queiramos projetar um programa para visualizar informações de uma conta bancária. Este sistema inclui a classe `BankStatementData` para armazenar os dados relativos a extratos bancários e as classes `TextDisplay`, `BarGraphDisplay` e `PieChartDisplay` para exibir os dados². A Fig. 13.26 mostra o projeto de nosso sistema. A classe `TextDisplay` exibe os dados em forma de texto, a classe `BarGraphDisplay` exibe os dados em forma de gráfico de barras e a classe `PieChartDisplay` exibe os dados em forma de gráfico de torta. Queremos projetar o sistema de maneira que o objeto `BankStatementData` notifique os objetos que exibem os dados de uma mudança nos dados. Também queremos projetar o sistema para afrouxar o *acoplamento* – o grau em que as classes dependem umas das outras em um sistema.



Observação de engenharia de software 13.9

As classes frouxamente acopladas são mais fáceis de reutilizar e modificar do que são as classes fortemente acopladas, as quais dependem pesadamente umas das outras. A modificação em uma classe em um sistema fortemente acoplado normalmente resulta na modificação de outras classes naquele sistema. A modificação em uma classe de um grupo de classes frouxamente acopladas poderia exigir pouca ou nenhuma modificação em outras classes do grupo.

O padrão de projeto *Observer* é apropriado para os sistemas como aquele da Fig. 13.26. Este padrão favorece o acoplamento frágil entre o objeto sujeito e objetos observadores – um sujeito notifica os observadores quando o sujeito muda de estado. Quando notificados pelo sujeito, os observadores mudam em resposta à mudança no sujeito. Em nosso exemplo, o objeto `BankStatementData` é o sujeito e os objetos que exibem os dados são os observadores. Um sujeito pode notificar diversos observadores; portanto, o sujeito contém um relacionamento um para muitos com os observadores.

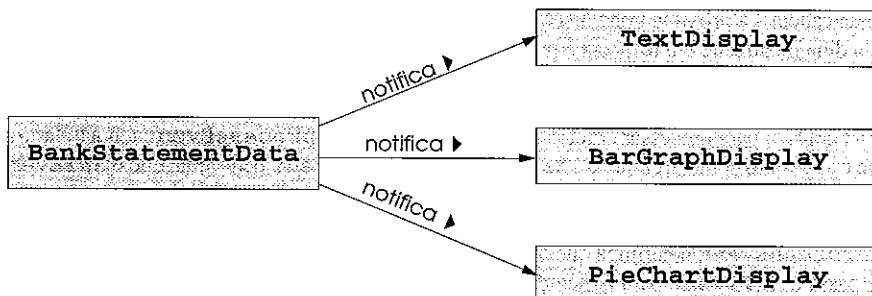


Fig. 13.26 Base para o padrão de projeto *Observer*.

A Java API contém classes que usam o padrão de projeto *Observer*. A classe `java.util.Observable` representa um sujeito. A classe `Observable` fornece o método `addObserver`, que recebe um argumento `java.util.Observer`. A interface `Observer` permite que o objeto `Observable` notifique o `Observer` quando os objetos `Observable` mudam de estado. O `Observer` pode ser uma instância de qualquer classe que implementa a interface `Observer`; como o objeto `Observable` invoca métodos definidos na interface `Observer`, os objetos ficam frouxamente acoplados. Se o desenvolvedor muda a maneira pela qual um `Observer` em particular responde a mudanças no objeto `Observable`, o desenvolvedor não precisa mudar o objeto `Observable`. O objeto `Observable` interage com seus `Observers` somente através da interface `Observer`, que permite o acoplamento frágil.

² Esta abordagem é a base para o padrão de arquitetura *Model-View-Controller*, discutido nas Seções 13.17 e 17.11.

O estudo opcional sobre simulação de elevador, nas seções “Pensando em objetos”, usa o padrão de projeto *Observer* para permitir que o objeto **ElevatorModel** (o sujeito) notifique o objeto **ElevatorView** (o observador) das mudanças no **ElevatorModel**. A simulação não usa a classe **Observable** e a interface **Observer** da biblioteca de Java – em vez disso, ela usa uma interface personalizada **EMListener** que fornece funcionalidade semelhante àquela da interface **Observable**.

Os componentes GUI Swing usam o padrão de projeto *Observer*. Os componentes GUI colaboram com seus *ouvintes* para responder a interações do usuário. Por exemplo, um **ActionListener** observa as mudanças de estado em um **JButton** (o sujeito) se registrando para tratar os eventos daquele **JButton**. Ao ser pressionado pelo usuário, o **JButton** notifica seus objetos **ActionListeners** (os observadores) de que o estado do **JButton** mudou (isto é, o **JButton** foi pressionado).

Strategy

O *padrão de projeto Strategy* é semelhante ao padrão de projeto *State* (discutido na Seção 9.24.3). Mencionamos que o padrão de projeto *State* contém um objeto estado, que encapsula o estado de um objeto de contexto. O padrão de projeto *Strategy* contém um *objeto estratégia*, que é análogo ao objeto estado do padrão de projeto *State*. A diferença fundamental entre um objeto estado e um objeto estratégia é que o objeto estratégia encapsula um *algoritmo* em vez das informações sobre o estado.

Por exemplo, os componentes `java.awt.Container` implementam o padrão de projeto *Strategy* usando *LayoutManagers* (discutidos na Seção 12.14) como objetos estratégia. No pacote `java.awt`, as classes `FlowLayout`, `BorderLayout` e `GridLayout` implementam a interface `LayoutManager`. Cada classe usa o método `addLayoutComponent` para adicionar componentes GUI a um objeto `Container` – entretanto, cada método usa um algoritmo diferente para exibir aqueles componentes GUI: um `FlowLayout` exibe componentes numa seqüência da esquerda para a direita; um `BorderLayout` exibe componentes em cinco regiões e um `GridLayout` exibe componentes em um formato de linhas e colunas.

A classe `Container` contém uma referência para um objeto `LayoutManager` (o objeto estratégia). Como a referência para interface (isto é, a referência para o objeto `LayoutManager`) pode armazenar referências para objetos de classes que implementam aquela interface (isto é, os objetos `FlowLayout`, `BorderLayout` ou `GridLayout`), o objeto `LayoutManager` pode fazer referência a um `FlowLayout`, `BorderLayout` ou `GridLayout` a qualquer momento. A classe `Container` pode mudar esta referência através do método `setLayout`, para selecionar layouts diferentes durante a execução.

A classe `FlowLayoutDemo` (Fig. 12.24) demonstra a aplicação do padrão *Strategy* – a linha 16 declara um novo objeto `FlowLayout` e a linha 19 invoca o método `setLayout` do objeto `Container` para atribuir o objeto `FlowLayout` ao objeto `Container`. Neste exemplo, o `FlowLayout` fornece a estratégia para dispor os componentes.

Template Method

O *padrão de projeto Template Method* também lida com algoritmos. O padrão de projeto *Strategy* permite que diversos objetos contenham algoritmos distintos. Entretanto, o padrão de projeto *Template Method* exige que todos os objetos compartilhem um único algoritmo definido por uma superclasse.

Por exemplo, considere o projeto da Fig. 13.26, que mencionamos na discussão do padrão de projeto *Observer*. Objetos das classes `TextDisplay`, `BarGraphDisplay` e `PieChartDisplay` usam o mesmo algoritmo básico para a aquisição e exibição dos dados – obter todos os extratos de conta do objeto `BankStatementData`, analisar os extratos de conta e depois exibir os extratos de conta. O padrão de projeto *Template Method* nos permite criar uma superclasse abstrata chamada `BankStatementDisplay` que oferece o algoritmo central para exibição dos dados. Neste exemplo, os métodos abstratos `getData`, `parseData` e `displayData` compõem o algoritmo. As classes `TextDisplay`, `BarGraphDisplay` e `PieChartDisplay` estendem a classe `BankStatementDisplay` para herdar o algoritmo, de modo que cada objeto pode usar o mesmo algoritmo. Cada subclasse de `BankStatementDisplay`, então, sobrescreve cada método de uma maneira específica para aquela subclasse, porque cada classe implementa os algoritmos de maneira diferente uma da outra. Por exemplo, as classes `TextDisplay`, `BarGraphDisplay` e `PieChartDisplay` poderiam obter e analisar os dados de maneira idêntica, mas cada classe exibe aqueles dados de maneira diferente.

O padrão de projeto *Template Method* permite estender o algoritmo para outras subclasse `BankStatementDisplay` – por exemplo, poderíamos criar classes, como `LineGraphDisplay` ou a classe `3DimensionalDisplay`, que usam o mesmo algoritmo herdado da classe `BankStatementDisplay`.

13.18.4 Conclusão

Nesta seção “Descobrindo padrões de projeto”, discutimos como os componentes Swing tiram proveito de padrões de projeto e como os desenvolvedores podem integrar os padrões de projeto com aplicativos Java com GUI. Na Seção 15.13 de “Descobrindo padrões de projeto”, discutimos padrões de projeto simultâneos, que são particularmente úteis para desenvolver sistemas *multithreaded*.

Resumo

- **JTextAreas** fornecem uma área para manipular múltiplas linhas de texto. Assim como a classe **JTextField**, a classe **JTextArea** herda de **JTextComponent**.
- Um evento externo (isto é, um evento gerado por um componente GUI diferente) normalmente indica quando o texto em uma **JTextArea** deve ser processado.
- Fornecemos barras de rolagem a uma **JTextArea** associando-a com um objeto **JScrollPane**.
- O método **getSelectedText** devolve o texto selecionado de uma **JTextArea**. O texto é selecionado arrastando o mouse sobre o texto desejado para destacá-lo.
- O método **setText** configura o texto em uma **JTextArea**.
- Para fornecer mudança automática de linha em uma **JTextArea**, associe esta a um **JScrollPane** com a regra de barra de rolagem horizontal **JScrollPane.HORIZONTAL_SCROLLBAR_NEVER**.
- As regras para as barras de rolagem vertical e horizontal de um **JScrollPane** são configuradas quando um **JScrollPane** é construído ou com os métodos **setHorizontalScrollBarPolicy** e **setVerticalScrollBarPolicy** da classe **JScrollPane**.
- Pode-se usar um **JPanel** como uma área dedicada de desenho que pode receber eventos de mouse e freqüentemente é estendida para criar novos componentes GUI.
- Os componentes do Swing que herdam da classe **JComponent** contêm o método **paintComponent**, que ajuda a desenhar adequadamente no contexto de uma GUI Swing. O método **paintComponent** de **JComponent** deve ser sobreescrito para chamar a versão da superclasse de **paintComponent** como o primeiro comando em seu corpo.
- As classes **JFrame** e **JApplet** não são subclasses de **JComponent**; portanto, elas não contêm o método **paintComponent** (elas têm o método **paint**).
- A chamada **repaint** para um componente GUI do Swing indica que o componente deve ser pintado logo que possível. O fundo do componente GUI é limpo somente se o componente for opaco. A maioria dos componentes do Swing é transparente por *default*. O método **setOpaque** de **JComponent** pode receber um argumento **boolean** para indicar se o componente é opaco (**true**) ou transparente (**false**). Os componentes GUI do pacote **java.awt** são diferentes dos componentes Swing no sentido que **repaint** resulta em uma chamada ao método **update** de **Component** (que limpa o fundo do componente) e **update** chama o método **paint** (em vez de **paintComponent**).
- O método **setTitle** exibe um **String** na barra de título de uma janela.
- O desenho em qualquer componente GUI é realizado com coordenadas que são medidas em relação ao canto superior esquerdo (0, 0) desse componente GUI.
- Os gerenciadores de layout freqüentemente utilizam o método **getPreferredSize** do componente GUI para determinar a largura e a altura preferidas de um componente ao desenhar esse componente como parte de uma GUI. Se um novo componente tem uma largura e uma altura preferidas, ele deve sobreescriver o método **getPreferredSize** para devolver essa largura e essa altura como um objeto da classe **Dimension** (pacote **java.awt**).
- O tamanho *default* de um objeto **JPanel** é 10 pixels de largura e 10 pixels de altura.
- Uma operação de arrastar o mouse inicia com um evento de pressionar o mouse. Todos os eventos subsequentes de arrasto do mouse (para os quais **mouseDragged** será chamado) são enviados para o componente GUI que recebeu o evento de pressionar o mouse original.
- **JSliders** (controles deslizantes) permitem ao usuário selecionar dentro de um intervalo de valores inteiros. Os **JSliders** podem exibir marcas de medidas principais, marcas de medida secundares e rótulos para as marcas de medida. Além disso, eles suportam aderência às marcas, na qual se posicionar o marcador entre duas marcas de medida faz o marcador aderir à marca de medida mais próxima.
- A maioria dos componentes GUI do Swing suporta interações com o usuário tanto pelo mouse como pelo teclado.
- Se um **JSlider** tem o foco, a tecla *seta para a esquerda* e a tecla *seta para a direita* fazem com que o marcador do **JSlider** diminua ou aumente uma unidade. A tecla *seta para baixo* e a tecla *seta para cima* também fazem com que o marcador do **JSlider** diminua ou aumente uma unidade, respectivamente. A tecla **PgDn** (página para baixo) e a tecla **PgUp** (página para cima) fazem com que o marcador do **JSlider** diminua ou aumente por incrementos de bloco de um décimo do intervalo de valores, respectivamente. A tecla *Home* move o marcador para o valor mínimo do **JSlider** e a tecla *End* move o marcador para o valor máximo do **JSlider**.

- **JSliders** têm uma orientação horizontal ou uma orientação vertical. Para um **JSlider** horizontal, o valor mínimo está na extrema esquerda e o valor máximo está na extrema direita do **JSlider**. Para um **JSlider** vertical, o valor mínimo está na extremidade inferior e o valor máximo está na extremidade superior do **JSlider**. A posição relativa do marcador indica o valor atual do **JSlider**.
- O método **setMajorTickSpacing** da classe **JSlider** configura o espaçamento para marcas de medida em um **JSlider**. O método **setPaintTicks** com um argumento **true** indica que as marcas de medida devem ser exibidas.
- **JSliders** geram **ChangeEvent**s (pacote `javax.swing.event`) quando o usuário interage com um **JSlider**. O **ChangeListener** (pacote `javax.swing.event`) define o método **stateChanged** que pode responder aos **ChangeEvent**s.
- O método **getValue** da classe **JSlider** devolve a posição atual do marcador.
- O **JFrame** é uma janela com uma barra de título e uma borda. A classe **JFrame** é uma subclasse de `java.awt.Frame` (que é uma subclasse de `java.awt.Window`).
- A classe **JFrame** suporta três operações quando o usuário fecha a janela. Por *default*, uma janela é oculta quando o usuário a fecha. Isso pode ser controlado com o método **setDefaultCloseOperation** de **JFrame**. A interface **WindowConstants** (pacote `javax.swing`) define três constantes para utilizar com esse método – **DISPOSE_ON_CLOSE**, **DO NOTHING_ON_CLOSE** e **HIDE_ON_CLOSE** (*o default*).
- Por *default*, uma janela só é exibida na tela quando seu método **setVisible** seja chamado com **true** como argumento. Uma janela também pode ser exibida através da chamada de seu método **show**.
- O tamanho de janela deve ser configurado com uma chamada ao método **setSize**. A posição de uma janela quando aparece na tela é especificada com o método **setLocation**.
- Todas as janelas geram eventos de janela quando o usuário as manipula. Os ouvintes de evento são registrados para eventos de janela com o método **addWindowListener** da classe **Window**. A interface **WindowListener** fornece sete métodos para tratar eventos de janela – **windowActivated** (chamado quando a janela se torna ativa clicando-se nela), **windowClosed** (chamado depois que a janela é fechada), **windowClosing** (chamado quando o usuário inicia o fechamento da janela), **windowDeactivated** (chamado quando outra janela se torna ativa), **windowIconified** (chamado quando o usuário minimiza uma janela), **windowDeiconified** (chamado quando uma janela é restaurada do seu estado minimizado) e **windowOpened** (chamado quando uma janela é exibida pela primeira vez na tela).
- Os argumentos de linha de comando são automaticamente passados a **main** como o **array** de **String**s chamado **args**. O primeiro argumento depois do nome da classe do aplicativo é o primeiro **String** no **array args** e o comprimento do **array** é o número total de argumentos de linha de comando.
- Os menus são uma parte integrante das GUIs. Os menus permitem ao usuário realizar ações sem “poluir” desnecessariamente uma interface gráfica com o usuário com componentes GUI extras.
- Na GUI do Swing, os menus podem ser anexados apenas aos objetos das classes que fornecem o método **setJMenuBar**. Duas dessas classes são **JFrame** e **JApplet**.
- As classes utilizadas para definir menus são **JMenuBar**, **JMenuItem**, **JMenu**, **JCheckBoxMenuItem** e a classe **JRadioButtonMenuItem**.
- O **JMenuBar** é um contêiner para menus.
- O **JMenuItem** é um componente GUI dentro de um menu que, quando selecionado, faz com que uma ação seja realizada. Pode-se utilizar **JMenuItem** para iniciar uma ação ou ser um submenu que fornece mais itens de menu que o usuário pode selecionar.
- O **JMenu** contém itens de menu e pode ser adicionado a uma **JMenuBar** ou a outros **JMenus** como submenus. Quando se clica num menu, o menu se expande para mostrar sua lista de itens.
- Quando um **JCheckBoxMenuItem** é selecionado, aparece uma marca de verificação à esquerda do item no menu. Quando o **JCheckBoxMenuItem** é selecionado novamente, a marca de verificação à esquerda do item no menu é removida.
- Quando vários **JRadioButtonMenuItem**s são mantidos como parte de um **ButtonGroup**, apenas um item no grupo pode ser selecionado de cada vez. Quando um **JRadioButtonMenuItem** é selecionado, aparece um círculo preenchido à esquerda do item do menu. Quando outro **JRadioButtonMenuItem** é selecionado, aparece o círculo preenchido à esquerda do item de menu previamente selecionado é removido.
- O método **setJMenuBar** de **JFrame** anexa uma barra de menus a um **JFrame**.
- O método **setMnemonic** de **AbstractButton** (herdado pela classe **JMenu**) especifica o mnemônico para um objeto **AbstractButton**. Pressionar a tecla **Alt** e o mnemônico realiza a ação do **AbstractButton** (no caso de um menu, abre o menu).
- Os caracteres mnemônicos normalmente são exibidos com um caractere de sublinhado.

- As caixas de diálogo podem ser modais ou não-modais. A caixa de diálogo modal não permite acessar qualquer outra janela no aplicativo até que a caixa de diálogo seja fechada. A caixa de diálogo não-modal permite que outras janelas sejam acessadas enquanto o diálogo é exibido. Por *default*, os diálogos exibidos com a classe **JOptionPane** são diálogos modais. A classe **JDialog** pode ser utilizada para criar seus próprios diálogos não-modais ou modais.
- O método **addSeparator** de **JMenu** adiciona uma linha separadora a um menu.
- Menus *pop-up* sensíveis ao contexto são criados com a classe **JPopupMenu**. Eles fornecem opções que são específicas do componente para o qual o evento de disparo do *pop-up* foi gerado. Na maioria dos sistemas, o evento de disparo do *pop-up* ocorre quando o usuário pressiona e libera o botão direito do mouse.
- O método **isPopupTrigger** de **MouseEvent** devolve **true** se o evento de disparo do *pop-up* ocorreu.
- O método **show** da classe **JPopupMenu** exibe um **JPopupMenu**. O primeiro argumento para o método **show** especifica o componente de origem, cuja posição ajuda a determinar onde o **JPopupMenu** aparecerá na tela. Os últimos dois argumentos são as coordenadas *x* e *y*, a partir do canto superior esquerdo do componente de origem, do ponto em que o **JPopupMenu** deve aparecer.
- A classe **UIManager** contém uma classe **public static** interna chamada **LookAndFeelInfo**, que é utilizada para manter as informações sobre uma determinada aparência e comportamento.
- O método **static getInstalledLookAndFeels** de **UIManager** obtém um *array* de objetos **UIManager.LookAndFeelInfo** que descrevem a aparência e o comportamento instalados.
- O método **static setLookAndFeel** de **UIManager** altera a aparência e o comportamento.
- O método **static updateComponentTreeUI** de **SwingUtilities** altera a aparência e o comportamento de cada componente associado ao seu argumento **Component** para uma nova aparência e comportamento.
- Muitos aplicativos atuais utilizam uma interface com múltiplos documentos (*multiple document interface – MDI*) [isto é, uma janela principal (frequentemente chamada janela-pai) que contém outras janelas (frequentemente chamadas janelas-filhas)] para gerenciar vários documentos abertos que estão sendo processados em paralelo.
- As classes **JDesktopPane** e **JInternalFrame** do Swing fornecem suporte para criar interfaces com múltiplos documentos.
- **BoxLayout** é um gerenciador de layout que permite que os componentes GUI sejam organizados da esquerda para a direita ou de cima para baixo em um contêiner. A classe **Box** define um contêiner com **BoxLayout** como seu gerenciador de layout *default* e fornece métodos estáticos para criar um **Box** com um **BoxLayout** horizontal ou vertical.
- **CardLayout** é um gerenciador de layout que empilha componentes como uma pilha de cartas. Cada contêiner na pilha pode utilizar qualquer gerenciador de layout. Somente o contêiner “na parte superior” da pilha é visível.
- O **GridLayout** é um gerenciador de layout semelhante ao **GridLayout**. Diferentemente de **GridLayout**, o tamanho de cada componente pode variar e os componentes podem ser adicionados em qualquer ordem.
- O método **static createHorizontalBox** de **Box** devolve um contêiner **Box** com um **BoxLayout** horizontal. O método **static createVerticalBox** da classe **Box** devolve um contêiner **Box** com um **BoxLayout** vertical.
- O método **static createVerticalStrut** de **Box** adiciona um suporte vertical a um contêiner. Um suporte vertical é um componente GUI invisível que tem uma altura fixa em *pixels* e é utilizada para garantir uma quantidade fixa de espaço entre componentes GUI. A classe **Box** também define o método **createHorizontalStrut** para **BoxLayouts** horizontais.
- O método **static createHorizontalGlue** de **Box** adiciona cola horizontal a um contêiner. A cola horizontal é um componente GUI invisível que pode ser utilizado entre componentes GUI de tamanho fixo para ocupar espaço adicional. A classe **Box** também define o método **createVerticalGlue** para **BoxLayouts** verticais.
- O método **static createRigidArea** de **Box** adiciona uma área rígida a um contêiner. A área rígida é um componente GUI invisível que sempre tem uma largura e uma altura fixas em *pixels*.
- O construtor **BoxLayout** recebe uma referência para o contêiner por meio do qual ele controla o layout e uma constante que indica se o layout é horizontal (**BoxLayout.X_AXIS**) ou vertical (**BoxLayout.Y_AXIS**).
- Os métodos **first**, **previous**, **next** e **last** de **CardLayout** são utilizados para exibir uma carta particular. O método **first** exibe a primeira carta. O método **previous** exibe a carta anterior. O método **next** exibe a próxima carta. O método **last** exibe a última carta.
- Para utilizar **GridBagLayout**, um objeto **GridBagConstraints** deve ser utilizado para especificar como um componente é colocado em um **GridBagLayout**.
- O método **setConstraints** da classe **GridBagLayout** recebe um argumento **Component** e um argumento **GridBagConstraints** e configura as limitações do **Component**.

Terminologia

<i>aderência a marcas</i>	<i>marcador de um JSlider</i>
<i>aparência de metal</i>	<i>marcas de medida secundárias</i>
<i>aparência e comportamento de janelas</i>	<i>menu</i>
<i>aparência e comportamento Motif</i>	<i>menu pop-up sensível ao contexto</i>
<i>aparência e comportamento plugável (PLAF)</i>	<i>método addSeparator da classe JMenu</i>
<i>área dedicada para desenho</i>	<i>método addWindowListener de Window</i>
<i>argumentos de linha de comando</i>	<i>método createHorizontalBox de Box</i>
<i>barra de menus</i>	<i>método createHorizontalGlue de Box</i>
<i>BoxLayout.X_AXIS</i>	<i>método createHorizontalStrut de Box</i>
<i>BoxLayout.Y_AXIS</i>	<i>método createRigidArea de Box</i>
<i>classe Box</i>	<i>método createVerticalBox de Box</i>
<i>classe ChangeEvent</i>	<i>método createVerticalGlue de Box</i>
<i>classe Dimension</i>	<i>método createVerticalStrut de Box</i>
<i>classe GridBagConstraints</i>	<i>método dispose da classe Window</i>
<i>classe JCheckBoxMenuItem</i>	<i>método first de CardLayout</i>
<i>classe JDesktopPane</i>	<i>método getClassName</i>
<i>classe JInternalFrame</i>	<i>método getInstalledLookAndFeels</i>
<i>classe JMenu</i>	<i>método getMinimumSize de Component</i>
<i>classe JMenuBar</i>	<i>método getPreferredSize de Component</i>
<i>classe JMenuItem</i>	<i>método getSelectedText</i>
<i>classe JPopupMenu</i>	<i>método getValue da classe JSlider</i>
<i>classe JRadioButtonMenuItem</i>	<i>método isPopupTrigger de MouseEvent</i>
<i>classe JSlider</i>	<i>método last da classe CardLayout</i>
<i>classe JTextArea</i>	<i>método next da classe CardLayout</i>
<i>classe JTextComponent</i>	<i>método paintComponent de JComponent</i>
<i>classe UIManager</i>	<i>método previous da classe CardLayout</i>
<i>classe UIManager.LookAndFeelInfo</i>	<i>método setConstraints</i>
<i>diálogo modal</i>	<i>método setDefaultCloseOperation</i>
<i>diálogo não-modal</i>	<i>método setHorizontalScrollBarPolicy</i>
<i>diretivas de barra de rolagem para um JScrollPane</i>	<i>método setJMenuBar</i>
<i>evento externo</i>	<i>método setLookAndFeel</i>
<i>gerenciador de leiaute BoxLayout</i>	<i>método setMajorTickSpacing</i>
<i>gerenciador de leiaute CardLayout</i>	<i>método setMnemonic de AbstractButton</i>
<i>gerenciador de leiaute GridBagLayout</i>	<i>método setOpaque da classe JComponent</i>
<i>GridBagConstraints.BOTH</i>	<i>método setPaintTicks da classe JSlider</i>
<i>GridBagConstraints.CENTER</i>	<i>método setSelected de AbstractButton</i>
<i>GridBagConstraints.EAST</i>	<i>método setTitle da classe Frame</i>
<i>GridBagConstraints.HORIZONTAL</i>	<i>método setVerticalScrollBarPolicy</i>
<i>GridBagConstraints.NONE</i>	<i>método show da classe JPopupMenu</i>
<i>GridBagConstraints.NORTH</i>	<i>método updateComponentTreeUI</i>
<i>GridBagConstraints.NORTHEAST</i>	<i>método windowActivated</i>
<i>GridBagConstraints.NORTHWEST</i>	<i>método windowClosed</i>
<i>GridBagConstraints.RELATIVE</i>	<i>método windowClosing</i>
<i>GridBagConstraints.REMAINDER</i>	<i>método windowDeactivated</i>
<i>GridBagConstraints.SOUTH</i>	<i>método windowDeiconified</i>
<i>GridBagConstraints.SOUTHEAST</i>	<i>método windowIconified</i>
<i>GridBagConstraints.SOUTHWEST</i>	<i>método windowOpened</i>
<i>GridBagConstraints.VERTICAL</i>	<i>mnemônico</i>
<i>GridBagConstraints.WEST</i>	<i>mudança automática de linha</i>
<i>interface ChangeListener</i>	<i>rótulos para marcas de medida</i>
<i>interface com múltiplos documentos (MDI)</i>	<i>submenu</i>
<i>interface WindowListener</i>	<i>super.paintComponent(g);</i>
<i>item de menu</i>	<i>suporte vertical</i>
<i>janela-filha</i>	<i>SwingConstants.HORIZONTAL</i>
<i>janela-pai</i>	<i>SwingConstants.VERTICAL</i>
<i>marca de medida</i>	<i>variável anchor de GridBagConstraints</i>
<i>marca de medida principal</i>	<i>variável fill de GridBagConstraints</i>

variável `gridheight`

variável `gridwidth`

variável `gridx` de `GridBagConstraints`

variável `gridy` de `GridBagConstraints`

variável `weightx` de `GridBagConstraints`

variável `weighty` de `GridBagConstraints`

`WindowConstants.DISPOSE_ON_CLOSE`

Exercícios de auto-revisão

- 13.1** Preencha as lacunas em cada uma das frases seguintes:

- A classe _____ é utilizada para criar um objeto de menu.
- O método _____ coloca uma barra separadora em um menu.
- Passar `false` para um método _____ de `TextArea` evita que seu texto seja modificado pelo usuário.
- Os eventos de `JSlider` são tratados pelo método da interface _____.
- A variável de instância _____ de `GridBagConstraints` é configurada como `CENTER` por *default*.

- 13.2** Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.

- Quando o programador cria um `JFrame`, no mínimo um menu deve ser criado e adicionado ao `JFrame`.
- A variável `fill` pertence à classe `GridBagLayout`.
- `JFrames` e `applets` não podem ser utilizados juntos no mesmo programa.
- O canto superior esquerdo de um `JFrame` ou `applet` tem uma coordenada de (0, 0).
- O texto de uma `JTextArea` é sempre apenas de leitura (*read-only*).
- A classe `JTextArea` é uma subclasse direta da classe `Component`.
- O layout *default* para um `Box` é `BoxLayout`.

- 13.3** Localize o(s) erro(s) em cada uma das instruções seguintes e explique como corrigi-lo(s).

```
a) JMenubar b;
b) mySlider = JSlider( 1000, 222, 100, 450 );
c) gbc.fill = GridBagConstraints.NORTHWEST; // configura fill
d) // sobrescreve para pintar em um componente personalizado Swing
   public void paintcomponent( Graphics g )
   {
      g.drawString( "HELLO", 50, 50 );
   }
e) // cria JFrame e a exibe
   JFrame f = new JFrame( "A Window" );
   f.setVisible( true );
```

Respostas aos exercícios de auto-revisão

- 13.1** a) `JMenu`. b) `addSeparator`. c) `setEditable`. d) `stateChanged, ChangeListener`. e) `anchor`.

- 13.2** a) Falsa. `JFrame` não exige nenhum menu.
 b) Falsa. A variável `fill` pertence à classe `GridBagConstraints`.
 c) Falsa. Eles podem ser utilizados juntos.
 d) Verdadeira.
 e) Falsa. `JTextAreas` são editáveis por *default*.
 f) Falsa. `JTextArea` deriva da classe `JTextComponent`.
 g) Verdadeira.

- 13.3** a) `JMenubar` deve ser `JMenuBar`.
 b) O primeiro argumento para o construtor deve ser `SwingConstants.HORIZONTAL` ou `SwingConstants.VERTICAL`, e o operador `new` deve ser utilizado depois do operador `=`.
 c) A constante deve ser `BOTH`, `HORIZONTAL`, `VERTICAL` ou `NONE`.
 d) `paintcomponent` deve ser `paintComponent` e o método deve chamar `super.paintComponent(g)` como sua primeira instrução.
 e) Também se deve chamar o método `setSize` de `JFrame` para determinar o tamanho da janela.

Exercícios

- 13.4** Preencha as lacunas em cada uma das frases seguintes:
- Uma área de desenho dedicada pode ser definida como uma subclasse de _____.
 - O **JMenuItem** que é um **JMenu** é chamado de _____.
 - JTextFields** e **JTextAreas** herdam diretamente da classe _____.
 - O método _____ anexa uma **JMenuBar** a um **JFrame**.
 - A classe contêiner _____ tem um leiaute **BoxLayout** por *default*.
 - O _____ gerencia um conjunto de janelas-filhas definidas com classe a **JInternalFrame**.
- 13.5** Diga se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- Os menus requerem um objeto **JMenuBar** para que possam ser anexados a um **JFrame**.
 - O objeto **JPanel** é capaz de receber eventos de mouse.
 - CardLayout** é o gerenciador de leiaute *default* para um **JFrame**.
 - O método **setEditable** é um método de **JTextComponent**.
 - O gerenciador de leiaute **GridBagLayout** implementa **LayoutManager**.
 - Os objetos **JPanel** são contêineres aos quais outros componentes GUI podem ser anexados.
 - A classe **JFrame** herda diretamente da classe **Container**.
 - JApplets** podem conter menus.
- 13.6** Localize o(s) erro(s) em cada uma das seguintes. Explique como corrigi-lo(s).
- `x.add(new JMenuItem("Submenu Color")); // cria submenu`
 - `container.setLayout(m = new GridbagLayout());`
 - `String s = JTextArea.getText();`
- 13.7** Escreva um programa que exibe um círculo de tamanho aleatório e calcula e exibe a área, o raio, o diâmetro e a circunferência. Utilize as seguintes equações: $diâmetro = 2 \times raio$, $área = \pi \times raio^2$, $circunferência = 2 \times \pi \times raio$. Utilize a constante **Math.PI** para pi (π). Todos os desenhos devem ser feitos em uma subclasse de **JPanel** e os resultados dos cálculos devem ser exibidos em uma **JTextArea** somente de leitura (*read-only*).
- 13.8** Aprimore o programa do Exercício 13.7 permitindo ao usuário alterar o raio com um **JSlider**. O programa deve funcionar para todos os raios no intervalo de 100 a 200. À medida que o raio muda, o diâmetro, a área e a circunferência devem ser atualizados e exibidos. O raio inicial deve ser 150. Utilize as equações de Exercício 13.7. Todos os desenhos devem ser feitos em uma subclasse de **JPanel** e os resultados dos cálculos devem ser exibidos em uma **JTextArea** somente de leitura.
- 13.9** Explore os efeitos de variar os valores **weightx** e **weighty** do programa da Fig. 13.20. O que acontece quando um componente tem um peso diferente de zero, mas não pode ocupar toda a área (isto é, o valor **fill** não é **BOTH**)?
- 13.10** Escreva um programa que utiliza o método **paintComponent** para desenhar o valor atual de um **JSlider** em uma subclasse de **JPanel**. Além disso, forneça um **JTextField** em que um valor específico possa ser digitado. O **JTextField** deve exibir o valor atual do **JSlider** durante todo o tempo. Deve-se usar um **JLabel** para identificar o **JTextField**. Os métodos **setValue** e **getValue** de **JSlider** devem ser utilizados. [Nota: o método **setValue** é um método **public** que não devolve um valor e recebe um argumento inteiro – o valor de **JSlider**, que determina a posição do marcador.]
- 13.11** Modifique o programa da Fig. 13.16 para utilizar uma única **JComboBox** em vez de quatro **JButtons** separados. Cada “carta” não deve ser modificada.
- 13.12** Modifique o programa da Fig. 13.16 adicionando um mínimo de duas novas “cartas” ao baralho.
- 13.13** Defina uma subclasse de **JPanel** chamada **MyColorChooser** que fornece três objetos **JSlider** e três objetos **JTextField**. Cada **JSlider** representa os valores de 0 a 255 para os componentes de vermelho, verde e azul de uma cor. Utilize os valores de vermelho, verde e azul como argumentos para o construtor de **Color** para criar um novo objeto **Color**. Exiba o valor atual de cada **JSlider** no **JTextField** correspondente. Quando o usuário altera o valor do **JSlider**, o **JTextField** deve ser alterado correspondentemente. Defina a classe **MyColorChooser** de modo que ela possa ser reutilizada em outros aplicativos ou *applets*. Utilize seu novo componente GUI como parte de um *applet* que exibe o valor de **Color** atual desenhando um retângulo preenchido.
- 13.14** Modifique a classe **MyColorChooser** do Exercício 13.13 para permitir que o usuário digite um valor inteiro em um **JTextField** para configurar o valor de vermelho, verde ou azul. Quando o usuário pressionar *Enter* no **JTextField**, o **JSlider** correspondente deve ser configurado com o valor apropriado.

13.15 Modifique o *applet* do Exercício 13.14 para desenhar a cor atual como um retângulo sobre uma instância de uma subclasse de **JPanel** chamada **DrawPanel**. A classe **DrawPanel** deve fornecer seu próprio método **paintComponent** para desenhar o retângulo e fornecer os métodos *set* para configurar os valores de vermelho, verde e azul para a cor atual. Quando qualquer método *set* for invocado para a classe **DrawPanel**, o objeto deve automaticamente repintar (**repaint**) a si próprio.

13.16 Modifique o *applet* do Exercício 13.15 para permitir que o usuário arraste o mouse através do **DrawPanel** a fim de desenhar uma forma com a cor atual. Permita ao usuário escolher que forma desenhar.

13.17 Modifique o programa do Exercício 13.16 para permitir que o programa seja executado como aplicativo. O código do *applet* existente só deve ser modificado adicionando-se um método **main** para dar partida ao aplicativo no seu próprio **JFrame**. Forneça ao usuário a possibilidade de terminar o aplicativo clicando no botão de fechamento da janela que é exibida e selecionando **Exit** do menu **File**. Utilize as técnicas mostradas na Fig. 13.9

13.18 (*Aplicativo de desenho completo*) Utilizando as técnicas desenvolvidas nos Exercícios 12.27 a 12.33 e Exercícios 13.13 a 13.17, crie um programa de desenho completo que possa ser executado como *applet* e como aplicativo. O programa deve utilizar os componentes GUI dos Capítulos 12 e 13 para permitir que o usuário selecione a forma, a cor e as características de preenchimento. Cada forma deve ser armazenada em um *array* de objetos **MyShape**, no qual **MyShape** é a superclasse em sua hierarquia de classes de forma (veja os Exercícios 9.28 e 9.29). Utilize um **JDesktopPane** e **JInternalFrames** para permitir que o usuário crie múltiplos desenhos separados em janelas-filhas separadas. Crie a interface com o usuário como uma janela-filha separada contendo todos os componentes GUI que permitem ao usuário determinar as características da forma a ser desenhada. O usuário então pode clicar em qualquer **JInternalFrame** para desenhar a forma.

13.19 Uma empresa paga seus empregados como gerentes (que recebem um salário fixo por semana), horistas (que recebem um salário/hora fixo pelas primeiras 40 horas trabalhadas e mais horas extras com 50% de acréscimo, isto é, 1,5 vez seu salário/hora, para as horas extras trabalhadas), comissionados (que recebem \$250 mais 5,7% bruto das vendas semanais) ou trabalhadores por produção (que recebem uma quantia fixa de dinheiro para cada item que eles produzem – cada trabalhador por produção trabalha apenas em um tipo de item nessa empresa). Escreva um aplicativo para calcular o pagamento semanal de cada empregado. Cada tipo de empregado tem seu próprio código de pagamento: gerentes têm código de pagamento 1, horistas têm código 2, comissionados têm código 3 e trabalhadores por produção têm código 4. Utilize um **switch** para calcular o pagamento de cada empregado com base no código de pagamento do empregado. Utilize um **CardLayout** para exibir os componentes GUI que permitem ao usuário digitar as informações de que o programa necessita para calcular o pagamento de cada empregado com base no código de pagamento.

Tratamento de exceções

Objetivos

- Entender exceções e tratamento de erros.
- Ser capaz de utilizar blocos `try` para delimitar o código em que uma exceção pode ocorrer.
- Ser capaz de disparar (`throw`) exceções.
- Utilizar os blocos `catch` para especificar tratadores de exceções.
- Utilizar o bloco `finally` para liberar recursos.
- Entender a hierarquia de exceções de Java.
- Criar exceções definidas pelo programador.

*É do senso comum capturar um método e experimentá-lo.
Se ele falhar, admita isso com franqueza e experimente outro. Mas acima de tudo, tente algo.*

Franklin Delano Roosevelt

*Oh, ponde fora a parte pior:
vivei mais pura com a outra metade.*

William Shakespeare

*If they're running and they don't look where they're going
I have to come out from somewhere and catch them.*

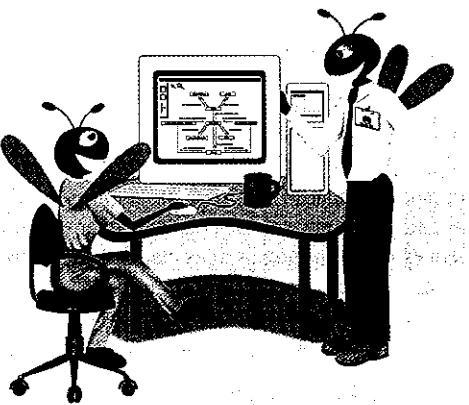
Jerome David Salinger

*Tentando por vezes desculpar-se de uma falta,
por fim mais grave a tornam.*

William Shakespeare

*Nunca me esqueço de um rosto, mas em seu caso farei uma
exceção.*

Groucho (Julius Henry) Marx



Sumário do capítulo

- 14.1 Introdução
- 14.2 Quando deve ser utilizado o tratamento de exceções
- 14.3 Outras técnicas de tratamento de erros
- 14.4 Princípios básicos de tratamento de exceções em Java
- 14.5 Blocos try
- 14.6 Disparando uma exceção
- 14.7 Capturando uma exceção
- 14.8 Exemplo de tratamento de exceções: divisão por zero
- 14.9 Disparando novamente uma exceção
- 14.10 Cláusula throws
- 14.11 Construtores, finalizadores e tratamento de exceções
- 14.12 Exceções e herança
- 14.13 O bloco finally
- 14.14 Utilizando printStackTrace e getMessage

*Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão
Exercícios*

14.1 Introdução

Neste capítulo, apresentamos o *tratamento de exceções*. A *exceção* é uma indicação de que ocorreu um problema durante a execução do programa. A extensibilidade de Java pode aumentar o número e os tipos de erros que podem ocorrer. Cada nova classe pode adicionar suas próprias possibilidades de erro. Os recursos apresentados aqui permitem escrever programas mais claros, mais robustos e mais tolerantes a falhas. Também analisamos quando o tratamento de exceções não deve ser utilizado.

O estilo e os detalhes do tratamento de exceções em Java apresentados neste capítulo são baseados em parte no trabalho de Andrew Koenig e Bjarne Stroustrup, como apresentado em seu artigo, “*Exception Handling for C++ (revised)*”, publicado no *Proceedings of the USENIX C++ Conference*, realizada em San Francisco em abril de 1990. O trabalho deles forma a base do tratamento de exceções em C++. Os projetistas de Java optaram por implementar um mecanismo de tratamento de exceções semelhante àquele utilizado em C++.

O código de tratamento de erro varia em natureza e quantidade entre sistemas de *software*, dependendo do aplicativo e de o *software* ser ou não um produto a ser liberado para uso por terceiros. Os produtos tendem a conter muito mais código de tratamento de erros do que um *software* “casual”.

Há muitos meios populares para lidar com erros. Mais comumente, o código de tratamento de erros é intercalado ao longo de todo o código de um sistema. Os erros são tratados nos lugares dentro do código em que os erros podem ocorrer. A vantagem dessa abordagem é que um programador que lê código pode ver o processamento de erros na vizinhança imediata do código e determinar se a verificação adequada de erros foi implementada.

O problema com esse esquema é que o código pode se tornar “poluído” com o processamento de erros. Fica mais difícil para um programador preocupado com o aplicativo em si ler o código e determinar se ele está funcionando corretamente. Isso pode tornar difícil entender e manter o aplicativo.

Alguns exemplos comuns de exceções são um subscripto de *array* fora dos limites, estouro aritmético (isto é, um valor fora do intervalo de valores representáveis), divisão por zero, parâmetros de método inválidos e esgotamento de memória.

Boa prática de programação 14.1



Utilizar o tratamento de exceções de Java permite remover o código de tratamento de erros da “linha principal” da execução do programa. Isso melhora a clareza de programa e torna mais fácil modificá-lo.

O tratamento de exceções é fornecido para permitir aos programas capturar e tratar erros em vez de deixá-los ocorrer e se sofrer as consequências. O tratamento de exceções é projetado para lidar com *erros síncronos* como uma tentativa de dividir por zero (que ocorre quando o programa executa a instrução de divisão). O tratamento de exceções não foi projetado para lidar com eventos *assíncronos* como o término de operações de E/S de disco, chegadas de mensagem pela rede, cliques de mouse, pressionamentos de teclas e coisas parecidas; esses são mais bem tratados por outros meios, como os *listeners* de eventos de Java.

O tratamento de exceções de Java permite a um programa capturar todas as exceções, todas as exceções de um certo tipo ou todas as exceções de tipos relacionados. Esta flexibilidade torna os programas mais robustos, reduzindo a probabilidade de que venham a não processar os problemas durante a execução do programa.

O tratamento de exceções é utilizado em situações em que o sistema pode se recuperar do mau funcionamento que causou a exceção. O procedimento de recuperação é chamado de *tratador de exceção*. O tratador de exceções pode ser definido no método que pode causar uma exceção ou em um método que o chama.



Observação de engenharia de software 14.1

Utilize exceções para maus funcionamentos que devem ser processados em um método diferente de onde eles são detectados. Utilize as técnicas de tratamento de erros convencionais para processamento local de erros em que um método é capaz de lidar com suas próprias exceções.

O tratamento de exceções é projetado para processar condições excepcionais – problemas que não acontecem freqüentemente, mas *podem* acontecer. É possível que o código de tratamento de exceções não seja otimizado para o mesmo nível de desempenho que outros elementos da linguagem.



Dica de desempenho 14.1

Quando não ocorre uma exceção, pouca ou nenhuma sobrecarga é imposta pela presença do código de tratamento da exceções. Quando ocorrem exceções, na verdade estas causam sobrecargas durante a execução.



Dica de teste e depuração 14.1

O tratamento de exceções ajuda a melhorar a tolerância a falhas de um programa.



Boa prática de programação 14.2

Utilizar tratamento de exceções padronizado de Java em vez de fazer os programadores utilizarem uma variedade de técnicas “caseiras” melhora a clareza do programa em projetos grandes.

Veremos que as exceções são objetos de classes derivadas da superclasse `Exception`. Mostraremos como lidar com exceções “não-capturadas”. Analisaremos como as exceções inesperadas são tratadas por Java. Mostraremos como os tipos relacionados de exceções podem ser representados por subclasses de exceção que são derivadas de uma superclasse comum de exceção.

O tratamento de exceções pode ser visto como outro meio de retornar o controle de um método ou sair de um bloco de código. Normalmente, quando ocorre uma exceção, ela é tratada por um chamador do método que gera a exceção, por um chamador desse chamador, ou tão longe para trás na pilha de chamadas quanto seja necessário para encontrar um tratador para essa exceção.



Observação de engenharia de software 14.2

O tratamento de exceções é particularmente adequado para os sistemas de componentes desenvolvidos separadamente. Tais sistemas são típicos de softwares do mundo real. O tratamento de exceções torna mais fácil combinar componentes e fazê-los trabalhar em conjunto de forma eficiente.



Observação de engenharia de software 14.3

Com outras linguagens de programação que não suportam tratamento de exceções, os programadores freqüentemente demoram para escrever o código de processamento de erro e, às vezes, simplesmente se esquecem de incluí-lo. Isso resulta em produtos menos robustos e, portanto, inferiores. Java força o programador a lidar com o tratamento de exceções desde a concepção inicial de um projeto. Ainda assim, o programador precisa de um esforço considerável para incorporar uma estratégia de tratamento de exceções aos projetos de software.

*Observação de engenharia de software 14.4*

É melhor incorporar sua estratégia de tratamento de exceções a um sistema desde a concepção inicial do processo de projeto. É difícil adicionar um tratamento de exceções eficiente depois que um sistema foi implementado.

14.2 Quando deve ser utilizado o tratamento de exceções

O tratamento de exceções deve ser utilizado

- Para processar situações excepcionais em que um método é incapaz de completar sua tarefa por razões que ele não pode controlar;
- Para processar exceções de componentes do programa que não são projetados para tratar essas exceções diretamente.
- Em projetos grandes, para tratar exceções de uma maneira uniforme em todo o projeto.

*Observação de engenharia de software 14.5*

O cliente de uma classe de biblioteca provavelmente terá em mente o processamento único de erro para uma exceção gerada na classe de biblioteca. É improvável que uma classe de biblioteca realize processamento de erros que atenda às necessidades particulares de todos os clientes. As exceções são um meio apropriado de lidar com os erros produzidos por classes de biblioteca.

14.3 Outras técnicas de tratamento de erros

Apresentamos várias maneiras de lidar com situações excepcionais antes deste capítulo. O programa pode ignorar alguns tipos de exceções. Isso pode ser devastador para produtos de *software* liberados para o público em geral ou para um *software* de uso especial necessário para situações de missão crítica. Mas para um *software* desenvolvido para seus próprios propósitos, é comum ignorar muitos tipos de erros. O programa pode ser dirigido para abortar ao encontrar uma situação excepcional. Isso evita que um programa seja executado até a conclusão e produza resultados incorretos. Para muitos tipos de erros, isso é uma boa estratégia. Uma estratégia assim é imprópria para aplicativos de missão crítica. As questões de recursos também são importantes aqui. Se o programa obtiver um recurso, o programa deve retornar aquele recurso antes da conclusão do programa.

*Erro comum de programação 14.1*

Abortar um programa pode deixar um recurso em um estado em que outros programas não seriam capazes de utilizar o recurso; portanto, teríamos uma assim chamada “perda de recurso”.

*Boa prática de programação 14.3*

Se seu método for capaz de tratar um dado tipo de exceção, então trate-a em vez de passar a exceção para outras regiões de seu programa. Isso torna os programas mais claros.

*Dica de desempenho 14.2*

Se um erro pode ser processado localmente em vez de disparar uma exceção, faça assim. Isso melhorará a velocidade de execução do programa. O tratamento de exceções é lento comparado ao processamento local.

14.4 Princípios básicos de tratamento de exceções em Java

Nessa seção, damos uma visão geral do processo de tratamento de exceções em Java. Ao longo do capítulo, apresentamos discussões detalhadas das etapas discutidas aqui.

O tratamento de exceções de Java é voltado para situações em que o método que detecta um erro é incapaz de lidar com ele. Esse método irá *disparar uma exceção*. Não há garantia de que haverá “qualquer coisa lá fora” (isto é, um *tratador de exceção* – código que é executado quando o programa detecta uma exceção) para processar aquele tipo de exceção. Se houver, a exceção será *capturada e tratada*. A *Dica de teste e depuração* a seguir descreve o que acontece se nenhum tratador de exceção apropriado puder ser localizado.



Dica de teste e depuração 14.2

Todos os applets Java e certos aplicativos Java baseiam-se em GUI. Alguns aplicativos Java não são baseados em GUI; esses são freqüentemente chamados de aplicativos de linha de comando (ou aplicativos de console). Quando uma exceção não é capturada em um aplicativo de linha de comando, o programa termina (isto é, Java encerra) depois que o tratador de exceção default é executado. Quando uma exceção não é capturada em um applet ou em um aplicativo baseado em GUI, a GUI continua a ser exibida e o usuário pode continuar utilizando o applet ou o aplicativo mesmo depois que o tratador de exceção padrão foi executado. Entretanto, o programa pode estar em um estado inconsistente e pode produzir resultados incorretos.

O programador inclui em um bloco **try** o código que pode gerar uma exceção e qualquer código que não deve ser executado se uma exceção ocorrer. O bloco **try** é imediatamente seguido por zero ou mais blocos **catch**. Cada bloco **catch** especifica o tipo de exceção que ele pode capturar e contém um tratador de exceção. Depois do último bloco **catch**, um bloco **finally** opcional fornece código que é sempre executado, independentemente de ocorrer ou não uma exceção. Como veremos, o bloco **finally** é um lugar ideal para código que libera recursos para evitar “perdas de recursos”. O bloco **try** deve ser seguido por um bloco **catch** ou por um bloco **finally**.

Quando um método dispara uma exceção, o controle de programa sai do bloco **try** e continua a execução no primeiro bloco **catch**. O programa pesquisa nos blocos **catch** em ordem, procurando um tratador apropriado (em breve discutiremos o que torna um tratador “apropriado”). Se o tipo da exceção disparada corresponder ao tipo de parâmetro em um dos blocos **catch**, o código daquele bloco **catch** é executado. Se um bloco **try** termina com sucesso, sem disparar qualquer exceção, o programa pula os tratadores de exceção para aquele bloco e retorna a execução depois do último bloco **catch**. Se um bloco **finally** aparece depois do último bloco **catch**, ele é executado independentemente de ter ocorrido ou não uma exceção.

Em uma definição de método, uma cláusula **throws** especifica as exceções que o método dispara. Esta cláusula aparece depois da lista de parâmetros e antes do corpo do método. A cláusula contém uma lista separada por vírgulas das exceções que o método poderá potencialmente disparar se ocorrer um problema enquanto o método está sendo executado. Tais exceções podem ser disparadas por instruções no corpo do método ou elas podem ser disparadas por métodos chamados no corpo. O ponto no qual o disparo ocorre é chamado de *ponto de disparo*.

Quando ocorre uma exceção, o bloco em que a exceção ocorreu expira (termina) – o controle do programa não pode retornar diretamente para o ponto de disparo. Java utiliza o *modelo de terminação no tratamento de exceções* em vez do *modelo de retomada no tratamento de exceções*. No modelo de retomada, o controle retornaria para o ponto em que a exceção ocorreu e retomaria a execução.

Quando ocorre uma exceção, é possível passar informações para o tratador de exceções a partir da vizinhança em que ocorreu a exceção. Essas informações são o tipo de objeto de exceção disparado ou informações colhidas da vizinhança em que a exceção ocorreu e colocadas no objeto disparado.

14.5 Blocos **try**

Uma exceção que ocorre em um bloco **try** normalmente é capturada por um tratador de exceções especificado por um bloco **catch** imediatamente após aquele bloco **try**, como em

```
try {
    instruções que podem disparar uma exceção
}
catch( TipoDeExceção referênciaParaExceção ) {
    instruções para processar uma exceção
}
```

O bloco **try** pode ser seguido por zero ou mais blocos **catch**. Se um bloco **try** é executado e nenhuma exceção é disparada, todos os tratadores de exceções são pulados e o controle é retomado na primeira instrução depois do último tratador de exceções. Se um bloco **finally** (apresentado na Seção 14.13) segue o último bloco **catch**, o código no bloco **finally** é executado independentemente de uma exceção ser ou não disparada. Observe que um tratador de exceção não pode acessar objetos definidos no bloco **try** correspondente, porque o bloco **try** expira antes que o tratador comece a ser executado.



Erro comum de programação 14.2

É um erro de sintaxe separar com outro código os tratadores **catch** que correspondem a um bloco **try** particular.

14.6 Disparando uma exceção

A instrução **throw** é executada para indicar que uma exceção ocorreu (isto é, um método não pode ser completado com sucesso). Esse processo se chama *disparar uma exceção*. A instrução **throw** especifica um objeto a ser disparado. O operando de um **throw** pode ser de qualquer classe derivada da classe **Throwable** (pacote `java.lang`). As duas subclasses imediatas da classe **Throwable** são **Exception** e **Error**. **Errors** são problemas de sistema particularmente sérios que geralmente não devem ser capturados. **Exceptions** são causadas por problemas que devem ser capturados e processados durante a execução do programa para torná-lo mais robusto. Se o operando de **throw** é um objeto da classe **Exception**, ele é chamado de *objeto de exceção*.



Dica de teste e depuração 14.3

Quando **toString** é invocado para qualquer objeto **Throwable**, seu **String** resultante inclui o **String** descriptivo que foi fornecido para o construtor ou simplesmente o nome da classe, se nenhum **String** foi fornecido.



Dica de teste e depuração 14.4

Um objeto pode ser disparado sem conter informações sobre o problema que ocorreu. Nesse caso, simplesmente saber que uma exceção de um tipo particular ocorreu pode fornecer informações suficientes para o tratador processar o problema corretamente.

Quando uma exceção é disparada, o controle sai do bloco **try** atual e prossegue para um tratador **catch** apropriado (se existir algum) depois daquele bloco **try**. É possível que o ponto de disparo esteja em um escopo profundamente aninhado dentro de um bloco **try**; ainda assim, o controle prosseguirá para o tratador **catch**. Também é possível que o ponto de disparo esteja em uma chamada de método profundamente aninhada; mesmo assim, o controle prosseguirá para o tratador **catch**.

O bloco **try** pode parecer não conter a verificação de erros e não incluir instruções **throw**, mas os métodos chamados a partir do bloco **try** podem disparar exceções. Além disso, as instruções em um bloco **try** que não invocam métodos podem causar exceções. Por exemplo, uma instrução que utiliza subscritos para acessar elementos de um objeto **array** dispara uma **ArrayIndexOutOfBoundsException** se a instrução especificar um subscrito de **array** inválido. Qualquer chamada de método pode invocar código que pode disparar uma exceção ou chamar um outro método que dispara uma exceção.

14.7 Capturando uma exceção

Os tratadores de exceções estão contidos em blocos **catch**. Cada bloco **catch** inicia com a palavra-chave **catch** seguida por parênteses que contêm um nome de classe (que especifica o tipo de exceção a ser capturada) e um nome de parâmetro. O tratador pode fazer referência ao objeto disparado através desse parâmetro. Depois dele há um bloco que delimita o código de tratamento de exceções. Quando um tratador captura uma exceção, o código no bloco **catch** é executado.



Erro comum de programação 14.3

Podem ocorrer erros de lógica se você supuser que, depois de uma exceção ser processada, o controle retornará para a primeira instrução depois do **throw**. O controle do programa continua com a primeira instrução após os tratadores **catch**.



Erro comum de programação 14.4

Especificar uma lista de parâmetros separados por vírgulas para um **catch** é um erro de sintaxe. O **catch** pode ter apenas um único argumento.



Erro comum de programação 14.5

É um erro de sintaxe capturar o mesmo tipo em dois blocos `catch` diferentes associados com um bloco `try` em particular.

O `catch` que captura um objeto `Throwable`

```
catch( Throwable throwable )
```

captura todas as exceções e erros. Similarmente, o `catch` que captura um objeto `Exception`

```
catch( Exception exception )
```

captura todas as exceções. Em geral, os programas não definem os tratadores de exceção para o tipo `Throwable`, porque `Errors` normalmente não devem ser capturados em um programa.



Erro comum de programação 14.6

Colocar `catch(Exception exception)` antes de outros blocos `catch` que capturam tipos específicos de exceções impede que esses blocos sejam executados; o tratador de exceções que captura o tipo `Exception` deve ser colocado por último na lista de tratadores de exceções que segue um bloco `try`, ou ocorre um erro de sintaxe.

É possível que o bloco `try` não tenha um tratador `catch` correspondente que coincida com um objeto disparado em particular. Isso faz com que a procura por um tratador `catch` correspondente continue no próximo bloco `try` que o envolve. À medida que esse processo continua, em algum momento o programa pode determinar que não há um tratador na pilha de execução que corresponda ao tipo do objeto disparado. Nesse caso, o aplicativo não-baseado em GUI é finalizado – *applets* e aplicativos baseados em GUI retornam para seu processamento regular de eventos. Embora *applets* e aplicativos baseados em GUI continuem a ser executados, eles podem ser executados incorretamente.



Observação de engenharia de software 14.6

Se você sabe que um método pode disparar uma exceção, inclua o código apropriado de tratamento de exceções em seu programa. Isto tornará o seu programa mais robusto.



Boa prática de programação 14.4

Leia a documentação on-line da API para um método antes de usar aquele método em um programa. A documentação específica as exceções disparadas pelo método (se houver alguma) e indica as razões pelas quais tais exceções podem ocorrer.



Boa prática de programação 14.5

Leia a documentação on-line da API para uma classe de exceção antes de escrever código de tratamento de exceção para aquele tipo de exceção. A documentação de uma classe de exceção geralmente contém razões em potencial para que tais exceções ocorram durante a execução do programa.

É possível que vários tratadores de exceções forneçam uma correspondência aceitável com o tipo da exceção. Isso pode acontecer por várias razões: pode haver um tratador `catch(Exception exception)` que “captura tudo” e que capturará qualquer exceção. Ademais, os relacionamentos de herança permitem que um objeto de subclasse seja capturado por um tratador que especifique o tipo da subclasse ou por tratadores que especificarem os tipos de quaisquer superclasses dessa classe. O primeiro tratador de exceções que corresponder ao tipo da exceção é executado – todos os outros tratadores de exceção para o bloco `try` correspondente são ignorados.



Observação de engenharia de software 14.7

Se vários tratadores correspondem ao tipo de uma exceção e se cada um desses trata a exceção de forma diferente, então a ordem dos tratadores afetará a maneira como a exceção é tratada.



Erro comum de programação 14.7

Causará um erro de sintaxe se um `catch` que captura um objeto de superclasse for colocado antes de um `catch` para tipos de subclasses daquela classe.

Às vezes, o programa pode processar muitos tipos de exceções intimamente relacionados. Em vez de fornecer tratadores `catch` separados para cada um, o programador pode fornecer um único tratador `catch` para um grupo de exceções.

O que acontece quando ocorre uma exceção em um tratador de exceções? O bloco `try` que percebeu a exceção expira antes que o tratador de exceções comece a ser executado, assim as exceções que ocorrem em um tratador de exceções são processadas por tratadores `catch` para um bloco `try` mais externo. O bloco `try` externo espera erros que ocorrem nos tratadores `catch` do bloco `try` original. O bloco `try` mais externo é um bloco `try` que contém uma seqüência `try/catch` completa ou um bloco `try` em um método que chamou.

Os tratadores de exceções podem ser escritos de várias maneiras. Eles podem disparar novamente uma exceção (como veremos na Seção 14.9). Podem converter um tipo de exceção em outro disparando um tipo diferente de exceção. Podem realizar qualquer recuperação necessária e retomar a execução depois do último tratador de exceções. Podem ver a situação que causa o erro, remover a causa do erro e tentar novamente chamando o método original que causou uma exceção. Podem retornar um valor de estado para seu ambiente, etc.

Não é possível retornar para o ponto de disparo usando uma instrução `return` em um tratador `catch`. Esse `return` simplesmente devolve o controle para o método que chamou o método contendo o bloco `catch`. Novamente, o ponto de disparo está em um bloco que expirou, de modo que retornar através de uma instrução `return` não faria sentido.

Observação de engenharia de software 14.8



Outra razão para não utilizar exceções para o fluxo de controle convencional é que essas exceções “adicionalis” podem “entrar no caminho” de verdadeiras exceções do tipo erro. Fica mais difícil para o programador monitorar o número maior de casos de exceção resultante. Situações excepcionais devem ser raras, não lugares-comuns.

Erro comum de programação 14.8



Supor que uma exceção disparada de um tratador `catch` será processada por esse tratador ou qualquer outro tratador associado com o mesmo bloco `try` pode levar a erros de lógica.

14.8 Exemplo de tratamento de exceções: divisão por zero

Agora vamos analisar um exemplo simples de tratamento de exceções. O aplicativo das Figuras 14.1 e 14.2 utiliza `try`, `throw` e `catch` para detectar, indicar e tratar exceções. O aplicativo exibe dois `JTextFields` em que o usuário pode digitar inteiros. Quando o usuário pressiona a tecla *Enter* no segundo `JTextField`, o programa chama o método `actionPerformed` para ler os dois inteiros dos `JTextFields` e passa os inteiros para o método `quotient` que calcula o quociente dos dois valores e retorna um resultado `double`. Se o usuário digita 0 no segundo `JTextField`, o programa utiliza uma exceção para indicar que o usuário está tentando uma divisão por zero. Além disso, se o usuário digita um valor que não é um inteiro em qualquer `JTextField`, ocorre uma `NumberFormatException`. Em exemplos anteriores, que lêem valores numéricos digitados pelo usuário, simplesmente supusemos que o usuário iria digitar um valor inteiro apropriado. Entretanto, os usuários às vezes cometem enganos. Este programa demonstra como capturar a `NumberFormatException` que ocorre quando o programa tenta converter um `String` que não representa um valor inteiro para um valor `int` com o método `parseInt` de `Integer`.

Antes de discutirmos o programa, considere os exemplos de execução mostrados nas cinco janelas de saída. A primeira janela mostra uma execução bem-sucedida. O usuário digitou os valores 100 e 7. O terceiro `JTextField` mostra o resultado da divisão realizada pelo método `quotient`. Na segunda janela de saída, o usuário digitou o string “hello” no segundo `JTextField`. Quando o usuário pressiona *Enter* no segundo `JTextField`, exibe-se um diálogo de mensagem de erro para indicar que se deve digitar um inteiro. Nas últimas duas janelas, digita-se um denominador zero e o programa detecta o problema, dispara uma exceção e emite uma mensagem de diagnóstico apropriada.

Agora vamos comentar o programa, começando pela classe `DivideByZeroException` da Fig. 14.1. Java pode testar se ocorreu uma divisão por zero quando os valores na divisão são ambos inteiros. Se descobrir uma tentativa de dividir por zero em aritmética inteira, Java dispara um `ArithmaticException`. Entretanto, nosso programa executa uma divisão de ponto flutuante de dois inteiros fazendo a coerção do primeiro inteiro para um `double` antes de executar o cálculo. Java permite divisão por zero em ponto flutuante. O resultado é um valor infinito positivo ou negativo (as classes `Float` e `Double` do pacote `java.lang` fornecem constantes que representam

estes valores). Mesmo que Java permita divisão por zero em ponto flutuante, gostaríamos de usar tratamento de exceções neste exemplo para indicar para o usuário de nosso programa que ele está tentando dividir por zero.

Como veremos, o método `quotient` dispara uma exceção quando ele recebe zero como seu segundo argumento. O método pode disparar uma exceção de qualquer tipo existente na Java API ou de um tipo específico para o programa. Neste exemplo, demonstramos a definição de um novo tipo de exceção. Na verdade, poderíamos usar a classe `ArithmetricException` (pacote `java.lang`) com uma mensagem de erro personalizada neste programa.



Boa prática de programação 14.6

Associar cada tipo de mau funcionamento sério durante a execução com uma classe `Exception` com um nome adequado melhora a clareza do programa.



Observação de engenharia de software 14.9

Se possível, use um tipo de exceção existente em vez de criar uma nova classe. A Java API contém muitos tipos de exceções que podem ser adequados para seu programa..

```

1 // Fig. 14.1: DivideByZeroException.java
2 // Definição da classe DivideByZeroException.
3 // Usada para disparar uma exceção quando
4 // se tenta uma divisão por zero.
5 public class DivideByZeroException extends ArithmetricException {
6
7     // construtor sem argumentos especifica mensagem de erro default
8     public DivideByZeroException()
9     {
10         super( "Attempted to divide by zero" );
11     }
12
13    // construtor para permitir mensagem de erro personalizada
14    public DivideByZeroException( String message )
15    {
16        super( message );
17    }
18
19 } // fim da classe DivideByZeroException

```

Fig. 14.1 Classe de exceção `DivideByZeroException`.

A classe `DivideByZeroException` estende a classe `ArithmetricException`. Escolhemos estender a classe `ArithmetricException` porque a divisão por zero ocorre durante uma operação aritmética. Como qualquer outra classe, uma classe de exceção pode conter variáveis de instância e métodos. A classe de exceção típica contém somente dois construtores – um que não recebe argumentos e especifica uma mensagem de exceção *default* e outro que recebe uma mensagem de exceção personalizada como um `String`. O construtor *default* (linhas 8 a 11) especifica o `string` `"Attempted to divide by zero"` como mensagem de exceção que indica o que deu errado. Esse `string` é passado para o construtor da superclasse para inicializar mensagem de erro associada com o objeto de exceção. O outro construtor (linhas 14 a 17) passa seu argumento – um `string` de mensagem de erro personalizada – para o construtor da superclasse.



Observação de engenharia de software 14.10

Ao definir seu próprio tipo de exceção, derive-o de um tipo de exceção relacionado existente na Java API. Será necessário investigar as exceções existentes na Java API. Se os tipos existentes não forem apropriados para derivar sua subclasse, a nova classe de exceção deve estender `Exception`, se o cliente de seu código for obrigado a tratar a exceção, ou estender `RunTimeException`, se o cliente de seu código tiver a opção de ignorar a exceção.

Agora considere o aplicativo `DivideByZeroTest` (Fig. 14.2). O construtor do aplicativo (linhas 21 a 51) constrói uma interface gráfica com o usuário com três `JLabels` (todos alinhados à direita) e três `JTextFields` e registra o objeto `DivideByZeroTest` como o `ActionListener` para `JTextField inputField2`.

```

1 // Fig. 14.2: DivideByZeroTest.java
2 // Um exemplo simples de tratamento de exceções.
3 // Verificando se não ocorreu um erro de divisão por zero.
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8 import java.text.DecimalFormat;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 public class DivideByZeroTest extends JFrame
14     implements ActionListener {
15
16     private JTextField inputField1, inputField2, outputField;
17     private int number1, number2;
18     private double result;
19
20     // configura a GUI
21     public DivideByZeroTest()
22     {
23         super( "Demonstrating Exceptions" );
24
25         // obtém painel de conteúdo e configura seu layout
26         Container container = getContentPane();
27         container.setLayout( new GridLayout( 3, 2 ) );
28
29         // configura rótulo e inputField1
30         container.add(
31             new JLabel( "Enter numerator ", SwingConstants.RIGHT ) );
32         inputField1 = new JTextField( 10 );
33         container.add( inputField1 );
34
35         // configura rótulo e inputField2; registra listener
36         container.add(
37             new JLabel( "Enter denominator and press Enter ",
38                         SwingConstants.RIGHT ) );
39         inputField2 = new JTextField( 10 );
40         container.add( inputField2 );
41         inputField2.addActionListener( this );
42
43         // configura rótulo e outputField
44         container.add(
45             new JLabel( "RESULT ", SwingConstants.RIGHT ) );
46         outputField = new JTextField();
47         container.add( outputField );
48
49         setSize( 425, 100 );
50         setVisible( true );
51     }
52
53     // processa eventos da GUI
54     public void actionPerformed( ActionEvent event )

```

Fig. 14.2 Um exemplo simples de tratamento de exceções com divisão por zero (parte 1 de 3).

```

55     {
56         DecimalFormat precision3 = new DecimalFormat( "0.000" );
57
58         outputField.setText( "" ); // limpa outputField
59
60         // lê dois números e calcula o quociente
61         try {
62             number1 = Integer.parseInt( inputField1.getText() );
63             number2 = Integer.parseInt( inputField2.getText() );
64
65             result = quotient( number1, number2 );
66             outputField.setText( precision3.format( result ) );
67         }
68
69         // processa dados de entrada formatados de maneira imprópria
70         catch ( NumberFormatException numberFormatException ) {
71             JOptionPane.showMessageDialog( this,
72                 "You must enter two integers",
73                 "Invalid Number Format",
74                 JOptionPane.ERROR_MESSAGE );
75         }
76
77         // processa tentativas de dividir por zero
78         catch ( ArithmeticException arithmeticException ) {
79             JOptionPane.showMessageDialog( this,
80                 arithmeticException.toString(),
81                 "Arithmetic Exception",
82                 JOptionPane.ERROR_MESSAGE );
83         }
84     }
85
86     // método quotient demonstrado disparando uma exceção
87     // quando ocorre um erro de divisão por zero
88     public double quotient( int numerator, int denominator )
89         throws DivideByZeroException
90     {
91         if ( denominator == 0 )
92             throw new DivideByZeroException();
93
94         return ( double ) numerator / denominator;
95     }
96
97     // executa o aplicativo
98     public static void main( String args[] )
99     {
100         DivideByZeroTest application = new DivideByZeroTest();
101
102         application.setDefaultCloseOperation(
103             JFrame.EXIT_ON_CLOSE );
104     }
105
106 } // fim da classe DivideByZeroTest

```

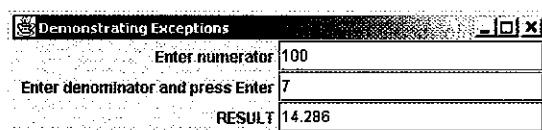


Fig. 14.2 Um exemplo simples de tratamento de exceções com divisão por zero (parte 2 de 3).

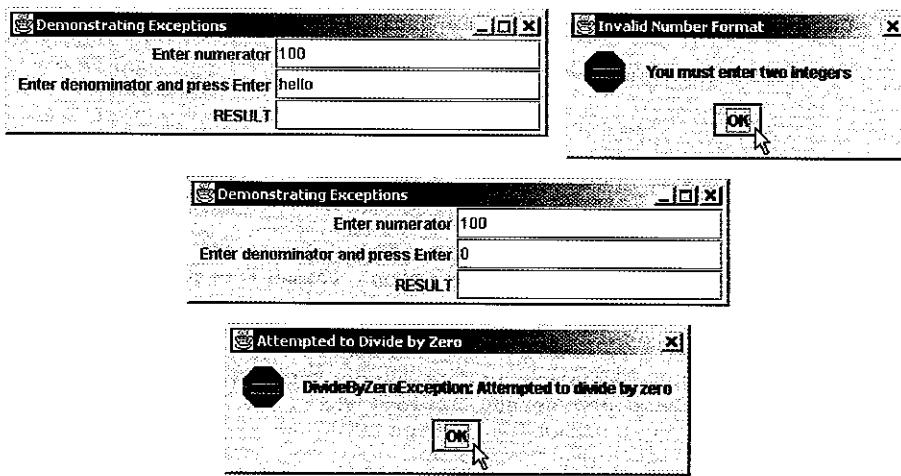


Fig. 14.2 Um exemplo simples de tratamento de exceções com divisão por zero (parte 3 de 3).

Quando o usuário digita o denominador e pressiona a tecla *Enter*, o programa chama o método `actionPerformed` (linhas 54 a 84). Em seguida, o método `actionPerformed` prossegue com um bloco `try` (linhas 61 a 67) que inclui o código que pode disparar uma exceção e qualquer código que não deva ser executado se ocorrer uma exceção. Cada uma das instruções que lêem os inteiros dos `JTextFields` (linhas 62 e 63) utiliza o método `Integer.parseInt` para converter os `Strings` em valores `int`. O método `parseInt` dispara uma `NumberFormatException` se o seu argumento `String` não for um inteiro válido. A divisão que pode causar o erro de divisão por zero não é executada explicitamente no bloco `try`. Em vez disso, a chamada para o método `quotient` (linha 65) invoca o código que tenta a divisão. O método `quotient` (linhas 89 a 96) dispara o objeto `DivideByZeroException`, como veremos em breve. Em geral, os erros podem emergir por meio de código explicitamente mencionado em um bloco `try`, por chamadas para um método ou mesmo por chamadas de métodos profundamente aninhadas iniciadas por código em um bloco `try`.

O bloco `try` neste exemplo é seguido por dois blocos `catch` – as linhas 70 a 75 contêm o tratador de exceção para a `NumberFormatException` e as linhas 78 a 83 contêm o tratador de exceção para a `DivideByZeroException`. Em geral, quando detecta uma exceção durante a execução de um bloco `try`, o programa captura a exceção em um bloco `catch` que especifica um tipo de exceção apropriado (isto é, o tipo no `catch` coincide exatamente com o tipo da exceção disparada ou é uma superclasse do tipo da exceção disparada). Na Fig. 14.2, o primeiro bloco `catch` especifica que ele capturará objetos de exceção do tipo `NumberFormatException` (esse tipo corresponde ao tipo do objeto exceção disparado no método `Integer.parseInt`) e o segundo bloco `catch` especifica que ele capturará objetos de exceção do tipo `ArithmeticException` (esse tipo é uma superclasse do objeto exceção disparado no método `quotient`). Apenas o tratador `catch` correspondente é executado quando ocorre uma exceção. Nossos dois tratadores de exceções simplesmente exibem um diálogo de mensagem de erro, mas os tratadores de exceções podem ser mais elaborados que isso. Depois de executar um tratador de exceções, o controle do programa segue para primeira instrução depois do último bloco `catch` (ou no bloco `finally`, se houver um).

Se o código no bloco `try` não dispara uma exceção, os tratadores `catch` são pulados e a execução é retomada na primeira linha do código depois dos tratadores `catch` (ou no bloco `finally`, se houver um). Na Fig. 14.2, o método `actionPerformed` simplesmente retorna, mas o programa poderia continuar executando mais instruções depois dos blocos `catch`.

Dica de teste e depuração 14.5



Com o tratamento de exceções, o programa pode continuar rodando depois de tratar um problema. Isso ajuda a assegurar aplicativos robustos que contribuem para o que se chama de computação de missão crítica ou computação de negócios críticos.

Agora vamos examinar o método `quotient` (linhas 89 a 96). Quando a estrutura `if` determina que `denominator` é zero, o corpo do `if` executa uma instrução `throw` que cria e dispara um novo objeto `DivideByZeroException`. Esse objeto será capturado pelo bloco `catch` (linhas 78 a 83) que especifica o tipo `DivideByZeroException` depois do bloco `try`. O bloco `catch` especifica o nome de parâmetro `arithmeticException` para receber o objeto de exceção disparado. O tratador de `ArithmeticException` converte a exceção para um `String` que usa `toString` e passa esse `String` como a mensagem a exibir no diálogo de mensagem de erro.

Se `denominator` não for zero, `quotient` não dispara uma exceção. Em vez disso, `quotient` executa a divisão e retorna o resultado da divisão para o ponto de invocação do método `quotient` no bloco `try` (linha 65). A linha 66 exibe o resultado do cálculo no terceiro `JTextField`. Nesse caso, o bloco `try` é completado com sucesso, de modo que o programa pula os blocos `catch` e o método `actionPerformed` completa a execução normalmente.

Observe que, quando `quotient` dispara a `DivideByZeroException`, o bloco de `quotient` expira (isto é, o método termina). Isso faria com que qualquer uma de suas variáveis locais fossem destruídas – os objetos que eram mencionados pelas variáveis locais no bloco teriam suas contagens de referência decrementadas adequadamente (e possivelmente seriam marcados para coleta de lixo). Além disso, o bloco `try` do qual o método foi chamado também expira antes que a linha 66 possa ser executada. Aqui também, se houvesse variáveis locais criadas no bloco `try` antes de a exceção ser disparada, essas variáveis seriam destruídas.

Se uma `NumberFormatException` é gerada pelas linhas 62 e 63, o bloco `try` expira e a execução continua com o tratador de exceções na linha 70, que exibe uma mensagem de erro para dizer ao usuário para digitar inteiros. Então, o método `actionPerformed` continua com a próxima instrução válida depois dos blocos `catch` (nesse exemplo, o método termina).

14.9 Disparando novamente uma exceção

É possível que o tratador `catch` que captura uma exceção decida que não é capaz de processar a exceção ou talvez ele queira deixar que algum outro tratador `catch` trate a exceção. Nesse caso, o tratador que recebeu a exceção pode disparar novamente a exceção com a instrução

```
throw referênciaParaExceção;
```

onde `referênciaParaExceção` é o nome do parâmetro para a exceção no tratador `catch`. O `throw`, assim, dispara novamente a exceção para o próximo bloco `try` que o envolve.

Mesmo se um tratador conseguir processar uma exceção, e independentemente de fazer ou não qualquer processamento dessa exceção, o tratador ainda pode disparar novamente a exceção para processamento adicional fora do tratador. Uma exceção disparada novamente é detectada pelo próximo bloco `try` que a envolve e é tratada por um tratador de exceção listado depois desse bloco `try` mais externo.

14.10 Cláusula `throws`

A cláusula `throws` lista as exceções que podem ser disparadas por um método, como em

```
int nomeDeFunção( listaDeParâmetros )
    throws TipoDeExceção1, TipoDeExceção2, TipoDeExceção3, ...
{
    // corpo do método
}
```

Os tipos de exceções que são disparadas por um método são especificados na definição do método com uma lista separada por vírgulas em uma cláusula `throws`. Um método pode disparar objetos das classes indicadas ou pode disparar objetos de suas subclasses.

Algumas exceções podem ocorrer em qualquer ponto durante a execução do programa. Muitas dessas exceções podem ser evitadas por codificação adequada. Trata-se de exceções durante a execução e elas derivam da classe `RuntimeException`. Por exemplo, se o programa tentar acessar um subscrito de `array` fora do intervalo, ocorre uma exceção do tipo `ArrayIndexOutOfBoundsException` (derivada de `RuntimeException`). O programa evidentemente pode evitar esse problema; portanto, é uma exceção durante a execução.

Outra exceção em tempo de execução ocorre quando o programa cria uma referência para o objeto, mas ainda não criou um objeto e o associou à referência. Tentar utilizar uma referência `null` faz com que uma `NullPointerException` seja disparada. Evidentemente, o programa pode evitar essa circunstância; portanto, é uma exceção em tempo de execução. Outra exceção durante a execução é uma coerção inválida, que dispara uma `ClassCastException`.

Há uma variedade de exceções que não são `RuntimeExceptions`. Duas das mais comuns são `InterruptedExceptions` (veja o Capítulo 15) e `IOExceptions` (veja o Capítulo 16).

Nem todos os erros e exceções que podem ser disparados a partir de um método precisam ser listados na cláusula `throws`. Os `Errors` não precisam ser listados, nem as `RuntimeExceptions` (as exceções evitáveis). `Errors` são problemas sérios de sistema que podem ocorrer em quase qualquer lugar, e a maioria dos programas não é capaz de se recuperar deles. Os métodos devem processar `RuntimeExceptions` capturadas em seu corpo diretamente, em vez de passá-las adiante para outros componentes do programa. Se um método dispara quaisquer exceções que não sejam `RuntimeExceptions`, ele deve especificar aqueles tipos de exceção em sua cláusula `throws`.



Observação de engenharia de software 14.11

Se uma exceção que não seja uma `RuntimeException` for disparada por um método, ou se esse método chamar métodos que disparam exceções que não sejam uma `RuntimeException` cada uma dessas exceções deve ser declarada na cláusula `throws` desse método ou ser capturada em um `try/catch` nesse método.

Java distingue `Exceptions verificadas` de `RuntimeExceptions não-verificadas` e `Errors`. As exceções verificadas de um método precisam ser listadas na cláusula `throws` desse método. `Errors` e `RuntimeExceptions` podem ser disparados de quase todos os métodos, de modo que seria incômodo para os programadores serem obrigados a listá-las em todas as definições de métodos. Tais exceções e erros não precisam ser listadas na cláusula `throw` de um método e, portanto, dizemos que elas são “não-verificadas” pelo compilador. Todas as exceções que não sejam `RuntimeExceptions` que um método pode disparar devem ser listadas na cláusula `throws` desse método e, portanto, dizemos que elas são “verificadas” pelo compilador. Se uma exceção que não é `RuntimeException` não é listada na cláusula `throws`, o compilador emite uma mensagem de erro indicando que a exceção deve ser capturada (com o `try/catch` no corpo do método) ou declarada (com uma cláusula `throws`).



Erro comum de programação 14.9

Haverá um erro de sintaxe se um método disparar uma exceção verificada que não estiver na cláusula `throws` desse método.



Erro comum de programação 14.10

Tentar disparar uma exceção verificada a partir de um método que não tem uma cláusula `throws` é um erro de sintaxe.



Observação de engenharia de software 14.12

Se seu método chamar outros métodos que explicitamente disparam exceções verificadas, essas exceções devem ser listadas na cláusula `throws` de seu método, a menos que ele capture essas exceções. Esse é o requisito “capture ou declare” (“catch-or-declare”) de Java.



Erro comum de programação 14.11

Se um método de subclasse sobreescrivar um método de superclasse, é um erro o método de subclasse listar mais exceções em sua lista da cláusula `throws` do que lista o método sobreescrito da superclasse. A lista da cláusula `throws` da subclasse pode conter um subconjunto da lista da cláusula `throws` da superclasse.

O requisito `catch-or-declare` de Java exige que o programador capture cada exceção verificada ou a coloque na cláusula `throws` de um método. Naturalmente, colocar uma exceção verificada na cláusula `throws` forçaria outros métodos a também processar a exceção verificada. Se o programador acredita que é improvável que ocorra uma determinada exceção verificada, ele pode optar por capturar essa exceção verificada e não fazer nada com ela para não ser forçado a lidar com ela mais tarde. Naturalmente, isso pode voltar a “assombrá-lo”, já que, à medida que o programa se desenvolve, pode tornar-se importante lidar com essa exceção verificável.



Dica de teste e depuração 14.6

Não tente contornar o requisito `catch-or-declare` de Java simplesmente capturando exceções e nada fazendo com elas. Geralmente, as exceções são de uma natureza suficientemente séria e é necessário tratá-las e não as surpreender.



Dica de teste e depuração 14.7

O compilador Java, através da cláusula `throws` utilizada com o tratamento de exceções, força os programadores a processar as exceções que podem ser disparadas de cada método que um programa chama. Isso ajuda a evitar falhas que surgem nos programas quando os programadores ignoram o fato de que os problemas acontecem e não se preparam para enfrentá-los.



Observação de engenharia de software 14.13

Os métodos de subclasse que não sobrescrevem seus métodos correspondentes de superclasse exibem o mesmo comportamento de tratamento de exceções dos métodos herdados da superclasse. A cláusula `throws` de um método de subclasse que anula um método de superclasse pode conter a mesma lista de exceções que o método de superclasse sobreescrito ou um subconjunto dessa lista.



Erro comum de programação 14.12

O compilador Java exige que um método capture quaisquer exceções verificadas e disparadas no método (seja diretamente a partir do bloco do método, seja indiretamente através de métodos chamados) ou declare `Exceptions` verificadas que o método pode disparar para outros métodos; caso contrário, o compilador Java emite um erro de sintaxe.



Dica de teste e depuração 14.8

Suponha que um método dispara todas as subclasses de uma superclasse de exceção particular. Você pode ficar tentado a listar apenas a superclasse na cláusula `throws`. Em vez disso, liste explicitamente todas as subclasses. Isso focaliza a atenção do programador nas `Exceptions` específicas que podem ocorrer e freqüentemente ajudará a evitar falhas causadas pela execução de processamento genérico para uma categoria de tipos de exceção.

As Figs. 14.3 a 14.8 listam muitos `Errors` e `Exceptions` de Java hierarquicamente para os pacotes `java.lang`, `java.util`, `java.io`, `java.awt` e `java.net`. Nestas tabelas, a classe recuada abaixo de a outra classe é uma subclasse. As classes de exceção e de erro para os outros pacotes da API de Java podem ser encontradas na documentação *on-line* de Java. A documentação *on-line* para cada método na API especifica se esse método dispara exceções e quais são as exceções que podem ser disparadas. Mostramos uma parte da hierarquia `Error` de Java na Fig. 14.3. A maioria dos programadores de Java simplesmente ignora `Errors`. Trata-se de eventos sérios mas raros.

A Fig. 14.4 é particularmente importante porque lista muitas das `RuntimeExceptions` de Java. Ainda que os programadores de Java não sejam obrigados a declarar essas exceções em cláusulas `throws`, essas exceções são as que comumente serão capturadas e tratadas em aplicativos Java.

Erros do pacote `java.lang`

`Error` (todos no pacote `java.lang`, exceto `AWTError`, que está no pacote `java.awt`)

```

LinkageError
ClassCircularityError
ClassFormatError
ExceptionInInitializerError
IncompatibleClassChangeError
AbstractMethodError
IllegalAccessError

```

Fig. 14.3 Erros do pacote `java.lang` (parte 1 de 2).

Erros do pacote java.lang

```
InstantiationException
NoSuchFieldError
NoSuchMethodError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
ThreadDeath
VirtualMachineError (Abstract class)
InternalError
OutOfMemoryError
StackOverflowError
UnknownError
AWTError (em java.awt)
```

Fig. 14.3 Erros do pacote java.lang (parte 2 de 2).

Exceções do pacote java.lang

```
Exception
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchFieldException
NoSuchMethodException
RuntimeException
ArithmaticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalThreadStateException
NumberFormatException
IllegalMonitorStateException
IllegalStateException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
```

Fig. 14.4 Exceções do pacote java.lang.

A Fig. 14.5 lista os outros três tipos de dados **RuntimeExceptions** de Java. Encontraremos essas exceções no Capítulo 20 quando estudarmos a classe **Vector**. O **Vector** é um *array* dinâmico que pode crescer e encolher para acomodar os requisitos de armazenamento variáveis de um programa.

A Fig. 14.6 lista as **IOExceptions** de Java. Todas essas são exceções verificadas que podem ocorrer durante a entrada/saída e o processamento de arquivos.

A Fig. 14.7 lista a única **Exception** verificada do pacote **java.awt**, a **AWTException**. Trata-se de uma exceção verificada que é disparada por vários métodos do Abstract Windowing Toolkit.

Exceções do pacote **java.util**

```
Exception
  RuntimeException
    EmptyStackException
    MissingResourceException
    NoSuchElementException
    TooManyListenersException
```

Fig. 14.5 Exceções do pacote **java.util**.

Exceções do pacote **java.io**

```
Exception
  IOException
    CharConversionException
    EOFException
    FileNotFoundException
    InterruptedIOException
    ObjectStreamException
      InvalidClassException
      InvalidObjectException
      NotActiveException
      NotSerializableException
      OptionalDataException
      StreamCorruptedException
      WriteAbortedException
    SyncFailedException
    UnsupportedCodingException
    UTFDataFormatException
```

Fig. 14.6 Exceções do pacote **java.io**.

Exceções do pacote `java.awt`

```

Exception
  AWTException
  RuntimeException
    IllegalStateException
    IllegalComponentStateException

```

Fig. 14.7 Exceções do pacote `java.awt`.

A Fig. 14.8 lista as `IOExceptions` do pacote `java.net`. Todas são `Exceptions` verificadas que indicam vários problemas de rede.

A maioria dos pacotes na Java API define `Exceptions` e `Errors` específicos para o pacote. Para obter uma relação completa destes tipos, consulte a documentação *on-line* da API para o pacote.

Exceções do pacote `java.net`

```

Exception
  IOException
    BindException
    MalformedURLException
    ProtocolException
    SocketException
      ConnectException
      NoRouteToHostException
    UnknownHostException
    UnknownServiceException

```

Fig. 14.8 Exceções do pacote `java.net`.

14.11 Construtores, finalizadores e tratamento de exceções

Primeiro, vamos tratar de uma questão que só mencionamos, mas que ainda precisa ser resolvida satisfatoriamente. O que acontece quando se detecta um erro em um construtor? O problema é que o construtor não pode retornar um valor, então como fazemos para que o programa saiba que um objeto não foi construído adequadamente? Uma das soluções é simplesmente retornar o objeto construído inadequadamente e esperar que qualquer pessoa utilizando o objeto faça os testes apropriados para determinar que o objeto é de fato inadequado. Entretanto, isto contradiz diretamente os comentários do Capítulo 8, nos quais indicamos que você deve manter um objeto em um estado consistente durante todo o tempo. Outra possibilidade é configurar alguma variável de instância com um indicador de erro fora do construtor, mas essa é uma prática de programação pobre. Em Java, o mecanismo típico (e apropriado) é disparar uma exceção a partir do construtor para o código que está criando o objeto. O objeto disparado contém informações sobre a chamada ao construtor que falhou e o chamador é responsável por tratar da falha.

Quando ocorre uma exceção em um construtor, outros objetos criados por aquele construtor são marcados para a coleta de lixo em algum momento. Antes de cada objeto ser jogado no lixo, o método `finalize` será chamado.

14.12 Exceções e herança

Várias classes de exceção podem se derivar de uma superclasse comum. Caso se escreva um `catch` para capturar objetos de exceção de um tipo de superclasse, ele também pode capturar todos os objetos de subclasses dessa superclasse. Isso pode permitir o processamento polimórfico de exceções relacionadas.

Utilizar herança com exceções permite que um tratador de exceções capture erros relacionados com uma notação concisa. Certamente pode-se capturar individualmente o objeto de exceção de cada subclasse se estas exceções necessitarem de processamento diferente, mas é mais conciso capturar o objeto de exceção da superclasse. Naturalmente, isso só faria sentido se o comportamento do tratamento fosse o mesmo para todas as subclasses. Caso contrário, capture cada exceção de subclasse individualmente.



Dica de teste e depuração 14.9

A captura individual de objetos de exceção de subclasse está sujeita a erros se o programador esquecer de testar explicitamente um ou mais dos tipos de subclasse; capturar a superclasse garante que os objetos de todas as subclasses sejam capturados. Frequentemente, a captura do tipo da superclasse segue todos os outros tratadores de exceção de subclasses para assegurar que todas as exceções sejam processadas apropriadamente.

14.13 O bloco `finally`

Os programas que obtêm certos tipos de recursos devem devolver explicitamente esses recursos para o sistema para evitar a *perda de recursos*. Em linguagens de programação como C e C++, o tipo mais comum de perda de recurso é a perda de memória. Java realiza coleta automática de lixo de memória não mais necessária para os programas, evitando, assim, a maioria das perdas de memória. Mas outros tipos de perdas de recursos podem ocorrer em Java.



Observação de engenharia de software 14.14

O bloco `finally` em geral contém código para liberar recursos adquiridos em seu bloco `try` correspondente; trata-se de uma maneira eficiente de eliminar as perdas de recursos. Por exemplo, o bloco `finally` deve fechar quaisquer arquivos abertos no bloco `try`.



Dica de teste e depuração 14.10

Na realidade, Java não elimina completamente as perdas de memória. Há uma questão sutil aqui. Java não coleta um objeto para o lixo até não haver mais referências ao objeto. Portanto, podem ocorrer perdas de memória, mas apenas se os programadores mantiverem erroneamente referências para os objetos não-desejados. A maioria dos problemas de perda de memória é resolvida por meio da coleta de lixo de Java.

O bloco `finally` é opcional. Se estiver presente, ele é colocado depois do último bloco `catch` de um bloco `try`, como segue:

```
try {
    instruções;
    instruções de aquisição de recursos
}
catch ( UmTipoDeExceção exceção1 ) {
    instruções de tratamento de exceção
}
catch ( UmOutroTipoDeExceção exceção2 ) {
    instruções de tratamento de exceção
}
finally {
    instruções
    instruções de liberação de recursos
}
```

Java garante que o bloco `finally` (se houver um) será executado independentemente de qualquer exceção ser ou não disparada no bloco `try` correspondente ou quaisquer de seus blocos `catch` correspondentes. Java também garante que o bloco `finally` (se houver um) será executado se um bloco `try` for encerrado com uma instrução `return`, `break` ou `continue`.

O código de liberação de recursos é colocado em um bloco **finally**. Suponha que o recurso seja alocado em um bloco **try**. Se nenhuma exceção ocorrer, os tratadores **catch** são pulados e o controle prossegue para o bloco **finally**, que libera o recurso. Depois o controle prossegue para a primeira instrução após o bloco **finally**.

Se ocorrer uma exceção, o programa pula o resto do bloco **try**. Se o programa capturar a exceção em um dos tratadores **catch**, o programa processa a exceção. Assim, o bloco **finally** libera o recurso e depois o controle prossegue para a primeira instrução depois do bloco **finally**.

Se uma exceção que ocorrer no bloco **try** não puder ser capturada por um dos tratadores **catch**, o programa pula o resto do bloco **try** e o controle prossegue para o bloco **finally**, que libera o recurso. Assim, o programa passa a exceção para cima na cadeia de chamadas até algum método chamador decidir capturá-la. Se nenhum método decidir tratá-la, o aplicativo não-baseado em GUI é terminado.

Se um tratador **catch** dispara uma exceção, o bloco **finally** ainda é executado. Então, a exceção é passada para cima na cadeia de chamadas para um método chamador a capture e a trate.

O aplicativo Java da Fig. 14.9 demonstra que o bloco **finally** (se houver um) é executado mesmo se uma exceção não for disparada no bloco **try** correspondente. O programa contém os métodos **main** (linhas 7 a 21), **throwException** (linhas 24 a 50) e **doesNotThrowException** (linhas 53 a 75). Os métodos **throwException** e **doesNotThrowException** são declarados **static** para que **main** (outro método **static**) possa chamá-los diretamente.

O método **main** começa a ser executado, entra em seu bloco **try** e imediatamente chama **throwException** (linha 11). O método **throwException** dispara uma **Exception** (linha 29), captura a mesma (linha 33) e a dispara novamente (linha 37). A exceção disparada novamente será tratada em **main**, mas primeiro o bloco **finally** (linhas 44 a 47) é executado. O método **main** detecta a exceção disparada novamente no bloco **try** em **main** (linhas 10 a 12) e a trata com o bloco **catch** (linhas 15 a 18). Em seguida, o método **doesNotThrowException** (linha 20) é chamado. Nenhuma exceção é disparada no bloco **try** de **doesNotThrowException**, de modo que o programa pula o bloco **catch** (linhas 61 a 64), mas o bloco **finally** (linhas 68 a 71) é executado mesmo assim. O controle prossegue para depois do bloco **finally**. Então, o controle retorna para **main** e o programa é terminado.

```

1 // Fig. 14.9: UsingExceptions.java
2 // Demonstração do mecanismo de tratamento
3 // de exceções try-catch-finally.
4 public class UsingExceptions {
5
6     // executa o aplicativo
7     public static void main( String args[] )
8     {
9         // chama o método throwException
10        try {
11            throwException();
12        }
13
14        // captura Exceptions disparadas pelo método throwException
15        catch ( Exception exception )
16        {
17            System.err.println( "Exception handled in main" );
18        }
19
20        doesNotThrowException();
21    }
22
23    // demonstra try/catch/finally
24    public static void throwException() throws Exception
25    {
26        // dispara uma exceção e a captura imediatamente
27        try {
28            System.out.println( "Method throwException" );

```

Fig. 14.9 Demonstração do mecanismo de tratamento de exceção **try-catch-finally** (parte 1 de 2).

```

29         throw new Exception();    // gera exceção
30     }
31
32     // captura exceção disparada no bloco try
33     catch ( Exception exception )
34     {
35         System.err.println(
36             "Exception handled in method throwException" );
37         throw exception;    // dispara novamente para processamento adicional
38
39         // qualquer código aqui não seria alcançado
40     }
41
42     // este bloco é executado independentemente
43     // do que ocorrer no bloco try/catch
44     finally {
45         System.err.println(
46             "Finally executed in throwException" );
47     }
48
49     // qualquer código aqui não seria alcançado
50 }
51
52 // demonstra finally quando não ocorrer nenhuma exceção
53 public static void doesNotThrowException()
54 {
55     // o bloco try não dispara uma exceção
56     try {
57         System.out.println( "Method doesNotThrowException" );
58     }
59
60     // catch não é executado, porque nenhuma exceção foi disparada
61     catch( Exception exception )
62     {
63         System.err.println( exception.toString() );
64     }
65
66     // este bloco é executado independentemente
67     // do que ocorrer no bloco try/catch
68     finally {
69         System.err.println(
70             "Finally executed in doesNotThrowException" );
71     }
72
73     System.out.println(
74         "End of method doesNotThrowException" );
75 }
76
77 } // fim da classe UsingExceptions

```

```

Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException

```

Fig. 14.9 Demonstração do mecanismo de tratamento de exceção `try-catch-finally` (parte 2 de 2).

O aplicativo Java na Fig. 14.10 demonstra que, quando uma exceção disparada em um bloco **try** não é capturada no bloco **catch** correspondente, a exceção será detectada no próximo bloco externo **try** e tratada por um bloco **catch** apropriado (se houver um) associado com aquele bloco externo **try**.

```

1 // Fig. 14.10: UsingExceptions.java
2 // Demonstração de desempilhamento.
3 public class UsingExceptions {
4
5     // executa o aplicativo
6     public static void main( String args[] )
7     {
8         // chama throwException para demonstrar desempilhamento
9         try {
10             throwException();
11         }
12
13         // captura exceção disparada em throwException
14         catch ( Exception exception ) {
15             System.err.println( "Exception handled in main" );
16         }
17     }
18
19     // throwException dispara uma exceção que
20     // não é capturada no corpo deste método
21     public static void throwException() throws Exception
22     {
23         // dispara uma exceção e a captura em main
24         try {
25             System.out.println( "Method throwException" );
26             throw new Exception(); // gera exceção
27         }
28
29         // catch é do tipo incorreto, de modo que Exception não é capturada
30         catch( RuntimeException runtimeException ) {
31             System.err.println(
32                 "Exception handled in method throwException" );
33         }
34
35         // bloco finally sempre é executado
36         finally {
37             System.err.println( "Finally is always executed" );
38         }
39     }
40 }
41 } // fim da classe UsingExceptions

```

```

Method throwException
Finally is always executed
Exception handled in main

```

Fig. 14.10 Demonstração de um desempilhamento.

Quando o método **main** é executado, a linha 10 no bloco **try** chama **throwException** (linhas 21 a 39). No bloco **try** do método **throwException**, a linha 26 dispara uma **Exception**. Isso termina o bloco **try** imediatamente e o controle prossegue para o tratador **catch** na linha 30. O tipo que está sendo capturado (**RuntimeException**) não corresponde exatamente ao tipo disparado (isto é, **Exception**) e não é uma superclasse do tipo disparado, de modo que a exceção não é capturada no método **throwException**. A exceção deve ser tratada antes que a execução normal do programa possa continuar. Portanto, o método **throwException** termina

(mas não antes de seu bloco **finally** ser executado) e retorna o controle para o ponto do qual ele foi chamado no programa (linha 10). A linha 10 está no bloco externo **try**. Se a exceção ainda não foi tratada, o bloco **try** é terminado e é feita uma tentativa de capturar a exceção na linha 14. O tipo que está sendo capturado (**Exception**) corresponde ao tipo disparado. Portanto, o tratador **catch** processa a exceção e o programa termina no fim de **main**.

Como vimos, um bloco **finally** é executado por uma variedade de razões, como o término bem-sucedido de um **try**, o tratamento de uma exceção em um **catch** local, o disparo de uma exceção para a qual nenhum **catch** local está disponível, ou a execução de uma instrução de controle de programa como um **return**, **break** ou **continue**. Normalmente, o bloco **finally** é executado, então se comporta adequadamente (chamaremos isso de “ação de continuação” desse **finally**), de acordo com a razão por que se entrou nele. Por exemplo, se uma exceção é disparada no bloco **finally**, a ação de continuação será que essa exceção seja processada no próximo bloco externo **try**. Infelizmente, se houvesse uma exceção que ainda não tivesse sido capturada, ela seria perdida e a exceção mais recente seria processada. Isso é perigoso.

Erro comum de programação 14.13



*Se for disparada uma exceção para a qual nenhum **catch** local está disponível, quando o controle entrar no bloco **finally** local, o bloco **finally** também poderá disparar uma exceção. Se isso acontecer, a primeira exceção será perdida.*

Dica de teste e depuração 14.11



*Evite colocar código que possa disparar uma exceção em um bloco **finally**. Se for necessário código como esse, coloque o código dentro de um **try/catch** dentro do bloco **finally**.*

Boa prática de programação 14.7



*O mecanismo de tratamento de exceções de Java foi projetado para remover o código de processamento de erros do fluxo principal do código de um programa para melhorar a sua clareza. Não coloque **try/catch/finally** em torno de cada instrução que possa disparar uma exceção. Isso torna os programas difíceis de ler. Em vez disso, coloque um bloco **try** em torno de uma parte significativa de seu código, coloque após esse bloco **try** os blocos **catch** que tratam cada exceção possível, e depois siga os blocos **catch** com um único bloco **finally** (se for necessário um).*

Dica de desempenho 14.3



Via de regra, os recursos devem ser liberados logo que fique claro que eles não são mais necessários. Isso torna esses recursos imediatamente disponíveis para reutilização e pode melhorar o desempenho do programa.

Observação de engenharia de software 14.15



*Se um bloco **try** tem um bloco **finally** correspondente, o bloco **finally** será executado mesmo que o bloco **try** encerre com **return**, **break** ou **continue**; então, o efeito de **return**, **break** ou **continue** ocorrerá.*

14.14 Utilizando `printStackTrace` e `getMessage`

As exceções derivam-se da classe `Throwable`. A classe `Throwable` oferece um método `printStackTrace` que imprime a pilha de chamadas de métodos. Chamando esse método para um objeto `Throwable` que foi capturado, o programa pode imprimir a pilha de chamadas de métodos. Isso é frequentemente útil no processo de teste e depuração. Duas outras versões sobreescarregadas de `printStackTrace` permitem que o programa direcione a monitorização da pilha para um fluxo `PrintStream` ou `PrintWriter`. Nesta seção, analisamos um exemplo que exercita o método `printStackTrace` e um outro método útil, `getMessage`.

Dica de teste e depuração 14.12



Todos os objetos `Throwable` contêm um método `printStackTrace` que imprime um monitoramento da pilha para o objeto.

Dica de teste e depuração 14.13



Uma exceção que não é capturada acaba, em algum momento, fazendo com que o tratador de exceções default de Java seja executado. Isso exibe o nome da exceção, o string de caracteres opcional que foi fornecido quando a exceção foi construída e um monitoramento completo da pilha de execução. O monitoramento da pilha mostra a pi-

lha completa de chamadas de métodos. Isso permite que o programador veja o caminho de execução que conduziu à exceção arquivo por arquivo (e, portanto, classe por classe) e método por método. Essas informações são úteis na depuração de um programa.

Há dois construtores para a classe **Throwable**. O primeiro construtor

```
public Throwable()
```

não recebe argumentos. O segundo construtor

```
public Throwable( String informationString )
```

recebe o argumento **informationString** que contém informações descritivas sobre o objeto **Throwable**. O **informationString** armazenado no objeto **Throwable** pode ser obtido com o método **getMessage**.



Dica de teste e depuração 14.14

*As classes Throwable têm um construtor que aceita um argumento **String**. Utilizar essa forma do construtor é útil na determinação da origem da exceção através do método **getMessage** ().*

A Fig. 14.11 demonstra **getMessage** e **printStackTrace**. O método **getMessage** devolve o **String** descritivo armazenado em uma exceção. O método **printStackTrace** envia para o fluxo de saída de erro padrão (normalmente, a linha de comando ou console) uma mensagem de erro com o nome da classe de exceção, o **String** descritivo armazenado na exceção e uma lista dos métodos que não tinham terminado de ser executados quando a exceção foi disparada (isto é, todos os métodos atualmente residindo na pilha de chamadas de métodos).

No programa, **main** invoca **method1**, **method1** invoca **method2** e **method2** invoca **method3**. Nesse ponto, a pilha de chamadas de métodos para o programa é

```
method3
method2
method1
main
```

com o último método chamado (**method3**) no topo e o primeiro método chamado (**main**) na base. Quando **method3** dispara uma **Exception** (linha 36), uma mensagem de monitoramento de pilha é gerada e armazenada no objeto **Exception**. O monitoramento da pilha reflete o ponto de disparo no código (isto é, a linha 36). Então, a pilha é desempilhada até o primeiro método na pilha de chamadas de métodos em que a exceção pode ser capturada (isto é, **main**, porque este contém um tratador **catch** para **Exception**). O tratador **catch** então utiliza **getMessage** e **printStackTrace** sobre o objeto **Exception exception** para produzir a saída. Observe que os números de linhas na janela de saída correspondem aos números de linha no programa.

```

1 // Fig. 14.11: UsingExceptions.java
2 // Demonstra os métodos getMessage e printStackTrace
3 // herdados por todas as classes de exceção.
4 public class UsingExceptions {
5
6     // executa o aplicativo
7     public static void main( String args[] )
8     {
9         // chama method1
10        try {
11            method1();
12        }
13
14        // captura Exceptions disparadas de method1
15        catch ( Exception exception ) {
16            System.err.println( exception.getMessage() + "\n" );
17            exception.printStackTrace();
18        }
19    }
}
```

Fig. 14.11 Usando **getMessage** e **printStackTrace** (parte 1 de 2).

```

20      // chama method2; dispara exceções de volta para main
21      public static void method1() throws Exception
22      {
23          method2();
24      }
25
26
27      // chama method3; dispara exceções de volta para method1
28      public static void method2() throws Exception
29      {
30          method3();
31      }
32
33      // dispara Exception de volta para method2
34      public static void method3() throws Exception
35      {
36          throw new Exception( "Exception thrown in method3" );
37      }
38
39  } // fim da classe UsingExceptions

```

```

Exception thrown in method3

java.lang.Exception: Exception thrown in method3
    at UsingExceptions.method3 (UsingExceptions.java:36)
    at UsingExceptions.method2 (UsingExceptions.java:30)
    at UsingExceptions.method1 (UsingExceptions.java:24)
    at UsingExceptions.main (UsingExceptions.java:11)

```

Fig. 14.11 Usando `getMessage` e `printStackTrace` (parte 2 de 2).

Resumo

- Alguns exemplos comuns de exceções são esgotamento de memória, um subscripto de *array* fora dos limites, estouro aritmético, divisão por zero e parâmetros de método inválidos.
- O tratamento de exceções foi projetado para lidar com mau funcionamentos síncronos (isto é, aqueles que ocorrem como resultado da execução de um programa).
- O tratamento de exceções é utilizado em situações em que o mau funcionamento será tratado em um escopo diferente daquele que detectou o mau funcionamento.
- Deve-se usar o tratamento de exceções para processar exceções de componentes de *software* como métodos, bibliotecas e classes que provavelmente serão muito utilizados e nos quais não faz sentido para esses componentes tratar suas próprias exceções.
- Deve-se usar o tratamento de exceções em grandes projetos para tratar processamento de erros de uma maneira padronizada para o projeto inteiro.
- O tratamento de exceções de Java volta-se para situações em que o método que detecta um erro é incapaz de lidar com ele. Esse método irá disparar uma exceção. Se a exceção corresponder ao tipo do parâmetro em um dos blocos `catch`, o código para esse bloco `catch` é executado.
- O programador inclui em um bloco `try` o código que pode gerar um erro que produzirá uma exceção. O bloco `try` é imediatamente seguido por um ou mais blocos `catch`. Cada bloco `catch` especifica o tipo de exceção que ele pode capturar e tratar. Cada bloco `catch` é um tratador de exceções.
- Se nenhuma exceção for disparada no bloco `try`, os tratadores de exceções para esse bloco são pulados. Então, o programa retoma a execução depois do último bloco `catch`, depois de executar um bloco `finally`, se algum for fornecido.
- As exceções são disparadas em um bloco `try` em um método ou de um método chamado direta ou indiretamente a partir do bloco `try`.
- O operando de um `throw` pode ser de qualquer classe derivada de `Throwable`. As subclasses imediatas de `Throwable` são `Error` e `Exception`.

- Dizemos que `RuntimeExceptions` e `Errors` são “não-verificados”. Dizemos que as exceções `RuntimeExceptions` são “verificadas”. As exceções verificadas disparadas por um método particular devem ser especificadas na cláusula `throws` desse método.
- As exceções são capturadas pelo tratador de exceções mais próximo (para o bloco `try` a partir do qual a exceção foi disparada) especificando-se um tipo adequado.
- A exceção termina o bloco em que ocorreu.
- O tratador pode disparar novamente o objeto para um bloco `try` externo.
- `catch(Exception exception)` captura todas as `Exceptions`.
- `catch(Throwable throwable)` captura todas as `Exceptions` e `Errors`.
- Se nenhum tratador corresponder a um objeto disparado particular, a procura por uma correspondência continua em um bloco `try` mais externo.
- Os tratadores de exceção são pesquisados em ordem para uma correspondência apropriada baseada no tipo. O primeiro tratador correspondente é executado. Quando esse tratador termina de ser executado, o controle é retornado na primeira instrução depois do último bloco `catch`.
- A ordem dos tratadores de exceções afeta a maneira como se trata uma exceção.
- O objeto de subclasse pode ser capturado por um tratador que especifica o tipo dessa subclasse, ou por tratadores que especificam os tipos de quaisquer superclasses diretas ou indiretas dessa subclasse.
- Se nenhum tratador for localizado para uma exceção, o aplicativo não-baseado em GUI é terminado; o *applet* ou o aplicativo baseado em GUI retornará para seu tratamento de evento regular.
- O tratador de exceção não pode acessar variáveis no escopo de seu bloco `try` porque, no momento em que o tratador de exceção começa a ser executado, o bloco `try` expirou. As informações de que o tratador precisa normalmente são passadas no objeto disparado.
- Os tratadores de exceção podem disparar novamente uma exceção. Podem converter um tipo de exceção em outro disparando uma exceção diferente. Podem realizar qualquer recuperação necessária e retomar a execução depois do último tratador de exceção. Podem analisar a situação que causa o erro, remover a causa do erro e tentar novamente chamando o método que causou a exceção originalmente. Podem simplesmente retornar algum valor de estado para seu ambiente.
- O tratador que captura um objeto de subclasse deve ser colocado antes de um tratador que captura um objeto de superclasse. Se o tratador de superclasse for o primeiro, ele irá capturar os objetos de superclasse e os objetos de subclasses dessa superclasse.
- Quando se captura uma exceção, é possível que recursos tenham sido alocados, mas ainda não tenham sido liberados no bloco `try`. O bloco `finally` deve liberar esses recursos.
- É possível que o tratador que captura uma exceção decida que não é capaz de processar a exceção. Nesse caso, o tratador pode simplesmente disparar novamente a exceção. O `throw` seguido pelo nome do objeto de exceção dispara novamente a exceção.
- Mesmo se um tratador puder processar uma exceção e independentemente de ele fazer qualquer processamento dessa exceção, o tratador pode disparar novamente a exceção para processamento adicional fora do tratador. A exceção disparada novamente é detectada pelo próximo bloco `try` que a envolve (normalmente em um método chamador) e é tratado por um tratador de exceções apropriado (se houver algum) listado depois desse bloco `try`.
- A cláusula `throws` lista as exceções verificadas que podem ser disparadas de um método. O método pode disparar (`throw`) as exceções indicadas ou pode disparar tipos de subclasse. Se uma exceção verificada não-listada na cláusula `throws` é disparada, ocorre um erro de sintaxe.
- Uma forte razão para utilizar herança com exceções é capturar uma variedade de erros relacionados facilmente com notação concisa. Certamente pode-se capturar individualmente cada tipo de objeto de exceção de subclasse, mas é mais conciso simplesmente capturar o objeto de exceção da superclasse.

Terminologia

<code>ArithmetricException</code>	<i>classe Error</i>
<code>ArrayIndexOutOfBoundsException</code>	<i>classe Exception</i>
<code>bloco catch</code>	<i>classe Throwable</i>
<code>bloco finally</code>	<i>classe de exceção de biblioteca</i>
<code>bloco try</code>	<i>computação de missão crítica</i>
<code>capturar todas as exceções</code>	<i>computação de negócios críticos</i>
<code>capturar um grupo de exceções</code>	<i>declarar exceções que podem ser disparadas</i>
<code>capturar uma exceção</code>	<i>desempilhamento</i>
<code>ClassCastException</code>	<i>disparar novamente uma exceção</i>

<i>disparar uma exceção</i>	<i>modelo de retomada do tratamento de exceções</i>
EmptyStackException	<i>modelo de terminação do tratamento de exceções</i>
<i>erro síncrono</i>	NegativeArraySizeException
<i>esgotamento de memória</i>	NoClassDefFoundException
<i>exceção</i>	NullPointerException
<i>exceção não durante a execução</i>	<i>objeto de exceção</i>
<i>exceções de array</i>	operator instanceof
Exceptions não-verificadas	OutOfMemoryError
Exceptions verificadas	<i>perda de recurso</i>
FileNotFoundException	<i>ponto de disparo</i>
<i>hierarquia da classe Error</i>	<i>referência null</i>
<i>hierarquia da classe Exception</i>	<i>requisito catch-or-declare</i>
IllegalAccessException	RuntimeException
IncompatibleClassChangeException	<i>throws</i>
InstantiationException	<i>tolerância a falhas</i>
<i>instrução throw</i>	<i>tratador de exceções</i>
InternalException	<i>tratador de exceções default</i>
InterruptedException	<i>tratamento de erro</i>
IOException	<i>tratamento de exceções</i>
<i>método getMessage da classe Throwable</i>	<i>tratar uma exceção</i>
<i>método printStackTrace (Throwable)</i>	UnsatisfiedLinkException

Exercícios de auto-revisão

- 14.1** Liste cinco exemplos de exceções comuns.
- 14.2** Por que as técnicas de tratamento de exceções não devem ser utilizadas para controle convencional de programa?
- 14.3** Por que as exceções são particularmente adequadas para lidar com erros produzidos por classes de biblioteca e métodos?
- 14.4** O que é uma “perda de recurso”?
- 14.5** Se nenhuma exceção for disparada em um bloco **try**, para onde segue o controle quando o bloco **try** completa a execução?
- 14.6** O que acontece se ocorrer uma exceção e um tratador de exceções apropriado não puder ser localizado?
- 14.7** Dê uma vantagem fundamental de utilizar **catch (Exception)**.
- 14.8** Um *applet* convencional ou aplicativo deve capturar objetos **Error**?
- 14.9** O que acontece se vários tratadores correspondem ao tipo do objeto disparado?
- 14.10** Por que um programador especificaria um tipo de superclasse como o tipo de um tratador **catch** e depois dispara objetos de tipos de subclasse?
- 14.11** Como um tratador **catch** pode ser escrito para processar tipos de erros relacionados sem utilizar herança entre classes de exceção?
- 14.12** Qual é a razão fundamental para utilizar blocos **finally**?
- 14.13** Disparar uma **Exception** causa necessariamente a terminação do programa?
- 14.14** O que acontece quando um tratador **catch** dispara uma **Exception**?
- 14.15** O que acontece com uma referência local em um bloco **try** quando esse bloco dispara uma **Exception**?

Respostas aos exercícios de auto-revisão

- 14.1** Esgotamento de memória, subscripto de *array* fora dos limites, estouro aritmético, divisão por zero, parâmetros inválidos de método.
- 14.2** (a) O tratamento **Exception** foi projetado para tratar situações que ocorrem raramente, mas que freqüentemente resultam na terminação do programa; portanto, os escritores de compiladores não são obrigados a implementar tra-

tamento de exceções de forma a otimizar o desempenho. (b) O fluxo de controle com estruturas de controle convencionais é geralmente mais claro e mais eficiente do que com exceções. (c) Podem ocorrer problemas porque a pilha é desempilhada quando ocorre uma exceção e os recursos alocados antes da exceção não podem ser liberados. (d) As exceções “adicionais” podem entrar no caminho de exceções verdadeiras do tipo erro. Torna-se mais difícil para o programador monitorar o maior número de casos de exceção.

14.3 É improvável que classes de biblioteca e métodos possam realizar processamento de erros que atenda às necessidades particulares de todos os usuários.

14.4 Ocorre uma perda de recursos quando um programa em execução não libera apropriadamente um recurso quando o recurso não mais é necessário. Se o programa tentar utilizar o recurso novamente no futuro, ele pode não ser capaz de acessar o recurso.

14.5 Os tratadores de exceções (nos blocos `catch`) para esse bloco `try` são pulados e o programa retoma a execução depois do último bloco `catch`. Se houver um bloco `finally`, ele será executado e o programa retomará a execução depois do bloco `finally`.

14.6 O aplicativo não-baseado em GUI é terminado; um *applet* ou um aplicativo baseado em GUI retoma o processamento de eventos regular.

14.7 A forma `catch (Exception exception)` captura qualquer tipo de exceção disparada em um bloco `try`. Uma vantagem é que nenhuma `Exception` disparada pode deixar de ser capturada.

14.8 `Errors` são problemas normalmente sérios com o sistema Java subjacente; a maioria dos programas não irá querer capturar `Errors`.

14.9 O primeiro tratador `Exception` correspondente depois do bloco `try` é executado.

14.10 Essa é uma boa maneira de capturar tipos de exceções relacionadas, mas deve ser utilizada com cuidado.

14.11 Forneça uma única subclasse `Exception` e um tratador `catch` para um grupo de exceções. Quando ocorrer cada exceção, o objeto de exceção pode ser criado com dados de instâncias diferentes. O tratador `catch` pode examinar esses dados para distinguir o tipo de `Exception`.

14.12 O bloco `finally` é a maneira preferida de evitar perdas de recursos.

14.13 Não, mas termina o bloco em que é disparada a `Exception`.

14.14 A exceção será processada por um tratador `catch` (se houver um) associado ao bloco `try` (se houver um) que inclui o tratador `catch` que causou a exceção.

14.15 A referência é removida da memória e a contagem de referências ao objeto mencionado é decrementada. Se a contagem de referências for zero, o objeto é marcado para coleta de lixo.

Exercícios

14.16 Sob quais circunstâncias você utilizaria a seguinte instrução?

```
catch ( Exception exception ) {
    throw exception;
}
```

14.17 Liste os benefícios do tratamento de exceções sobre os meios convencionais de processamento de erro.

14.18 Descreva uma técnica orientada a objetos para tratamento de exceções relacionadas.

14.19 Até este capítulo, descobrimos que lidar com erros detectados por construtores é um pouco estranho. Explique por que o tratamento de exceções é um meio eficaz de lidar com falhas de construtores.

14.20 Suponha que um programa dispare uma exceção e o tratador de exceções apropriado comece a ser executado. Agora suponha que o próprio tratador de exceção dispare a mesma exceção. Isso cria uma recursão infinita? Explique sua resposta.

14.21 Utilize herança para criar uma superclasse de exceção e várias subclasses de exceção. Escreva um programa para demonstrar que o `catch` que especifica a superclasse captura exceções de subclasse.

14.22 Escreva um programa Java que mostra que nem todos os finalizadores para os objetos construídos em um bloco são necessariamente chamados depois que uma exceção for disparada a partir desse bloco.

14.23 Escreva um programa Java que demonstra como várias exceções são capturadas com

```
catch ( Exception exception )
```

14.24 Escreva um programa Java que mostra que a ordem dos tratadores de exceções é importante. Se você tentar capturar um tipo de exceção de superclasse antes de um tipo de subclasse, o compilador deve gerar erros. Explique por que ocorrem esses erros.

14.25 Escreva um programa Java que mostra um construtor passando as informações sobre falha do construtor para um tratador de exceções após um bloco `try`.

14.26 Escreva um programa Java que ilustra o novo disparo de uma exceção.

14.27 Escreva um programa Java que mostra que um método com seu próprio bloco `try` não precisa capturar cada possível erro gerado dentro do `try`. Algumas exceções podem escorregar para outros escopos e neles ser tratadas.

15

Multithreading

Objetivos

- Entender a noção de *multithreading*.
- Perceber como o *multithreading* pode melhorar o desempenho.
- Entender como criar, gerenciar e destruir *threads*.
- Entender o ciclo de vida de uma *thread*.
- Estudar vários exemplos de sincronização de *threads*.
- Entender prioridades e escalonamento de *threads*.
- Entender *threads daemon* e grupos de *threads*.

The spider's touch, how exquisitely fine!

Feels at each thread, and lives along the line.

Alexander Pope

Uma pessoa com um relógio sabe que horas são; uma pessoa com dois relógios nunca está segura.

Provérbio

Conversation is but carving!

Give no more to every guest,

Then he's able to digest.

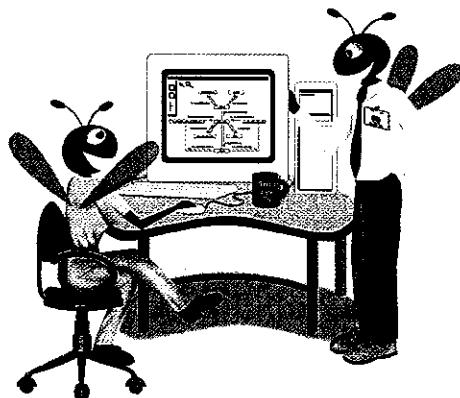
Jonathan Swift

Aprenda a trabalhar e a esperar.

Henry Wadsworth Longfellow

A definição mais geral de beleza: ... a Multiplicidade na Unidade

Samuel Taylor Coleridge



Sumário do capítulo

- 15.1 Introdução**
- 15.2 Classe Thread: uma visão geral dos métodos de Thread**
- 15.3 Estados de threads: ciclo de vida de uma thread**
- 15.4 Prioridades de threads e escalonamento de threads**
- 15.5 Sincronização de threads**
- 15.6 Relacionamento produtor/consumidor sem sincronização de threads**
- 15.7 Relacionamento produtor/consumidor com sincronização de threads**
- 15.8 Relacionamento produtor/consumidor no buffer circular**
- 15.9 Threads daemon**
- 15.10 A interface Runnable**
- 15.11 Grupos de threads**
- 15.12 (Estudo de caso opcional) Pensando em objetos: multithreading**
- 15.13 (Opcional) Descobrindo padrões de projeto: padrões de projeto simultâneos**

Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão •

Exercícios

15.1 Introdução

Seria perfeito se pudéssemos “fazer uma coisa por vez” e “fazê-la bem”, mas não é assim que o mundo funciona. O corpo humano realiza uma grande variedade de operações *em paralelo* ou, como diremos em todo este capítulo, *simultaneamente*. A respiração, a circulação sanguínea e a digestão, por exemplo, podem ocorrer simultaneamente. Todos os sentidos – visão, olfato, tato, paladar e audição – podem ocorrer simultaneamente. Um automóvel pode estar acelerando, manobrando, com o ar condicionado e a música funcionando simultaneamente. Os computadores também executam operações simultaneamente. Hoje é comum os computadores pessoais de mesa estarem compilando um programa, imprimindo um arquivo e recebendo mensagens de correio eletrônico através de uma rede simultaneamente.

A simultaneidade é importante em nossas vidas. Ironicamente, apesar disso, a maioria das linguagens de programação não permite que os programadores especifiquem atividades simultâneas. Em vez disso, geralmente as linguagens de programação fornecem somente um conjunto simples de estruturas de controle que permite aos programadores realizar uma ação por vez e depois passar para a próxima ação depois que a anterior é terminada. O tipo de simultaneidade que os computadores realizam hoje é normalmente implementado como “primitivas” de sistemas operacionais disponíveis somente para “programadores de sistemas” altamente experientes.

A linguagem de programação Ada, desenvolvida pelo Departamento de Defesa dos Estados Unidos, deixou as primitivas de simultaneidade amplamente disponíveis para as empresas contratadas pelo Departamento de Defesa para construir sistemas de comando e controle. Mas a linguagem Ada não foi amplamente utilizada nas universidades e nas empresas comerciais.

Java é única, entre as linguagens de programação de uso geral e popular, no sentido de que torna as primitivas de simultaneidade disponíveis para o programador de aplicativos. O programador especifica que os aplicativos contêm *fluxos de execução (threads)**, cada *thread* designando uma parte de um programa que pode ser executado simultaneamente com outras *threads*. Esse recurso, chamado *multithreading* ou *multiescalonamento*, oferece ao programador Java recursos poderosos, não disponíveis em C e C++, as linguagens em que Java é baseada. C e C++ são chamadas linguagens *de uma única thread*. [Nota: em muitas plataformas de computadores, os programas C e C++ podem executar multithreading usando bibliotecas de código específicas para o sistema.]

* N. de T. Embora o termo *thread* venha sendo traduzido por “fluxo de execução”, preferimos mantê-lo no original por se tratar de uso já consagrado no meio acadêmico.



Observação de engenharia de software 15.1

Diferente de muitas linguagens que não têm multithreading predefinido (como C e C++) e devem, portanto, fazer chamadas a primitivas de multithreading do sistema operacional, Java inclui primitivas de multithreading como parte da própria linguagem (na realidade, nas classes `Thread`, `ThreadGroup`, `ThreadLocal` e `ThreadDeath` do pacote `java.lang`). Isso incentiva o uso de multithreading por uma parte maior da comunidade de programação de aplicativos.

Discutiremos muitas aplicações da programação simultânea. Quando os programas baixam arquivos grandes como clipes de áudio ou videoclipes da World Wide Web, não queremos esperar até que um clipe inteiro seja baixado antes de iniciar a reprodução. Então podemos colocar múltiplas *threads* para trabalhar: uma que descarrega o clipe e outra que reproduz o clipe de modo que essas atividades, ou tarefas, possam continuar a ser executadas de forma simultânea. Para evitar a reprodução instável, coordenaremos as *threads* de modo que a *thread player* não inicie até que haja uma quantidade suficiente do clipe na memória para manter a *thread player* ocupada.

Outro exemplo de *multithreading* é a coleta de lixo automática de Java. Em C e C++, o programador é responsável por recuperar memória alocada dinamicamente. Java fornece uma *thread* coletora de lixo que automaticamente recupera a memória alocada dinamicamente de que o programa não precisa mais.



Dica de teste e depuração 15.1

Em C e C++, os programadores devem fornecer instruções explícitas para recuperar memória alocada dinamicamente. Quando a memória não é recuperada (porque o programador esqueceu de fazer isso, ou devido a um erro de lógica, ou porque uma exceção desvia o controle do programa), isso resulta em um erro muito comum denominado *perda de memória*, que pode acabar esgotando o estoque de memória livre e pode causar o término prematuro do programa. A coleta de lixo automática de Java elimina a grande maioria das perdas de memória, isto é, aquelas que se devem a objetos que ficaram órfãos (sem referência).

O coletor de lixo de Java executa como uma *thread* de baixa prioridade. Quando Java determina que não há mais nenhuma referência para um objeto, a linguagem marca o objeto para ser coletado como lixo em algum momento. A *thread* coletora de lixo é executada quando o tempo do processador está disponível e quando não há *threads* executáveis de prioridade mais alta. O coletor de lixo irá, porém, ser executado imediatamente quando o sistema estiver sem memória.



Dica de desempenho 15.1

Configurar uma referência para um objeto como `null` marca esse objeto para a coleta de lixo a qualquer momento (se não houver outras referências para o objeto). Isso pode ajudar a poupar memória num sistema em que uma variável local que se refere a um objeto não sai do escopo porque o método em que ela aparece é executado por um período longo.

Escrever programas *multithreaded*, ou *multiescalonados*, pode ser difícil. Embora a mente humana possa realizar muitas funções simultaneamente, as pessoas acham difícil ir e vir entre “correntes de pensamentos” paralelas. Para ver por que *multithreading* pode ser difícil de programar e de entender, tente a seguinte experiência: abra três livros na página 1. Agora tente ler os livros simultaneamente. Leia algumas palavras do primeiro livro, depois leia algumas palavras do segundo livro, depois leia algumas palavras do terceiro livro, depois volte para o início e leia as próximas palavras do primeiro livro e assim por diante. Em pouco tempo você descobrirá os desafios do *multithreading*: trocar de livro, ler rapidamente, lembrar onde parou em cada livro, mover o livro que você está lendo para mais perto a fim de poder enxergar o texto, afastar o livro que você não está lendo e, no meio de todo esse caos, tentar compreender o conteúdo dos livros!



Dica de desempenho 15.2

Um problema com os aplicativos de uma única thread é que as atividades demoradas devem ser completadas antes de outras atividades poderem iniciar. Em uma aplicação multithreaded, as threads podem compartilhar um processador (ou um conjunto de processadores), de modo que múltiplas tarefas sejam executadas em paralelo.

Embora Java seja talvez a linguagem de programação mais portável do mundo, certas partes da linguagem são dependentes de plataforma. Em particular, há diferenças entre as primeiras três plataformas Java implementadas, que são a implementação Solaris e a implementação Win32 (isto é, implementações baseadas em Windows para Windows 95 e Windows NT).

A plataforma Java Solaris executa uma *thread* de uma determinada prioridade até sua conclusão ou até que uma *thread* de prioridade mais alta fique pronta. Nesse ponto, ocorre a *preempção* (isto é, o processador é alocado à *thread* de prioridade mais alta enquanto a *thread* que está sendo executada antes deve esperar).

Nas implementações de 32 bits de Java para Windows 95 e Windows NT, as *threads* recebem uma fração de tempo. Isso significa que cada *thread* recebe uma quantidade limitada de tempo (chamada de *quantum* de tempo) para ser executada em um processador e, quando esse tempo expira, a *thread* é colocada em estado de espera enquanto todas as outras *threads* de igual prioridade obtêm suas oportunidades de utilizar seu quantum num esquema de *rodízio* (*round-robin*). Então a *thread* original retoma a execução. Portanto, no Windows 95 e no Windows NT, uma *thread* em execução pode sofrer preempção por uma *thread* de prioridade igual, ao passo que, na implementação Solaris, uma *thread* de Java em execução somente pode sofrer preempção por uma *thread* de prioridade mais alta. Espera-se que futuros sistemas Java Solaris também realizem o fracionamento de tempo.



Dica de portabilidade 15.1

O multithreading de Java é dependente de plataforma. Portanto, um aplicativo multithreaded pode comportar-se de maneira diferente em implementações diferentes de Java.

15.2 Classe Thread: uma visão geral dos métodos de Thread

Nesta seção, damos uma visão geral dos vários métodos relacionados com *thread* na Java API. Utilizamos muitos desses métodos em exemplos de código ativo ao longo do capítulo. O leitor deve consultar diretamente a Java API para obter mais detalhes sobre a utilização de cada método, especialmente as exceções disparadas por cada método.

A classe **Thread** (pacote `java.lang`) tem vários construtores. O construtor

```
public Thread( String threadName )
```

constrói um objeto **Thread** cujo nome é `threadName`. O construtor

```
public Thread()
```

constrói uma **Thread** cujo nome é "Thread- " concatenado com um número, como `Thread-1`, `Thread-2` e assim por diante.

O código que "faz o trabalho de verdade" de uma *thread* é colocado em seu método `run`. O método `run` pode ser sobreescrito em uma subclasse de **Thread** ou implementado em um objeto **Runnable**; **Runnable** é uma interface Java importante que estudamos na Seção 15.10.

O programa dispara a execução de uma *thread* chamando o método `start` da *thread*, que, por sua vez, chama o método `run`. Depois de `start` disparar a *thread*, `start` retorna para seu chamador imediatamente. O chamador então é executado simultaneamente com a *thread* disparada. O método `start` dispara uma **IllegalThreadStateException** se a *thread* que ele está tentando iniciar já tiver sido iniciada.

O método `static sleep` é chamado com um argumento que especifica quanto tempo a *thread* atualmente em execução deve dormir (em milissegundos); enquanto uma *thread* dorme, ela não disputa o processador; assim, outras *threads* podem ser executadas. Isso pode dar para as *threads* de prioridade mais baixa uma oportunidade de serem executadas.

O método `interrupt` é chamado para interromper uma *thread*. O método `static interrupted` devolve `true` se a *thread* atual foi interrompida e `false` caso contrário. O programa pode invocar o método `isInterrupted` de uma *thread* específica para determinar se essa *thread* foi interrompida.

O método `isAlive` devolve `true` se `start` foi chamado por uma dada *thread* e a *thread* não estiver morta (isto é, seu método de controle `run` não completou a execução).

O método `setName` configura o nome de uma **Thread**. O método `getName` devolve o nome da **Thread**. O método `toString` devolve um `String` que consiste no nome, na prioridade e no `ThreadGroup` da *thread* (discutidos na Seção 15.11).

O método `static currentThread` devolve uma referência para a **Thread** atualmente em execução.

O método `join` espera que a **Thread** para a qual a mensagem foi enviada morra antes que a **Thread** chamadora possa prosseguir; nenhum argumento ou um argumento de 0 milissegundo para o método `join` indica que a **Thread** atual irá esperar eternamente que a **Thread** de destino morra antes de a **Thread** chamadora prosseguir. Essa espera pode ser perigosa; pode conduzir a dois problemas particularmente sérios chamados *deadlock** e *adiamento indefinido*. Discutiremos esses problemas em breve.

* N. de T. "... um processo, um programa ou uma *thread*..."

Dica de teste e depuração 15.2

O método `dumpStack` é útil para depurar aplicativos multithreaded. O programa chama o método `static dumpStack` para imprimir um monitoramento da pilha de chamadas de métodos para a `Thread` atual.

15.3 Estados de threads: ciclo de vida de uma thread

Dizemos que uma *thread*, a qualquer momento, encontra-se em um dos vários *estados de thread* (ilustrados na Fig. 15.1). Digamos que uma *thread* que acaba de ser criada está no estado de *nascimento*. A *thread* permanece nesse estado até que o programa chame o método `start` da *thread*, o que faz com que a *thread* passe para o estado *pronta* (também conhecido como estado *executável*). A *thread* *pronta* de prioridade mais alta entra no *estado em execução* quando o sistema aloca um processador para a *thread*. Uma *thread* entra no estado *morta* quando seu método `run` completa ou termina por alguma razão – uma *thread morta* acabará sendo descartado pelo sistema em algum momento.

Uma maneira comum de uma *thread em execução* entrar no *estado bloqueada* é quando ela emite uma solicitação de entrada/saída. Nesse caso, a *thread bloqueada* torna-se *pronta* quando a E/S pela qual está esperando é concluída. A *thread bloqueada* não pode utilizar um processador, mesmo se houver um disponível.

Quando o programa chama o método `sleep` em uma *thread em execução*, essa *thread* entra no estado *adormecida*. A *thread adormecida* torna-se *pronta* depois que o tempo designado para dormir expira. Ela não pode utilizar um processador mesmo se houver um disponível. Se o programa chama o método `interrupt` para uma *thread adormecida*, aquela *thread* sai do estado adormecida e se torna pronta para ser executada.

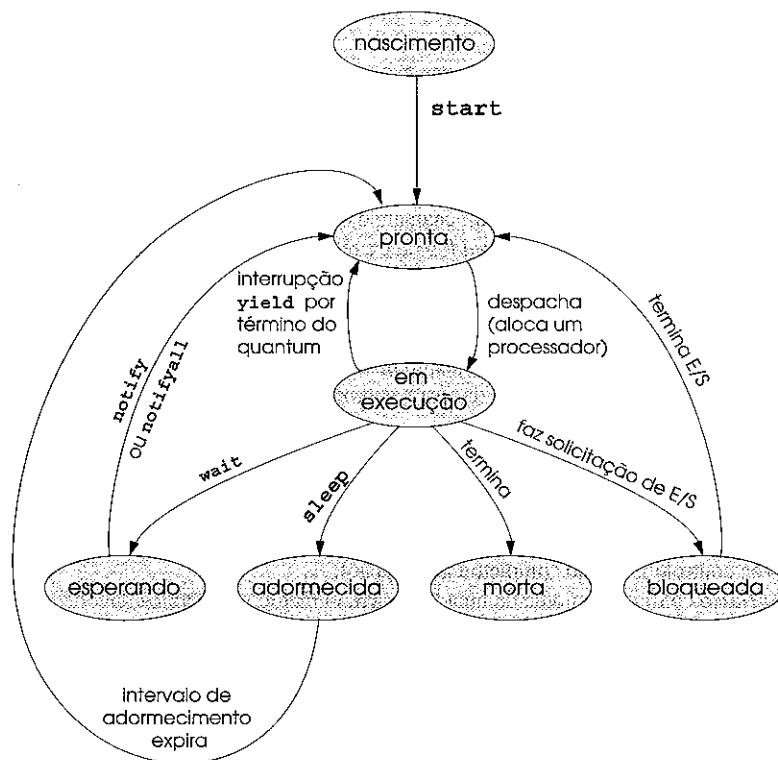


Fig. 15.1 Ciclo de vida de uma *thread*.

Quando uma *thread em execução* chama `wait`, ela entra num estado de *espera* pelo objeto particular para o qual `wait` foi chamado. A *thread* no estado de *espera* por um objeto particular torna-se *pronta* quando uma chamada para `notify` é emitida por outra *thread* associada com aquele objeto. Todas as *threads* no estado de *espera* por um objeto dado tornam-se *prontas* quando uma chamada para `notifyAll` é feita por outra *thread* associada com aquele objeto. Os métodos `wait`, `notify` e `notifyAll` serão discutidos em mais profundidade brevemente, quando pensarmos em monitores.

A *thread* entra no *estado morta* quando seu método `run` completa ou dispara uma exceção não-capturada.

15.4 Prioridades de *threads* e escalonamento de *threads*

Todo *applet* ou aplicativo Java é *multithreaded*. Toda *thread* Java tem uma prioridade no intervalo de `Thread.MIN_PRIORITY` (uma constante igual a 1) a `Thread.MAX_PRIORITY` (uma constante igual a 10). Por *default*, cada *thread* recebe prioridade `Thread.NORM_PRIORITY` (uma constante igual a 5). Cada nova *thread* herda a prioridade da *thread* que a cria.

Algumas plataformas Java suportam um conceito denominado *fracionamento de tempo* (*timeslicing*) e algumas não. Sem fracionamento de tempo, cada *thread* em um conjunto de *threads* de igual prioridade é executada até sua conclusão (a menos que a *thread* deixe o estado em execução e entre no estado de espera, adormecida ou bloqueada, ou a *thread* seja interrompida por uma *thread* de prioridade mais alta) antes que as colegas daquela *thread* obtenham uma oportunidade de ser executadas. Com o fracionamento de tempo, cada *thread* recebe um breve período de tempo do processador, chamado de *quantum*, durante o qual essa *thread* pode ser executada. No fim do quantum, mesmo se essa *thread* não tiver terminado a execução, o sistema operacional tira o controle do processador dessa *thread* e o aloca para a próxima *thread* de igual prioridade (se existir alguma disponível).

O trabalho do *scheduler* (escalonador) de Java é manter a *thread* com a mais alta prioridade em execução o tempo todo e, se o fracionamento de tempo estiver disponível, assegurar que várias *threads* de prioridade igualmente alta sejam executadas cada uma por um quantum, num esquema de *rodízio* (isto é, essas *threads* podem receber uma fração de tempo). A Fig. 15.2 ilustra a fila de múltiplos níveis de prioridade de Java para *threads*. Na figura, as *threads* A e B são executados por um quantum em sistema de rodízio até que ambas as *threads* completem a execução. Em seguida, a *thread* C é executada até sua conclusão. Então, as *threads* D, E e F são executadas por um quantum, em rodízio, até todas completarem sua execução. Esse processo continua até que todas as *threads* sejam executadas até sua conclusão. Observe que novas *threads* de prioridade mais alta poderiam adiar – indefinidamente, como maior possibilidade – a execução de *threads* de prioridade mais baixa. Esse *adiamento indefinido* é freqüentemente chamado, metaforicamente, de *inanição*.

A prioridade de uma *thread* pode ser ajustada com o método `setPriority`, que recebe um argumento `int`. Se o argumento não estiver no intervalo de 1 a 10, `setPriority` dispara uma `IllegalArgumentException`. O método `getPriority` devolve a prioridade da *thread*.

A *thread* pode chamar o método `yield` para dar a outras *threads* uma oportunidade de serem executadas. Na verdade, sempre que uma *thread* de prioridade mais alta torna-se pronta, o sistema operacional faz a preempção da *thread*. Assim, ela não pode ceder lugar (`yield`) a uma *thread* de prioridade mais alta porque a primeira *thread* terá sofrido preempção quando a *thread* de prioridade mais alta ficar pronta. De maneira semelhante, `yield` sempre permite que a *thread* pronta de prioridade mais alta seja executada, de modo que, se apenas *threads* de prioridade mais baixa estiverem prontas no momento de uma chamada a `yield`, a *thread* atual será a *thread* de prioridade mais alta e continuará sendo executada. Portanto, a *thread* chama `yield` para dar a oportunidade das *threads* de igual prioridade serem executadas. Em um sistema de fracionamento de tempo, isso é desnecessário, uma vez que as *threads* de prioridade igual serão executadas por seu quantum (ou até elas perderem o processador por alguma outra razão) e outras *threads* de prioridade igual serão executadas em modo de *rodízio*. Assim, `yield` é apropriado para sistemas sem fracionamento de tempo em que uma *thread* normalmente seria executada até sua conclusão antes que outra *thread* de prioridade igual tivesse a oportunidade de ser executada.

Dica de desempenho 15.3



Nos sistemas sem fracionamento de tempo, as *threads* cooperantes de igual prioridade devem chamar periodicamente `yield` para permitir que seus pares prossigam regularmente.

*Dica de portabilidade 15.2*

Os applets e aplicativos Java devem ser programados para funcionar em todas as plataformas Java para atingir o objetivo de verdadeira portabilidade de Java. Ao projetar applets que utilizam threads, você deve considerar os recursos para uso de threads de todas as plataformas em que os applets e os aplicativos serão executados.

A *thread* é executada a menos que ela morra, que se torne bloqueada pelo sistema operacional por entrada/saída (ou alguma outra razão), que chame **sleep**, chame **wait**, chame **yield**, sofra preempção por uma *thread* de prioridade mais alta ou seu quantum expire. A *thread* com prioridade mais alta que a *thread* em execução pode tornar-se pronta (e portanto fazer preempção da *thread* em execução) se uma *thread adormecida* parar de dormir, se for completada a E/S para uma *thread* que esteja esperando essa E/S ou se um **notify** ou **notifyAll** for chamado sobre uma *thread* que chamou **wait**.

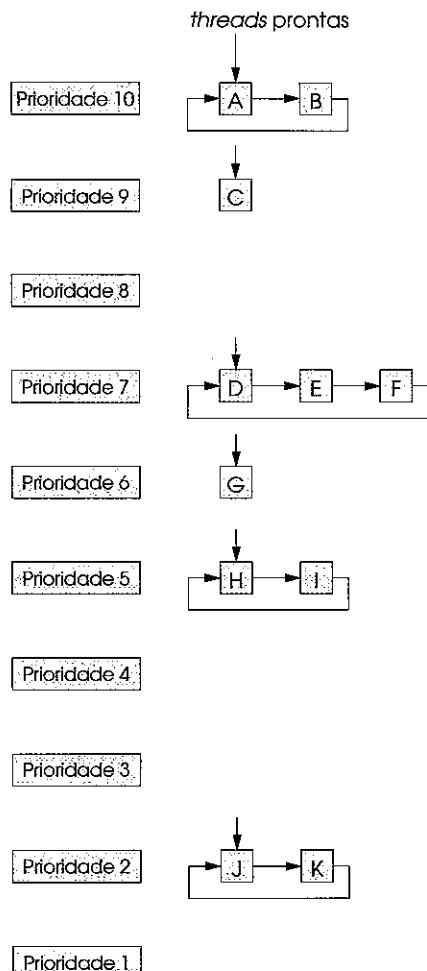


Fig. 15.2 O escalonamento por prioridade de *threads* em Java.

O aplicativo da Fig. 15.3 demonstra técnicas básicas de uso de *threads*, incluindo a criação de uma classe derivada de **Thread**, construção de uma **thread** e utilização do método **sleep** da classe **Thread**. Cada *thread* de execução que criamos no programa exibe seu nome depois de dormir por uma quantidade aleatória de tempo entre 0 e 5 segundos. Você verá que o método **main** (isto é, a *thread de execução main*) termina antes de o aplicativo terminar. O programa consiste em duas classes – **ThreadTester** (linhas 4 a 28) e **PrintThread** (linhas 33 a 71).

A classe **PrintThread** herda de **Thread**, de modo que cada objeto da classe possa ser executado em paralelo. A classe consiste na variável de instância **sleepTime**, num construtor e num método **run**. A variável **sleepTime** armazena um valor **int** aleatório escolhido quando o programa cria um objeto **PrintThread**. Cada objeto **PrintThread** dorme pela quantidade de tempo especificada por **sleepTime** e depois envia seu nome para a saída.

O construtor **PrintThread** (linhas 38 a 48) inicializa **sleepTime** com um inteiro aleatório entre 0 e 4999 (0 a 4,999 segundos). Depois o construtor envia para a saída o nome da *thread* e o valor de **sleepTime** para mostrar os valores para a **PrintThread** em particular que está sendo construída. O nome de cada *thread* é especificado como um argumento **String** para o construtor **PrintThread** e é passado para o construtor da superclasse na linha 40. [Nota: é possível permitir que a classe **Thread** escolha um nome para sua *thread* utilizando o construtor **default** da classe **Thread**.]

```

1 // Fig. 15.3: ThreadTester.java
2 // Mostra múltiplas threads imprimindo a intervalos de tempo aleatórios.
3
4 public class ThreadTester {
5
6     / cria e inicia as threads
7     public static void main( String args[] )
8     {
9         PrintThread thread1, thread2, thread3, thread4;
10
11        // cria quatro objetos PrintThread
12        thread1 = new PrintThread( "thread1" );
13        thread2 = new PrintThread( "thread2" );
14        thread3 = new PrintThread( "thread3" );
15        thread4 = new PrintThread( "thread4" );
16
17        System.err.println( "\nStarting threads" );
18
19        // começa a executar as PrintThreads
20        thread1.start();
21        thread2.start();
22        thread3.start();
23        thread4.start();
24
25        System.err.println( "Threads started\n" );
26    }
27
28 } // fim da classe ThreadTester
29
30 // Cada objeto desta classe escolhe um intervalo de adormecimento aleatório.
31 // Quando uma PrintThread é executada, ela imprime o nome, adormece,
32 // imprime o nome novamente e termina.
33 class PrintThread extends Thread {
34     private int sleepTime;
35
36     // o construtor PrintThread atribui um nome à thread
37     // chamando o construtor da superclasse Thread
38     public PrintThread( String name )
39     {

```

Fig. 15.3 Múltiplas *threads* imprimindo a intervalos de tempo aleatórios (parte 1 de 3).

```

40     super( name );
41
42     // adormece durante 0 a 5 segundos
43     sleepTime = ( int ) ( Math.random() * 5000 );
44
45     // exibe o nome e sleepTime
46     System.err.println(
47         "Name: " + getName() + ", sleep: " + sleepTime );
48 }
49
50 // controla a execução da thread
51 public void run()
52 {
53     // faz a thread adormecer durante um intervalo de tempo aleatório
54     try {
55         System.err.println( getName() + " going to sleep" );
56
57         // faz a thread adormecer
58         Thread.sleep( sleepTime );
59     }
60
61     // se a thread interrompeu enquanto estava adormecida,
62     // captura exceção e exibe mensagem de erro
63     catch ( InterruptedException interruptedException ) {
64         System.err.println( interruptedException.toString() );
65     }
66
67     // imprime o nome da thread
68     System.err.println( getName() + " done sleeping" );
69 }
70
71 } // fim da classe PrintThread

```

```

Name: thread1; sleep: 3593
Name: thread2; sleep: 2653
Name: thread3; sleep: 4465
Name: thread4; sleep: 1318

Starting threads
Threads started

thread1 going to sleep
thread2 going to sleep
thread3 going to sleep
thread4 going to sleep
thread4 done sleeping
thread2 done sleeping
thread1 done sleeping
thread3 done sleeping
Name: thread1; sleep: 2753
Name: thread2; sleep: 3199
Name: thread3; sleep: 2797
Name: thread4; sleep: 4639

Starting threads
Threads started

thread1 going to sleep

```

Fig. 15.3 Múltiplas threads imprimindo a intervalos de tempo aleatórios (parte 2 de 3).

```

thread2 going to sleep
thread3 going to sleep
thread4 going to sleep
thread1 done sleeping
thread3 done sleeping
thread2 done sleeping
thread4 done sleeping

```

Fig. 15.3 Múltiplas *threads* imprimindo a intervalos de tempo aleatórios (parte 3 de 3).

Quando o programa invoca o método **start** de uma **PrintThread** (herdado de **Thread**), o objeto **PrintThread** passa para o estado *pronta*. Quando o sistema aloca um processador para o objeto **PrintThread**, ele entra no estado *em execução* e seu método **run** começa a ser executado. O método **run** (linhas 51 a 69) imprime um **String** na janela de comando para indicar que a *thread* irá dormir (linha 55) e então invoca o método **sleep** (linha 58) para colocar a *thread* em um estado *adormecida*. Neste ponto, a *thread* perde o processador e o sistema permite que outra *thread* seja executada. Quando a *thread* acorda, ela é colocada em um estado *pronta* novamente até que o sistema aloca para ela. Quando o objeto **PrintThread** entra novamente no estado *em execução*, a linha 68 envia o nome da *thread* para a saída (indicando que a *thread* não está mais dormindo), seu método **run** termina e o objeto *thread* entra no estado *morta*. Observe que o método **sleep** pode disparar uma **InterruptedException** verificada (se outra *thread* invocar o método **interrupt** da *thread* adormecida); portanto, **sleep** deve ser chamado em um bloco **try** (nesse exemplo, simplesmente enviamos para a saída a representação como **String** da exceção se ocorrer alguma). Note que esse exemplo usa **System.err** em vez de **System.out** para enviar linhas de texto para a saída. **System.err** representa o objeto de saída de erros padrão, que envia para a saída mensagens de erro (normalmente para a janela de comando). **System.out** faz saída com *buffer* – é possível que uma mensagem enviada para a saída com **System.out** não seja colocada na saída imediatamente. Por outro lado, **System.err** usa saída sem *buffer* – as mensagens aparecem imediatamente quando elas são enviadas para a saída. Usar **System.err** em um programa com múltiplas *threads* ajuda a assegurar que as mensagens de nosso programa são enviadas para a saída na ordem correta.

O método **main** da classe **ThreadTester** (linhas 7 a 26) cria quatro objetos da classe **PrintThread** (linhas 12 a 15) e invoca o método **start** da classe **Thread** sobre cada um (linhas 20 a 23) para colocar todos os quatro objetos **PrintThread** em um estado *pronta*. Observe que o programa termina a execução quando a última **PrintThread** acorda e imprime seu nome. Note também que o método **main** (isto é, a *thread* de execução **main**) termina depois de iniciar as quatro **PrintThreads**, mas o aplicativo não termina até que a última *thread* morra.

15.5 Sincronização de *threads*

Java utiliza *monitores* (como foi discutido por C.A.R. Hoare em seu artigo de 1974 citado no Exercício 15.24) para fazer sincronização. Todo objeto com métodos **synchronized** tem um monitor. O monitor permite que uma *thread* de cada vez execute um método **synchronized** sobre o objeto. Isso se faz *bloqueando* o objeto quando o programa invoca o método **synchronized** – também conhecido como *obtenção do bloqueio*. Se houver vários métodos **synchronized**, somente um método **synchronized** pode estar atuando sobre um objeto de cada vez; todas as outras *threads* que tentam invocar métodos **synchronized** devem esperar. Quando um método **synchronized** termina de ser executado, o bloqueio sobre o objeto é liberado e o monitor permite que a *thread* *pronta* de prioridade mais alta tente invocar um método **synchronized** para prosseguir. [Nota: Java também tem *blocos* de código **synchronized**, que são discutidos no exemplo da Seção 15.10.]

A *thread* que está sendo executada em um método **synchronized** pode determinar que ela não pode prosseguir, de modo que a *thread* chama **wait** voluntariamente. Isso tira a *thread* da disputa pelo processador e da disputa pelo objeto monitor. A *thread* agora espera no estado de *espera* enquanto as outras *threads* tentam entrar no objeto monitor. Quando uma *thread* que está executando um método **synchronized** termina ou satisfaz a condição pela qual outra *thread* pode estar esperando, a *thread* pode notificar (**notify**) uma *thread* em espera para ela se tornar novamente *pronta*. Neste ponto, a *thread* original pode tentar readquirir o bloqueio sobre o objeto monitor e ser executada. O **notify** atua como sinal para a *thread* em espera de que a condição pela qual a *thread* em espera estava esperando agora está satisfeita, de modo que a *thread* em espera pode entrar novamente no monitor. Se uma

thread chama **notifyAll**, então todas as *threads* que estão esperando o objeto tornam-se elegíveis para entrar novamente no monitor (isto é, todas são colocadas em um estado *pronta*). Lembre-se de que somente uma dessas *threads* pode obter o bloqueio sobre o objeto de cada vez – outras *threads* que tentam adquirir o mesmo bloqueio serão *bloqueadas* pelo sistema operacional até que o bloqueio se torne disponível novamente. Os métodos **wait**, **notify** e **notifyAll** são herdados da classe **Object** por todas as classes. Então, qualquer objeto pode ter um monitor.



Erro comum de programação 15.1

As threads no estado de espera por um objeto monitor devem em algum momento ser acordadas explicitamente com um **notify** (ou **interrupt**), ou a thread esperará eternamente. Isso pode causar deadlock. [Nota: existem versões do método **wait** que recebem argumentos indicando o tempo de espera máximo. Se a thread não é notificada dentro do período de tempo especificado, a thread fica pronta para ser executada.]



Dica de teste e depuração 15.3

Certifique-se de que cada chamada para **wait** tem uma chamada correspondente a **notify** que em algum momento terminará a espera ou chamará **notifyAll** como salvaguarda.



Dica de desempenho 15.4

A sincronização feita para obter precisão em programas com múltiplas threads pode fazer com que os programas sejam executados mais lentamente devido à sobrecarga do monitor e à frequente movimentação de threads entre os estados em execução, de espera e pronta. Entretanto, não há muito a dizer sobre programas com múltiplas threads incorretos altamente eficientes!



Dica de teste e depuração 15.4

O bloqueio que ocorre com a execução dos métodos **synchronized** pode levar a um deadlock se os bloqueios não são nunca liberados. Quando ocorrem exceções, o mecanismo de exceção de Java coordena com o mecanismo de sincronização de Java para liberar bloqueios de sincronização adequados a fim de evitar esse tipo de deadlocks.

Os objetos monitores mantêm uma lista de todas as *threads* que esperam para entrar no objeto monitor para executar métodos **synchronized**. A *thread* é inserida na lista e espera pelo objeto se essa *thread* chamar um método **synchronized** do objeto enquanto outra *thread* já está sendo executada em um método **synchronized** desse objeto. A *thread* também é inserida na lista se ela chama **wait** enquanto está operando dentro do objeto. Entretanto, é importante distinguir entre *threads* em espera que bloquearam porque o monitor estava ocupado e *threads* que explicitamente chamaram **wait** dentro do monitor. Na conclusão de um método **synchronized**, as *threads* externas que bloquearam porque o monitor estava ocupado podem prosseguir para entrar no objeto. As *threads* que invocaram **wait** explicitamente só podem prosseguir quando forem notificadas através de uma chamada feita por outra *thread* para **notify** ou **notifyAll**. Quando é aceitável que uma *thread* em espera prossiga, o *scheduler* seleciona a *thread* com a prioridade mais alta.



Erro comum de programação 15.2

Ocorrerá um erro se uma thread emitir um **wait**, **notify** ou **notifyAll** sobre um objeto sem ter adquirido um bloqueio para esse objeto. Isso faz com que uma **IllegalMonitorStateException** seja disparada.

15.6 Relacionamento produtor/consumidor sem sincronização de threads

Em um relacionamento produtor/consumidor, a *thread produtora* que chama um método *produzir* pode ver que a *thread consumidora* não leu a última mensagem de uma região compartilhada da memória denominada *buffer*, de modo que a *thread produtora* chamará **wait**. Quando a *thread consumidora* ler a mensagem, ele chamará **notify** para permitir que uma produtora em espera prossiga. Quando a *thread consumidora* entra no monitor e descobre o *buffer* vazio, ele chama **wait**. A produtora que encontra o *buffer* vazio grava no *buffer*, e depois chama **notify**; assim, a consumidora em espera pode prosseguir.

Dados compartilhados podem ficar corrompidos se não sincronizarmos o acesso entre múltiplas *threads*. Considere um relacionamento produtor/consumidor em que uma *thread produtora* deposita uma sequência de números (utilizamos 1, 2, 3,...) em um escaninho na memória compartilhada. A *thread consumidora* lê esses dados da memória compartilhada e imprime os dados. Imprimimos o que o produtor produz à medida que ele produz e o que o consu-

midor consome à medida que ele consome. O programa da Fig. 15.4 demonstra um produtor e um consumidor que acessam uma única célula de memória compartilhada (a variável `int sharedInt` na Fig. 15.6), sem nenhuma sincronização. As *threads* não são sincronizadas, assim os dados podem se perder se o produtor colocar novos dados no escaninho antes de o consumidor consumir os dados anteriores. Ademais, os dados “podem ser duplicados” se o consumidor consumir os dados novamente antes que o produtor produza o próximo item. Para mostrar essas possibilidades, a *thread* consumidora nesse exemplo mantém um total de todos os valores que ela lê. A *thread* produtora produz valores de 1 a 10. Se a consumidora lesse cada valor produzido somente uma vez, o total seria 55. Entretanto, se você executar esse programa várias vezes, verá que o total raramente é, se for alguma vez, 55.

O programa consiste em quatro classes – `ProduceInteger` (Fig. 15.4), `ConsumeInteger` (Fig. 15.5), `HoldIntegerUnsynchronized` (Fig. 15.6) e `SharedCell` (Fig. 15.7).

A classe `ProduceInteger` (Fig. 15.4) – uma subclasse de `Thread` – consiste na variável de instância `sharedObject` (linha 4), um construtor (linhas 7 a 11) e um método `run` (linhas 15 a 37). O construtor inicializa a variável de instância `sharedObject` (linha 10) para fazer referência ao objeto `HoldIntegerUnsynchronized shared`, que foi passado como argumento. O método `run` da classe `ProduceInteger` (linhas 15 a 37) consiste em uma estrutura `for` que repete um laço 10 vezes. Cada iteração do laço primeiro invoca o método `sleep` para colocar o objeto `ProduceInteger` no estado *adormecido* por um intervalo de tempo aleatório entre 0 e 3 segundos. Quando a *thread* acorda, a linha 31 chama o método `setSharedInt` da classe `HoldIntegerUnsynchronized` e passa o valor da variável de controle `count` para configurar a variável de instância `sharedInt` do objeto compartilhado. Quando o laço se completa, as linhas 34 a 36 exibem uma linha de texto na janela de comando que indica que a *thread* terminou de produzir os dados e que está terminando (isto é, ela morre).

```

1 // Fig. 15.4: ProduceInteger.java
2 // Definição da classe com threads ProduceInteger
3 public class ProduceInteger extends Thread {
4     private HoldIntegerUnsynchronized sharedObject;
5
6     // inicializa o objeto thread ProduceInteger
7     public ProduceInteger( HoldIntegerUnsynchronized shared ) {
8         super( "ProduceInteger" );
9         sharedObject = shared;
10    }
11
12
13    // A thread ProduceInteger repete um laço 10 vezes e
14    // chama o método setSharedInt de sharedObject a cada vez
15    public void run()
16    {
17        for ( int count = 1; count <= 10; count++ ) {
18
19            // adormece durante um intervalo de tempo aleatório
20            try {
21                Thread.sleep( ( int )( Math.random() * 3000 ) );
22            }
23
24            // processa InterruptedException enquanto adormecida
25            catch( InterruptedException exception ) {
26                System.err.println( exception.toString() );
27            }
28
29            // chama o método sharedObject
30            // a partir desta thread de execução
31            sharedObject.setSharedInt( count );
32        }
33
34        System.err.println(
35            getName() + " finished producing values" +

```

Fig. 15.4 Classe `ProduceInteger` representa o produtor em um relacionamento produtor/consumidor (parte 1 de 2).

```

36         "\nTerminating " + getName() );
37     }
38
39 } // fim da classe ProduceInteger

```

Fig. 15.4 Classe `ProduceInteger` representa o produtor em um relacionamento produtor/consumidor (parte 2 de 2).

A classe `ConsumeInteger` (Fig. 15.5) – uma subclasse de `Thread` – consiste em uma variável de instância `sharedObject` (linha 4), um construtor (linhas 7 a 11) e um método `run` (linhas 15 a 39). O construtor inicializa a variável de instância `sharedObject` (linha 10) para fazer referência ao objeto `HoldIntegerUnsynchronized shared` que foi passado como argumento. O método `run` da classe `ConsumeInteger` (linhas 15 a 39) consiste em uma estrutura `do/while` que repete o laço até que o valor 10 seja lido do objeto `HoldIntegerUnsynchronized` ao qual `sharedObject` faz referência. Cada iteração do laço invoca o método `sleep` para colocar o objeto `ConsumeInteger` no estado adormecido por um intervalo de tempo aleatório entre 0 e 3 segundos. Em seguida, a linha 31 invoca o método `getSharedInt` da classe `HoldIntegerUnsynchronized` para obter o valor da variável de instância `sharedInt` do objeto compartilhado. Então, a linha 32 adiciona o valor devolvido por `getSharedInt` à variável `sum`. Quando o laço se completa, a thread `ConsumeInteger` exibe uma linha na janela de comando indicando que terminou de consumir dados e termina (isto é, a thread morre).

```

1 // Fig. 15.5: ConsumeInteger.java
2 // Definição da classe com threads ConsumeInteger
3 public class ConsumeInteger extends Thread {
4     private HoldIntegerUnsynchronized sharedObject;
5
6     // inicializa o objeto thread ConsumeInteger
7     public ConsumeInteger( HoldIntegerUnsynchronized shared ) {
8     {
9         super( "ConsumeInteger" );
10        sharedObject = shared;
11    }
12
13    // a thread ConsumeInteger repete um laço até receber 10
14    // do método getSharedInt de sharedObject
15    public void run()
16    {
17        int value, sum = 0;
18
19        do {
20
21            // adormece durante um intervalo de tempo aleatório
22            try {
23                Thread.sleep( (int) ( Math.random() * 3000 ) );
24            }
25
26            // processa InterruptedException enquanto adormecida
27            catch( InterruptedException exception ) {
28                System.err.println( exception.toString() );
29            }
30
31            value = sharedObject.getSharedInt();
32            sum += value;
33
34        } while ( value != 10 );
35
36        System.err.println(

```

Fig. 15.5 Classe `ConsumeInteger` representa o consumidor em um relacionamento produtor/consumidor (parte 1 de 2).

```

37         getName() + " retrieved values totaling: " + sum +
38         "\nTerminating " + getName() );
39     }
40
41 } // fim da classe ConsumeInteger

```

Fig. 15.5 Classe `ConsumeInteger` representa o consumidor em um relacionamento produtor/consumidor (parte 2 de 2).

A classe `HoldIntegerUnsynchronized` consiste em uma variável de instância `sharedInt` (linha 4), o método `setSharedInt` (linhas 7 a 13) e o método `getSharedInt` (linhas 16 a 22). Os métodos `setSharedInt` e `getSharedInt` não sincronizam o acesso à variável de instância `sharedInt`. Note que cada método usa o método `static Thread currentThread` para obter uma referência para a *thread* que está sendo executada correntemente e então usa o método `getName` de `Thread` para obter o nome da *thread*.

O método `main` da classe `SharedCell` (linhas 6 a 20) instancia o objeto compartilhado `HoldIntegerUnsynchronized sharedObject` e utiliza-o como o argumento para os construtores do objeto `ProduceInteger producer` e do objeto `ConsumeInteger consumer`. O objeto `sharedObject` contém os dados que serão compartilhados entre as duas *threads*. Em seguida, o método `main` invoca o método `start` da classe `Thread` sobre as *threads* `producer` e `consumer` para colocá-las no estado *pronta*. Isto dispara as *threads*.

```

1 // Fig. 15.6: HoldIntegerUnsynchronized.java
2 // Definição da classe HoldIntegerUnsynchronized.
3 public class HoldIntegerUnsynchronized {
4     private int sharedInt = -1;
5
6     // método não-sincronizado para colocar o valor em sharedInt
7     public void setSharedInt( int value )
8     {
9         System.err.println( Thread.currentThread().getName() +
10             " setting sharedInt to " + value );
11
12         sharedInt = value;
13     }
14
15     // método não-sincronizado devolve o valor de sharedInt
16     public int getSharedInt()
17     {
18         System.err.println( Thread.currentThread().getName() +
19             " retrieving sharedInt value " + sharedInt );
20
21         return sharedInt;
22     }
23
24 } // fim da classe HoldIntegerUnsynchronized

```

Fig. 15.6 Classe `HoldIntegerUnsynchronized` mantém os dados compartilhados entre as *threads* produtora e consumidora.

```

1 // Fig. 15.7: SharedCell.java
2 // Mostra múltiplas threads modificando um objeto compartilhado.
3 public class SharedCell {
4
5     // executa o aplicativo
6     public static void main( String args[] )

```

Fig. 15.7 Threads modificando um objeto compartilhado sem sincronização (parte 1 de 2).

```

7   {
8     HoldIntegerUnsynchronized sharedObject =
9       new HoldIntegerUnsynchronized();
10
11    // cria as threads
12    ProduceInteger producer =
13      new ProduceInteger( sharedObject );
14    ConsumeInteger consumer =
15      new ConsumeInteger( sharedObject );
16
17    // inicia as threads
18    producer.start();
19    consumer.start();
20  }
21
22 } // fim da classe SharedCell

```

```

ConsumeInteger retrieving sharedInt value -1
ConsumeInteger retrieving sharedInt value -1
ProduceInteger setting sharedInt to 1
ProduceInteger setting sharedInt to 2
ConsumeInteger retrieving sharedInt value 2
ProduceInteger setting sharedInt to 3
ProduceInteger setting sharedInt to 4
ProduceInteger setting sharedInt to 5
ConsumeInteger retrieving sharedInt value 5
ProduceInteger setting sharedInt to 6
ProduceInteger setting sharedInt to 7
ProduceInteger setting sharedInt to 8
ConsumeInteger retrieving sharedInt value 8
ConsumeInteger retrieving sharedInt value 8
ProduceInteger setting sharedInt to 9
ConsumeInteger retrieving sharedInt value 9
ConsumeInteger retrieving sharedInt value 9
ProduceInteger setting sharedInt to 10
ProduceInteger finished producing values
Terminating ProduceInteger
ConsumeInteger retrieving sharedInt value 10
ConsumeInteger retrieved values totaling: 49
Terminating ConsumeInteger

```

Fig. 15.7 Threads modificando um objeto compartilhado sem sincronização (parte 2 de 2).

Idealmente, gostaríamos que cada valor produzido pelo objeto `ProduceInteger` fosse consumido exatamente uma vez pelo objeto `ConsumeInteger`. Entretanto, quando estudamos a saída da Fig. 15.7, vemos que os valores 1, 3, 4, 6 e 7 são perdidos (isto é, nunca são vistos pelo consumidor) e que os valores 8 e 9 são recuperados incorretamente mais de uma vez pelo consumidor. Além disso, note que o consumidor recuperou duas vezes o valor -1 (o valor *default* de `sharedInt` configurado na linha 5 da Fig. 15.6) antes de o produtor ter sequer atribuído 1 à variável `sharedInt`. Esse exemplo demonstra claramente que o acesso a dados compartilhados por *threads* simultâneas deve ser cuidadosamente controlado ou um programa pode produzir resultados incorretos.

Para resolver os problemas de dados perdidos e duplicados no exemplo anterior, iremos *sincronizar* o acesso das *threads* simultâneas produtora e consumidora aos dados compartilhados. Cada método utilizado por um produtor ou consumidor para acessar os dados compartilhados é declarado com a palavra-chave `synchronized`. Quando um método declarado `synchronized` está sendo executado em um objeto, o objeto é *bloqueado* de modo que nenhum outro método `synchronized` possa ser executado nesse objeto ao mesmo tempo.

15.7 Relacionamento produtor/consumidor com sincronização de threads

O aplicativo na Fig. 15.8 demonstra um produtor e um consumidor acessando uma célula de memória compartilhada com sincronização, de modo que o consumidor consome somente depois de o produtor produzir um valor. As classes **ProduceInteger** (Fig. 15.8), **ConsumeInteger** (Fig. 15.9) e **SharedCell** (Fig. 15.11) são idênticas às Figs. 15.4, 15.5 e 15.7, exceto pelo fato de que utilizam a nova classe **HoldIntegerSynchronized** nesse exemplo.

A classe **HoldIntegerSynchronized** (Fig. 15.10) contém duas variáveis de instância – **sharedInt** (linha 6) e **writeable** (linha 7). Além disso, o método **setSharedInt** (linhas 12 a 39) e o método **getSharedInt** (linhas 44 a 70) são agora métodos **synchronized**. Os objetos da classe **HoldIntegerSynchronized** têm monitores, porque **HoldIntegerSynchronized** contém métodos **synchronized**. A variável de instância **writeable** é conhecida como *variável de condição* do monitor – é uma boolean utilizada pelos métodos **setSharedInt** e **getSharedInt** da classe **HoldIntegerSynchronized**. Se **writeable** for **true**, **setSharedInt** pode colocar um valor na variável **sharedInt**, porque a variável neste momento não contém informação. Entretanto, isto significa que **getSharedInt** neste momento não pode ler o valor de **sharedInt**. Se **writeable** for **false**, **getSharedInt** pode ler um valor da variável **sharedInt** porque a variável neste momento contém informação. Entretanto, significa que **setSharedInt**, neste momento, não pode colocar um valor em **sharedInt**.

```

1 // Fig. 15.8: ProduceInteger.java
2 // Definição da classe com threads ProduceInteger
3 public class ProduceInteger extends Thread {
4     private HoldIntegerSynchronized sharedObject;
5
6     // inicializa o objeto thread ProduceInteger
7     public ProduceInteger( HoldIntegerSynchronized shared ) {
8         super( "ProduceInteger" );
9         sharedObject = shared;
10    }
11
12
13    // A thread ProduceInteger repete um laço 10 vezes e chama
14    // o método setSharedInt de sharedObject a cada vez
15    public void run()
16    {
17        for ( int count = 1; count <= 10; count++ ) {
18
19            // adormece durante um intervalo de tempo aleatório
20            try {
21                Thread.sleep( ( int )( Math.random() * 3000 ) );
22            }
23
24            // processa InterruptedException enquanto adormecida
25            catch( InterruptedException exception ) {
26                System.err.println( exception.toString() );
27            }
28
29            // chama o método sharedObject a partir
30            // desta thread de execução
31            sharedObject.setSharedInt( count );
32        }
33
34        System.err.println(
35            getName() + " finished producing values" +
36            "\nTerminating " + getName() );
37    }
38
39 } // fim da classe ProduceInteger

```

Fig. 15.8 Classe **ProduceInteger** representa o produtor em um relacionamento produtor/consumidor.

```

1 // Fig. 15.9: ConsumeInteger.java
2 // Definição da classe com threads ConsumeInteger
3 public class ConsumeInteger extends Thread {
4     private HoldIntegerSynchronized sharedObject;
5
6     // inicializa o objeto thread ConsumerInteger
7     public ConsumeInteger( HoldIntegerSynchronized shared ) {
8     {
9         super( "ConsumeInteger" );
10        sharedObject = shared;
11    }
12
13    // a thread ConsumeInteger repete um laço até receber 10
14    // do método getSharedInt de sharedObject
15    public void run() {
16    {
17        int value, sum = 0;
18
19        do {
20
21            // adormece durante um intervalo de tempo aleatório
22            try {
23                Thread.sleep( (int) ( Math.random() * 3000 ) );
24            }
25
26            // processa InterruptedException enquanto adormecida
27            catch( InterruptedException exception ) {
28                System.err.println( exception.toString() );
29            }
30
31            value = sharedObject.getSharedInt();
32            sum += value;
33
34        } while ( value != 10 );
35
36        System.err.println(
37            getName() + " retrieved values totaling: " + sum +
38            "\nTerminating " + getName() );
39    }
40
41 } // fim da classe ConsumeInteger

```

Fig. 15.9 Classe `ConsumeInteger` representa o consumidor em um relacionamento produtor/consumidor.

Quando o objeto `thread ProduceInteger` invoca o método sincronizado `setSharedInt` (linha 31 da Fig. 15.8), a `thread` adquire um bloqueio sobre o objeto monitor `HoldIntegerSynchronized`. A estrutura `while` nas linhas 14 a 25 testa a variável `writeable` com a condição `!writeable`. Se essa condição for `true`, a `thread` invoca o método `wait`. Isso coloca o objeto `thread ProduceInteger` que chamou o método `setSharedInt` no estado de *espera* pelo objeto `HoldIntegerSynchronized` e libera o bloqueio sobre ele. Agora, outra `thread` pode invocar um método `synchronized` sobre o objeto `HoldIntegerSynchronized`.

O objeto `ProduceInteger` permanece no estado de *espera* até ser *notificado* de que pode prosseguir – ponto em que ele entra no estado *pronto* e espera que o sistema lhe atribua um processador. Quando o objeto `ProduceInteger` entra novamente no estado *em execução*, ele readquire o bloqueio sobre o objeto `HoldIntegerSynchronized` implicitamente, e o método `setSharedInt` continua sendo executado na estrutura `while` com a próxima instrução depois de `wait`. Não há mais instruções, assim o programa reavalia a condição `while`. Se a condição for `false`, o programa envia uma linha para a janela de comando para indicar que o produtor está configurando `sharedInt` com um novo valor, atribui `value` a `sharedInt`, configura `writeable` como `false` para indicar que agora a memória compartilhada está cheia (isto é, o consumidor pode ler o valor e o produtor ain-

da não pode colocar outro valor aí) e invoca o método **notify**. Se houver qualquer *thread* em espera, uma destas *threads* em espera passa para o estado *pronta*, indicando que a *thread* agora pode tentar sua tarefa novamente (logo que lhe for atribuído um processador). O método **notify** retorna imediatamente, e o método **setSharedInt** retorna ao seu chamador.

```

1 // Fig. 15.10: HoldIntegerSynchronized.java
2 // Definição da classe HoldIntegerSynchronized, que usa
3 // sincronização de threads para assegurar que as duas
4 // threads acessem sharedInt nos momentos apropriados.
5 public class HoldIntegerSynchronized {
6     private int sharedInt = -1;
7     private boolean writeable = true;    // variável condicional
8
9     // método sincronizado permite que somente uma thread de cada vez
10    // invoque este método para atribuir o valor
11    // para um objeto HoldIntegerSynchronized particular
12    public synchronized void setSharedInt( int value )
13    {
14        while ( !writeable ) { // não é a vez do produtor
15
16            // a thread que chamou este método precisa esperar
17            try {
18                wait();
19            }
20
21            // processa exceção Interrupted enquanto uma thread estava em espera
22            catch ( InterruptedException exception ) {
23                exception.printStackTrace();
24            }
25        }
26
27        System.err.println( Thread.currentThread().getName() +
28            " setting sharedInt to " + value );
29
30        // atribui um novo valor para sharedInt
31        sharedInt = value;
32
33        // indica que o produtor não pode armazenar um outro valor
34        // até que o consumidor recupere o valor atual de sharedInt
35        writeable = false;
36
37        // diz a uma thread em espera para se tornar pronta
38        notify();
39    }
40
41    // método sincronizado permite que somente uma thread invoque
42    // este método de cada vez, para obter o valor de um
43    // objeto HoldIntegerSynchronized particular
44    public synchronized int getSharedInt()
45    {
46        while ( writeable ) { // não é a vez do consumidor
47
48            // a thread que chamou este método precisa esperar
49            try {
50                wait();
51            }
52

```

Fig. 15.10 Classe **HoldIntegerSynchronized** monitora o acesso a um inteiro compartilhado (parte 1 de 2).

```

53     // processa exceção Interrupted enquanto a thread estava esperando
54     catch ( InterruptedException exception ) {
55         exception.printStackTrace();
56     }
57 }
58
59     // indica que o produtor pode armazenar outro valor
60     // porque um consumidor acabou de recuperar o valor de sharedInt
61     writeable = true;
62
63     // diz a uma thread em espera para se tornar pronta
64     notify();
65
66     System.err.println( Thread.currentThread().getName() +
67         " retrieving sharedInt value " + sharedInt );
68
69     return sharedInt;
70 }
71
72 } // fim da classe HoldIntegerSynchronized

```

Fig. 15.10 Classe `HoldIntegerSynchronized` monitora o acesso a um inteiro compartilhado (parte 2 de 2).

Os métodos `getSharedInt` e `setSharedInt` são implementados de maneira semelhante. Quando o objeto `ConsumeInteger` invoca o método `getSharedInt`, a *thread* que chama adquire um bloqueio sobre o objeto `HoldIntegerSynchronized`. A estrutura `while` nas linhas 46 a 57 testa a variável `writeable`. Se `writeable` for `true` (isto é, não há nada a consumir), a *thread* invoca o método `wait`. Isso coloca o objeto `thread ConsumeInteger` que chamou o método `getSharedInt` no estado de *espera* pelo objeto `HoldIntegerSynchronized` e libera o bloqueio sobre ele, de modo que outros métodos `synchronized` possam ser invocados sobre o objeto. O objeto `ConsumeInteger` permanece no estado de *espera* até ser *notificado* de que pode prosseguir – ponto em que ele passa para o estado *pronto* e espera que lhe seja atribuído um processador. Quando o objeto `ConsumeInteger` entra novamente no estado *em execução*, a *thread* readquire o bloqueio sobre o objeto `HoldIntegerSynchronized` e o método `setSharedInt` continua a ser executado na estrutura `while` com a próxima instrução depois de `wait`. Não há mais instruções, portanto o programa testa a condição `while` novamente. Se a condição for `false`, `writeable` é configurado como `true` para indicar que a memória compartilhada agora está vazia e invoca o método `notify`. Se houver alguma *thread* em espera, uma destas *threads* em *espera* é colocada no estado *pronta*, indicando que ela agora pode tentar sua tarefa novamente (logo que lhe for atribuído um processador). O método `notify` retorna imediatamente. Assim, `getSharedInt` envia uma linha para a janela de comando para indicar que o consumidor está recuperando `sharedInt`, e depois devolve o valor de `sharedInt` para o chamador de `getSharedInt`.

Estude a saída na Fig. 15.11. Observe que cada inteiro produzido é consumido uma vez – nenhum valor se perde e nenhum valor é duplicado. Além disso, o consumidor não pode ler o valor até que o produtor o produza.

```

1 // Fig. 15.11: SharedCell.java
2 // Mostra múltiplas threads modificando um objeto compartilhado.
3 public class SharedCell {
4
5     // executa o aplicativo
6     public static void main( String args[] )
7     {
8         HoldIntegerSynchronized sharedObject =
9             new HoldIntegerSynchronized();
10

```

Fig. 15.11 Threads modificando um objeto compartilhado com sincronização (parte 1 de 2).

```

11     // cria as threads
12     ProduceInteger producer =
13         new ProduceInteger( sharedObject );
14     ConsumeInteger consumer =
15         new ConsumeInteger( sharedObject );
16
17     // inicia as threads
18     producer.start();
19     consumer.start();
20 }
21
22 } // fim da classe SharedCell

```

```

ProduceInteger setting sharedInt to 1
ConsumeInteger retrieving sharedInt value 1
ProduceInteger setting sharedInt to 2
ConsumeInteger retrieving sharedInt value 2
ProduceInteger setting sharedInt to 3
ConsumeInteger retrieving sharedInt value 3
ProduceInteger setting sharedInt to 4
ConsumeInteger retrieving sharedInt value 4
ProduceInteger setting sharedInt to 5
ConsumeInteger retrieving sharedInt value 5
ProduceInteger setting sharedInt to 6
ConsumeInteger retrieving sharedInt value 6
ProduceInteger setting sharedInt to 7
ConsumeInteger retrieving sharedInt value 7
ProduceInteger setting sharedInt to 8
ConsumeInteger retrieving sharedInt value 8
ProduceInteger setting sharedInt to 9
ConsumeInteger retrieving sharedInt value 9
ProduceInteger setting sharedInt to 10
ProduceInteger finished producing values
Terminating ProduceInteger
ConsumeInteger retrieving sharedInt value 10
ConsumeInteger retrieved values totaling: 55
Terminating ConsumeInteger

```

Fig. 15.11 *Threads* modificando um objeto compartilhado com sincronização (parte 2 de 2).

15.8 Relacionamento produtor/consumidor: o *buffer circular*

O programa das Figs. 15.8 a 15.11 acessa os dados compartilhados corretamente, mas ele pode não ter um desempenho ótimo. Uma vez que as *threads* estão executando de forma assíncrona, não podemos prever suas velocidades relativas. Se o produtor quer produzir mais rápido que o consumidor pode consumir, ele não pode fazer isto. Para permitir que o produtor continue a produzir, podemos utilizar um *buffer circular* que tenha células extras suficientes para tratar a produção “extra”. O programa das Figs. 15.12 a 15.16 demonstra um produtor e um consumidor que acessam um *buffer circular* (nesse caso, um *array* compartilhado de cinco células) com sincronização, de modo que o consumidor apenas consuma um valor quando houver um ou mais valores no *array* e o produtor apenas produza um valor quando houver uma ou mais células disponíveis no *array*. Esse programa é implementado como aplicativo de janelas que envia sua saída para uma *JTextArea*. O construtor da classe **SharedCell** cria os objetos **HoldIntegerSynchronized**, **ProduceInteger** e **ConsumeInteger**. O construtor do objeto **HoldIntegerSynchronized sharedObject** recebe uma referência para um objeto **JTextArea** no qual a saída do programa será exibida.

Este é o primeiro programa no qual usamos *threads* separadas para modificar o conteúdo exibido em componentes GUI Swing. A natureza da programação com múltiplas *threads* impede que o programador saiba exatamente quando uma *thread* será executada. Os componentes GUI Swing não são seguros quanto ao uso de *threads* – se múltiplas *threads* acessam um componente GUI Swing, os resultados podem não ser corretos. Todas as interações com componentes GUI Swing devem ser executadas a partir de uma *thread* de cada vez. Normalmente, esta *thread* é a *thread de despacho de eventos* (também conhecida como *thread* tratadora de eventos). A classe ***SwingUtilities*** (pacote `javax.swing`) fornece o método ***static invokeLater*** para ajudar neste processo. O método ***invokeLater*** recebe um argumento ***Runnable***. Veremos na próxima seção que os objetos ***Runnable*** têm um método ***run***. Na verdade, a classe ***Thread*** implementa a interface ***Runnable***, de modo que todas as ***Threads*** têm um método ***run***. As *threads* neste exemplo passam objetos da classe ***UpdateThread*** (Fig. 15.12) para o método ***invokeLater***. Cada objeto ***UpdateThread*** recebe uma referência para a ***JTextArea*** à qual deve anexar a saída e a mensagem a exibir. O método ***run*** da classe ***UpdateThread*** anexa a mensagem à ***JTextArea***. Quando o programa chama ***invokeLater***, a atualização do componente GUI será enfileirada para execução na *thread de despacho de eventos*. O método ***run*** então será invocado como parte da *thread de despacho de eventos*, assegurando que o componente GUI seja atualizado de uma maneira segura.



Erro comum de programação 15.3

Interações com componentes GUI Swing que mudam ou obtêm valores de propriedades dos componentes podem ter resultados incorretos se as interações forem executadas a partir de múltiplas threads.



Observação de engenharia de software 15.2

*Use o método ***static invokeLater*** de ***SwingUtilities*** para assegurar que todas as interações com a GUI são executadas a partir da *thread de despacho de eventos*.*

A classe ***ProduceInteger*** (Fig. 15.13) foi ligeiramente modificada em relação à versão apresentada na Fig. 15.8. A nova versão coloca sua saída em uma ***JTextArea***. ***ProduceInteger*** recebe uma referência para a ***JTextArea*** quando o programa chama o construtor de ***ProduceInteger***. Observe o uso do método ***invokeLater*** de ***SwingUtilities*** nas linhas 42 a 44 para assegurar que a GUI seja atualizada apropriadamente.

```

1 // Fig. 15.12: UpdateThread.java
2 // Classe que atualiza JTextArea com dados de saída.
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class UpdateThread extends Thread {
8     private JTextArea outputArea;
9     private String messageToOutput;
10
11    // inicializa outputArea e message
12    public UpdateThread( JTextArea output, String message )
13    {
14        outputArea = output;
15        messageToOutput = message;
16    }
17
18    // método chamado para atualizar outputArea
19    public void run()
20    {
21        outputArea.append( messageToOutput );
22    }
23
24 } // fim da classe UpdateThread

```

Fig. 15.12 *UpdateThread usada pelo método ***invokeLater*** de ***SwingUtilities*** para assegurar que a GUI seja atualizada apropriadamente.*

```

1 // Fig. 15.13: ProduceInteger.java
2 // Definição da classe com threads ProduceInteger
3
4 // Pacotes de extensão de Java
5 import javax.swing.*;
6
7 public class ProduceInteger extends Thread {
8     private HoldIntegerSynchronized sharedObject;
9     private JTextArea outputArea;
10
11    // inicializa ProduceInteger
12    public ProduceInteger( HoldIntegerSynchronized shared,
13                           JTextArea output )
14    {
15        super( "ProduceInteger" );
16
17        sharedObject = shared;
18        outputArea = output;
19    }
20
21    // a thread ProduceInteger repete um laço 10 vezes e chama
22    // o método setSharedInt de sharedObject a cada vez
23    public void run()
24    {
25        for ( int count = 1; count <= 10; count++ ) {
26
27            // adormece durante um intervalo de tempo aleatório
28            // Nota: intervalo encurtado propositalmente para encher o buffer
29            try {
30                Thread.sleep( (int) ( Math.random() * 500 ) );
31            }
32
33            // processa InterruptedException enquanto adormecida
34            catch( InterruptedException exception ) {
35                System.err.println( exception.toString() );
36            }
37
38            sharedObject.setSharedInt( count );
39        }
40
41        // atualiza o componente Swing da GUI
42        SwingUtilities.invokeLater( new UpdateThread( outputArea,
43                                              "\n" + getName() + " finished producing values" +
44                                              "\nTerminating " + getName() + "\n" ) );
45    }
46
47 } // fim da classe ProduceInteger

```

Fig. 15.13 Classe `ProduceInteger` representa o produtor em um relacionamento produtor/consumidor.

A classe `ConsumeInteger` (Fig. 15.14) foi ligeiramente modificada em relação à versão apresentada na Fig. 15.9. A nova versão coloca sua saída em uma `JTextArea`. `ConsumeInteger` recebe uma referência para a `JTextArea` quando o programa chama o construtor de `ConsumeInteger`. Note o uso do método `invokeLater` de `SwingUtilities` nas linhas 45 a 47 para assegurar que a GUI seja atualizada apropriadamente.

```

1 // Fig. 15.14: ConsumeInteger.java
2 // Definição da classe com threads ConsumeInteger

```

Fig. 15.14 Classe `ConsumeInteger` representa o consumidor em um relacionamento produtor/consumidor (parte 1 de 2).

```

3 // Pacotes de extensão de Java
4 import javax.swing.*;
5
6
7 public class ConsumeInteger extends Thread {
8     private HoldIntegerSynchronized sharedObject;
9     private JTextArea outputArea;
10
11    // inicializa ConsumeInteger
12    public ConsumeInteger( HoldIntegerSynchronized shared,
13                           JTextArea output )
14    {
15        super( "ConsumeInteger" );
16
17        sharedObject = shared;
18        outputArea = output;
19    }
20
21    // a thread ConsumeInteger repete o laço até receber 10
22    // do método getSharedInt de sharedObject
23    public void run()
24    {
25        int value, sum = 0;
26
27        do {
28
29            // adormece durante um intervalo de tempo aleatório
30            try {
31                Thread.sleep( (int) ( Math.random() * 3000 ) );
32            }
33
34            // processa InterruptedException enquanto adormecida
35            catch( InterruptedException exception ) {
36                System.err.println( exception.toString() );
37            }
38
39            value = sharedObject.getSharedInt();
40            sum += value;
41
42        } while ( value != 10 );
43
44        // atualiza o componente Swing da GUI
45        SwingUtilities.invokeLater( new UpdateThread( outputArea,
46                                         "\n" + getName() + " retrieved values totaling: " +
47                                         sum + "\nTerminating " + getName() + "\n" ) );
48    }
49
50 } // fim da classe ConsumeInteger

```

Fig. 15.14 Classe `ConsumeInteger` representa o consumidor em um relacionamento produtor/consumidor (parte 2 de 2).

Mais uma vez, as principais alterações nesse exemplo estão na definição da classe `HoldIntegerSynchronized` (Fig. 15.15). A classe agora contém seis variáveis de instância – `sharedInt` é um array de inteiros de cinco elementos que é utilizado como *buffer* circular, `writeable` indica se o produtor pode escrever no *buffer* circular, `readable` indica se o consumidor pode ler a partir do *buffer* circular, `readLocation` indica a posição atual a partir da qual o consumidor pode ler o próximo valor, `writeLocation` indica a próxima localização em que o produtor pode colocar um valor e `outputArea` é a `JTextArea` utilizada pelas *threads* deste programa para exibir saída.

```

1 // Fig. 15.15: HoldIntegerSynchronized.java
2 // Definição da classe HoldIntegerSynchronized que usa
3 // sincronização de threads para assegurar que as duas
4 // threads acessem sharedInt nos momentos apropriados.
5
6 // Pacotes do núcleo de Java
7 import java.text.DecimalFormat;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class HoldIntegerSynchronized {
13
14     // array de posições compartilhadas
15     private int sharedInt[] = { -1, -1, -1, -1, -1 };
16
17     // variáveis para manter as informações sobre o buffer
18     private boolean writeable = true;
19     private boolean readable = false;
20     private int readLocation = 0, writeLocation = 0;
21
22     // componente GUI para exibir a saída
23     private JTextArea outputArea;
24
25     // inicializa HoldIntegerSynchronized
26     public HoldIntegerSynchronized( JTextArea output )
27     {
28         outputArea = output;
29     }
30
31     // método sincronizado permite que somente uma thread por
32     // vez invoque este método para colocar um valor em um
33     // objeto HoldIntegerSynchronized em particular
34     public synchronized void setSharedInt( int value )
35     {
36         while ( !writeable ) {
37
38             // a thread que chamou este método deve esperar
39             try {
40
41                 // atualiza o componente GUI Swing
42                 SwingUtilities.invokeLater( new UpdateThread(
43                     outputArea, " WAITING TO PRODUCE " + value ) );
44
45                 wait();
46             }
47
48             // processa InterruptedException enquanto a thread está esperando
49             catch ( InterruptedException exception ) {
50                 System.err.println( exception.toString() );
51             }
52         }
53
54         // coloca valor em writeLocation
55         sharedInt[ writeLocation ] = value;
56
57         // indica que o consumidor pode ler um valor
58         readable = true;
59     }

```

Fig. 15.15 Classe HoldIntegerSynchronized monitora o acesso a um array de inteiros compartilhado (parte 1 de 3).

```

60     // atualiza o componente GUI Swing
61     SwingUtilities.invokeLater( new UpdateThread( outputArea,
62         "\nProduced " + value + " into cell " +
63         writeLocation ) );
64
65     // atualiza writeLocation para operação de escrita futura
66     writeLocation = ( writeLocation + 1 ) % 5;
67
68     // atualiza o componente GUI Swing
69     SwingUtilities.invokeLater( new UpdateThread( outputArea,
70         "\twrite " + writeLocation + "\tread " +
71         readLocation ) );
72
73     displayBuffer( outputArea, sharedInt );
74
75     // testa se o buffer está cheio
76     if ( writeLocation == readLocation ) {
77         writeable = false;
78
79         // atualiza o componente GUI Swing
80         SwingUtilities.invokeLater( new UpdateThread( outputArea,
81             "\nBUFFER FULL" ) );
82     }
83
84     // diz a uma thread que está em espera para ficar pronta
85     notify();
86
87 } // fim do método setSharedInt
88
89 // método sincronizado permite que só uma thread de
90 // cada vez invoque este método para obter um valor de
91 // um objeto HoldIntegerSynchronized em particular
92 public synchronized int getSharedInt()
93 {
94     int value;
95
96     while ( !readable ) {
97
98         // a thread que chamou este método deve esperar
99         try {
100
101             // atualiza o componente Swing da GUI
102             SwingUtilities.invokeLater( new UpdateThread(
103                 outputArea, " WAITING TO CONSUME" ) );
104
105             wait();
106         }
107
108         // processa InterruptedException enquanto a thread estava em espera
109         catch ( InterruptedException exception ) {
110             System.err.println( exception.toString() );
111         }
112     }
113
114     // indica que o produtor pode escrever um valor
115     writeable = true;
116
117     // obtém um valor na posição readLocation corrente
118     value = sharedInt[ readLocation ];

```

Fig. 15.15 Classe `HoldIntegerSynchronized` monitora o acesso a um array de inteiros compartilhado (parte 2 de 3).

```

119      // atualiza o componente Swing da GUI
120      SwingUtilities.invokeLater( new UpdateThread( outputArea,
121          "\nConsumed " + value + " from cell " +
122          readLocation ) );
123
124      // atualiza a posição de leitura para futuras operações de leitura
125      readLocation = ( readLocation + 1 ) % 5;
126
127      // atualiza o componente Swing da GUI
128      SwingUtilities.invokeLater( new UpdateThread( outputArea,
129          "\ntwrite " + writeLocation + "\tread " +
130          readLocation ) );
131
132      displayBuffer( outputArea, sharedInt );
133
134      // testa se o buffer está vazio
135      if ( readLocation == writeLocation ) {
136          readable = false;
137
138          // atualiza o componente Swing da GUI
139          SwingUtilities.invokeLater( new UpdateThread(
140              outputArea, "\nBUFFER EMPTY" ) );
141
142      }
143
144      // diz a uma thread que está em espera para ficar pronta
145      notify();
146
147      return value;
148
149  } // fim do método getSharedInt
150
151  // exibe conteúdo do buffer compartilhado
152  public void displayBuffer( JTextArea outputArea,
153      int buffer[] )
154  {
155      DecimalFormat formatNumber = new DecimalFormat( " #;-#" );
156      StringBuffer outputBuffer = new StringBuffer();
157
158      // coloca elementos do buffer em outputBuffer
159      for ( int count = 0; count < buffer.length; count++ )
160          outputBuffer.append(
161              " " + formatNumber.format( buffer[ count ] ) );
162
163      // atualiza o componente Swing da GUI
164      SwingUtilities.invokeLater( new UpdateThread( outputArea,
165          "\tbuffer: " + outputBuffer ) );
166  }
167
168 } // fim da classe HoldIntegerSynchronized

```

Fig. 15.15 Classe `HoldIntegerSynchronized` monitora o acesso a um *array* de inteiros compartilhado (parte 3 de 3).

O método `setSharedInt` (linhas 34 a 87) realiza as mesmas tarefas que ele fez na Fig. 15.10, com algumas modificações. Quando a execução continua na linha 55 depois do laço `while`, o valor produzido é colocado na posição `writeLocation` do *buffer* circular. Em seguida, `readable` é configurado como `true` (linha 58), porque há pelo menos um valor no *buffer* que o cliente pode ler. As linhas 61 a 63 usam o método `invokeLater` de `SwingUtilities` para anexar o valor produzido e a célula na qual o valor foi colocado à `JTextArea` (o método `run` da classe `UpdateThread` executa a verdadeira operação de anexar). Assim, a linha 66 atualiza `write-`

Location para a próxima chamada a `setSharedInt`. A saída continua com os valores atuais de `writeLocation` e `readLocation` e os valores do `buffer` circular (linhas 69 a 73). Se `writeLocation` for igual a `readLocation`, o `buffer` circular está momentaneamente cheio, depois `writeable` é configurado como `false` (linha 77) e o programa exibe o string `BUFFER FULL` (linhas 80 e 81). Por fim, a linha 85 invoca o método `notify` para indicar que uma *thread* em espera deve mudar para o estado *pronta*.

O método `getSharedInt` (linhas 92 a 149) também executa as mesmas tarefas nesse exemplo que ele fez na Fig. 15.10, com algumas pequenas modificações. Quando a execução continua na linha 115 depois do laço `while`, `writeable` é configurado como `true`, porque existirá pelo menos uma posição livre no `buffer` na qual o produtor pode colocar um valor. Em seguida, a linha 118 atribui a `value` o valor que está na posição `readLocation` do `buffer` circular. As linhas 121 a 123 anexam à `JTextArea` o valor consumido e a célula da qual o valor foi lido. Então, a linha 126 atualiza `readLocation` para a próxima chamada para o método `getSharedInt`. As linhas 129 a 131 continuam a saída na `JTextArea` com os valores atuais de `writeLocation` e `readLocation` e os valores atuais no `buffer` circular. Se `readLocation` for igual a `writeLocation`, o `buffer` circular está atualmente vazio, então `readable` é configurado como `false` (linha 137) e as linhas 140 e 141 exibem o string `BUFFER EMPTY`. Por fim, a linha 145 invoca o método `notify` para colocar o próxima *thread* em espera no estado *pronta* e a linha 147 devolve o valor recuperado para o método chamador.

Nesta versão do programa, as saídas incluem os valores atuais de `writeLocation` e `readLocation` e o conteúdo atual do `buffer` `sharedInt`. Os elementos do array `sharedInt` foram inicializados com -1 para fins de saída, para que você possa ver cada valor inserido no `buffer`. Observe que, depois que o programa coloca o quinto valor no quinto elemento do `buffer`, o programa insere o sexto valor no início do array – produzindo, assim, o efeito de *buffer circular*. O método `displayBuffer` (linhas 152 a 166) utiliza um objeto `DecimalFormat` para formatar o conteúdo do array `buffer`. O string de controle de formato "#; -#" indica um formato de número positivo e um formato de número negativo – os formatos são separados por um ponto-e-vírgula (;). O formato especifica que os valores positivos devem ser precedidos por um espaço e os valores negativos devem ser precedidos por um sinal de menos.

```

1 // Fig. 15.16: SharedCell.java
2 // Mostra múltiplas threads modificando um objeto compartilhado.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.text.DecimalFormat;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class SharedCell extends JFrame {
13
14     // configura a GUI
15     public SharedCell()
16     {
17         super( "Demonstrating Thread Synchronization" );
18
19         JTextArea outputArea = new JTextArea( 20, 30 );
20         getContentPane().add( new JScrollPane( outputArea ) );
21
22         setSize( 500, 500 );
23         show();
24
25     // configura as threads
26     HoldIntegerSynchronized sharedObject =
27         new HoldIntegerSynchronized( outputArea );
28

```

Fig. 15.16 Threads modificando um array de células compartilhado (parte 1 de 2).

```

29     ProduceInteger producer =
30         new ProduceInteger( sharedObject, outputArea );
31
32     ConsumeInteger consumer =
33         new ConsumeInteger( sharedObject, outputArea );
34
35     // inicia as threads
36     producer.start();
37     consumer.start();
38 }
39
40 // executa o aplicativo
41 public static void main( String args[] )
42 {
43     SharedCell application = new SharedCell();
44
45     application.setDefaultCloseOperation(
46         JFrame.EXIT_ON_CLOSE );
47 }
48
49 } // fim da classe SharedCell

```

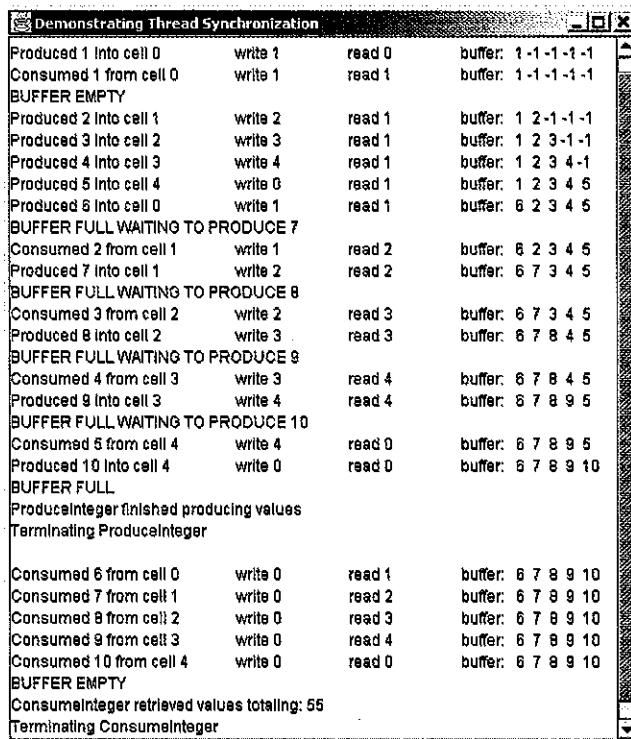


Fig. 15.16 Threads modificando um array de células compartilhado (parte 2 de 2).

15.9 Threads daemon

A *thread daemon* é aquela executada para o benefício de outras *threads*. Ao contrário das *threads* convencionais do usuário (isto é, qualquer *thread* não-*daemon* em um programa), as *threads* *daemon* não impedem que um programa termine. O coletor de lixo é uma *thread* *daemon*. As *threads* não-*daemon* são *threads* de usuário convencionais ou

threads *thread* que despacham eventos e que são usadas para processar eventos da GUI. Designamos uma *thread* como *daemon* com a chamada de método

```
setDaemon( true );
```

O argumento **false** significa que a *thread* não é *daemon*. O programa pode incluir uma mistura de *threads* *daemon* e não-*daemon*. Quando somente *threads* *daemon* permanecem em um programa, o programa encerra. Se a *thread* for *daemon*, ela deve ser configurada como tal antes que seu método **start** seja chamado, ou uma **IllegalThreadStateException** é disparada. O método **isDaemon** devolve **true** se a *thread* for *daemon* e **false**, caso contrário.



Erro comum de programação 15.4

Iniciar uma *thread* e depois tentar torná-la uma *daemon* causa uma **IllegalThreadStateException**.



Observação de engenharia de software 15.3

A *thread* de despacho de eventos é um laço infinito e não é uma *thread* *daemon*. Como tal, a *thread* de despacho de eventos não irá ser terminada, em um aplicativo com janelas, até que o aplicativo chame o método **exit** de **System**.



Boa prática de programação 15.1

Não atribua tarefas críticas a uma *thread* *daemon*. Elas são terminadas sem aviso, o que pode impedir que essas tarefas sejam completadas apropriadamente.

15.10 A interface Runnable

Até agora, estendemos a classe **Thread** para criar novas classes que suportam *multithreading*. Também sobrescrevemos o método **run** para especificar as tarefas que serão realizadas simultaneamente. Entretanto, se queremos suporte a *multithreading* em uma classe que já é derivada de uma classe diferente de **Thread**, devemos implementar a interface **Runnable** nessa classe, porque Java não permite que uma classe estenda mais de uma classe ao mesmo tempo. A própria classe **Thread** implementa a interface **Runnable** (pacote **java.lang**), como expresso no cabeçalho da classe

```
public class Thread extends Object implements Runnable
```

Implementar a interface **Runnable** em uma classe permite que um programa manipule objetos daquela classe como objetos **Runnable**. Assim como ocorre com a derivação da classe **Thread**, o código que controla a *thread* é colocado no método **run**.

O programa que usa o objeto **Runnable** cria um objeto **Thread** e associa o objeto **Runnable** com aquela **Thread**. A classe **Thread** oferece quatro construtores que podem receber referências para objetos **Runnable** como argumentos. Por exemplo, o construtor

```
public Thread( Runnable runnableObject )
```

registra o método **run** do **runnableObject** como o método que será invocado quando a *thread* começar a ser executada. O construtor

```
public Thread( Runnable runnableObject, String threadName )
```

constrói uma *thread* com o nome **threadName** e registra o método **run** de seu argumento **runnableObject** como o método que será invocado quando a *thread* começar a ser executada. Como sempre, o método **start** do objeto **Thread** deve ser chamado para começar a execução da *thread*.

A Fig. 15.7 demonstra um *applet* com uma classe interna **private** e uma classe interna anônima, e cada uma delas implementa a interface **Runnable**. O exemplo também demonstra como suspender uma *thread* (isto é, impedi-la temporariamente de ser executada), como retomar uma *thread* suspensa e como terminar uma *thread* que é executada até que uma condição se torne falsa. Cada uma dessas técnicas é importante porque os métodos **suspend**, **resume** e **stop** de **Thread** tornaram-se obsoletos (isto é, eles não devem mais ser usados em programas Java) com a introdução da plataforma Java 2. Como estes métodos são obsoletos, precisamos codificar nossos próprios mecanismos para suspender, retomar e parar *threads*. Como iremos demonstrar, estes mecanismos se baseiam em blocos de código **synchronized**, laços e variáveis indicadoras **boolean**.

A classe de *applet* RandomCharacters exibe três **JLabels** e três **JCheckboxes**. A *thread* de execução separada está associada com cada par de **JLabel** e botão. Cada *thread* exibe letras do alfabeto aleatoriamente em seu objeto **JLabel** correspondente. O *applet* define o **String alphabet** (linha 16) que contém as letras de A a Z. Esse *string* é compartilhado entre as três *threads*. O método **start** do *applet* (linhas 52 a 65) instancia três objetos **Thread** (linhas 59 e 60) e inicializa cada um com uma instância da classe **RunnableObject**, que implementa a interface **Runnable**. A linha 63 invoca o método **start** da classe **Thread** sobre cada **Thread**, colocando as *threads* no estado *pronta*.

A classe **RunnableObject** é definida nas linhas 115 a 185. O método **run** (linha 120) define duas variáveis locais. A linha 123 utiliza o método **static currentThread** da classe **Thread** para determinar o objeto **Thread** que está sendo executado atualmente. A linha 125 chama o método utilitário **getIndex** do *applet* (definido nas linhas 68 a 76) para determinar o índice, no array **threads**, da *thread* que está sendo executada atualmente. A *thread* atual exibe um caractere aleatório no objeto **JLabel** com o mesmo **index** no array **outputs**.

```

1 // Fig. 15.17: RandomCharacters.java
2 // Demonstrando a interface Runnable.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class RandomCharacters extends JApplet
12     implements ActionListener {
13
14     // declara as variáveis usadas pelo applet
15     // e pela classe interna RunnableObject
16     private String alphabet = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
17     private final static int SIZE = 3;
18
19     private JLabel outputs[];
20     private JCheckBox checkboxes[];
21
22     private Thread threads[];
23     private boolean suspended[];
24
25     // configura a GUI e os arrays
26     public void init()
27     {
28         outputs = new JLabel[ SIZE ];
29         checkboxes = new JCheckBox[ SIZE ];
30         threads = new Thread[ SIZE ];
31         suspended = new boolean[ SIZE ];
32
33         Container container = getContentPane();
34         container.setLayout( new GridLayout( SIZE, 2, 5, 5 ) );
35
36         // cria componentes da GUI, registra ouvintes
37         // e anexa os componentes ao painel de conteúdo
38         for ( int count = 0; count < SIZE; count++ ) {
39             outputs[ count ] = new JLabel();
40             outputs[ count ].setBackground( Color.green );
41             outputs[ count ].setOpaque( true );
42             container.add( outputs[ count ] );
43

```

Fig. 15.17 Demonstrando a interface **Runnable**, suspendendo *threads* e retomando *threads* (parte 1 de 4).

```

44     checkboxes[ count ] = new JCheckBox( "Suspended" );
45     checkboxes[ count ].addActionListener( this );
46     container.add( checkboxes[ count ] );
47   }
48 }
49
50 // Cria e inicia threads. Este método é chamado depois de init e quando
51 // o usuário visita novamente uma página da Web que contém este applet.
52 public void start()
53 {
54   // cria threads e inicia todas as vezes que start é chamado
55   for ( int count = 0; count < threads.length; count++ ) {
56
57     // cria Thread e a inicializa com o
58     // objeto que implementa Runnable
59     threads[ count ] = new Thread( new RunnableObject(),
60       "Thread " + ( count + 1 ) );
61
62     // começa a executar Thread
63     threads[ count ].start();
64   }
65 }
66
67 // determina a posição da thread no array de threads
68 private int getIndex( Thread current )
69 {
70   for ( int count = 0; count < threads.length; count++ )
71
72     if ( current == threads[ count ] )
73       return count;
74
75   return -1;
76 }
77
78 // chamado quando o usuário muda de página da Web; faz parar todas as threads
79 public synchronized void stop()
80 {
81   // Indica que cada thread deve terminar. Configurar
82   // estas referências como null faz com que o método
83   // run de cada thread complete a execução.
84   for ( int count = 0; count < threads.length; count++ )
85     threads[ count ] = null;
86
87   // torna todas as threads prontas para execução,
88   // de modo que elas possam terminar a si mesmas
89   notifyAll();
90 }
91
92 // trata eventos de botão
93 public synchronized void actionPerformed( ActionEvent event )
94 {
95   for ( int count = 0; count < checkboxes.length; count++ ) {
96
97     if ( event.getSource() == checkboxes[ count ] ) {
98       suspended[ count ] = !suspended[ count ];
99
100      // muda a cor do rótulo quando suspende/retoma

```

Fig. 15.17 Demonstrando a interface Runnable, suspensendo threads e retomando threads (parte 2 de 4).

```

101         outputs[ count ].setBackground(
102             !suspended[ count ] ? Color.green : Color.red );
103
104         // se a thread foi retomada, assegura que ela comece a ser executada
105         if ( !suspended[ count ] )
106             notifyAll();
107
108         return;
109     }
110 }
111 }
112
113 // classe interna privativa que implementa Runnable
114 // para que os objetos desta classe possam controlar threads
115 private class RunnableObject implements Runnable {
116
117     // Coloca caracteres na GUI aleatoriamente. As variáveis locais
118     // currentThread e index são declaradas final para que
119     // possam ser utilizadas em uma classe interna anônima.
120     public void run()
121     {
122         // obtém referência para a thread em execução
123         final Thread currentThread = Thread.currentThread();
124
125         // determina a posição da thread no array
126         final int index = getIndex( currentThread );
127
128         // condição do laço determina quando a thread deve parar
129         while ( threads[ index ] == currentThread ) {
130
131             // adormece de 0 a 1 segundo
132             try {
133                 Thread.sleep( ( int )( Math.random() * 1000 ) );
134
135                 // Determina se a thread deve ter sua execução
136                 // suspensa. Usa o applet como monitor.
137                 synchronized( RandomCharacters.this ) {
138
139                     while ( suspended[ index ] &&
140                         threads[ index ] == currentThread ) {
141
142                         // Suspende temporariamente a execução da thread.
143                         // Usa o applet como monitor.
144                         RandomCharacters.this.wait();
145                     }
146
147                 } // fim do bloco sincronizado
148             }
149
150             // processa InterruptedExceptions enquanto adormecida ou em espera
151             catch ( InterruptedException interruptedException ) {
152                 System.err.println( "sleep interrupted" );
153             }
154
155             // exibe caractere no rótulo correspondente
156             SwingUtilities.invokeLater(
157
158                 // classe interna anônima usada pelo método

```

Fig. 15.17 Demonstrando a interface `Runnable`, suspendendo `threads` e retomando `threads` (parte 3 de 4).

```

159      // invokeLater de SwingUtilities para assegurar
160      // que a GUI seja atualizada apropriadamente
161      new Runnable() {
162
163          // atualiza o componente GUI Swing
164          public void run() {
165              {
166                  // obtém um caractere aleatoriamente
167                  char displayChar = alphabet.charAt(
168                      ( int ) ( Math.random() * 26 ) );
169
170                  outputs[ index ].setText(
171                      currentThread.getName() + ": " +
172                      displayChar );
173              }
174
175          } // fim da classe interna anônima
176
177      }; // fim da chamada para SwingUtilities.invokeLater
178
179  } // fim do while
180
181  System.err.println(
182      currentThread.getName() + " terminating" );
183
184
185 } // fim da classe interna privativa RunnableObject
186
187 } // fim da classe RandomCharacters

```

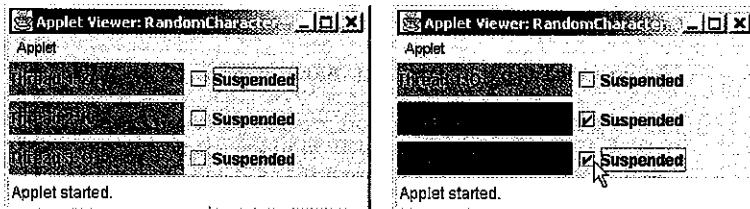


Fig. 15.17 Demonstrando a interface `Runnable`, suspendendo *threads* e retomando *threads* (parte 4 de 4).

O laço `while` nas linhas 129 a 179 continua a ser executado enquanto a referência a `Thread` especificada for igual à referência à `thread` que está atualmente sendo executada (`currentThread`). A cada iteração do laço, a `thread` adormece por um intervalo aleatório de 0 a 1 segundo.

Quando o usuário clica na `JCheckBox` à direita de um `JLabel` em particular, a `Thread` correspondente deve ser *suspensa* (impedida temporariamente de executar) ou *retomada* (permitida que continue a execução). Em versões anteriores de Java, forneciam-se os métodos `suspend` e `resume` da classe `Thread` para suspender e retomar a execução de uma `thread`. Esses métodos são agora obsoletos (isto é, eles não devem mais ser utilizados) porque introduzem a possibilidade de *deadlock* em um programa se eles não forem utilizados corretamente. Pode-se implementar a suspensão e a retomada de uma `thread` utilizando a sincronização de `threads` e os métodos `wait` e `notify` da classe `Object`. As linhas 137 a 147 definem um bloco de código `synchronized` (também chamado de instrução `synchronized`) que ajuda a suspender a `Thread` atualmente em execução. Quando a `Thread` alcança o bloco `synchronized`, o objeto `applet` (mencionado por `RandomCharacters.this`) é bloqueado e a estrutura `while` testa `suspended[index]` para determinar se a `Thread` deve ser suspensa (isto é, `true`). Se for o caso, a linha 144 invoca o método `wait` sobre o objeto `applet` para colocar a `Thread` no estado de *espera*. [Note o uso de `RandomCharacters.this` para acessar a referência `this` da classe do `applet` a partir da classe interna privativa `RunnableObject`.] Quando a `Thread` deve retomar, o programa diz a todas as

threads de espera para se tornarem prontas para execução (discutiremos isto em breve). Entretanto, somente a *thread* retomada terá chance de ser executada. As outras *threads* suspensas voltarão ao estado de espera. As linhas 156 e 157 usam o método `invokeLater` de `SwingUtilities` para atualizar o `JLabel` para a *thread* apropriada.

Observação de engenharia de software 15.4



Uma classe interna pode se referir à referência `this` de sua classe externa precedendo a referência `this` com o nome da classe externa e um operador ponto.

Se o usuário clica na caixa de marcação `Suspended` ao lado de um `JLabel` particular, o programa invoca o método `actionPerformed` (linhas 93 a 111). O método determina qual caixa de marcação recebeu o evento. Utilizando o índice dessa caixa de marcação no *array outputs*, a linha 98 complementa o `boolean` correspondente no *array suspended*. As linhas 101 e 102 configuram a cor de fundo do `JLabel` como vermelho se a *thread* estiver sendo suspensa e como verde se a *thread* estiver sendo retomada. Se a variável `boolean` apropriada for `false`, o programa chama o método `notify` (linha 106) para passar *todas as threads* em espera para o estado *pronta* e prepará-las para retomar a execução. Quando cada *thread* é despachada para que o processador retome a execução, a condição `while` nas linhas 139 e 140 no método `run` falha para a *thread* retomada e o laço termina. A execução do método `run` então continua a partir da linha 156. Para quaisquer outras *threads* que se tornaram prontas, mas ainda estão suspensas, a condição nas linhas 139 e 140 permanece verdadeira e as *threads* voltam ao estado de espera.

O método `stop` do *applet* (linhas 79 a 90) é fornecido para parar todas as três *threads* se o usuário deixa a página da Web em que esse *applet* reside (você pode simular isso selecionando **Stop** no menu **Applet** do *applet-viewer*). O laço `for` na linhas 84 e 85 configura cada referência `Thread` no *array threads* como `null`. A linha 89 invoca o método `notifyAll` de `Object` para assegurar que todas as *threads* em espera fiquem prontas para execução. Quando o programa encontra a condição do laço `while` na linha 129 para cada *thread*, a condição faila e o método `run` termina. Portanto, cada *thread* morre. Se o usuário retorna à página da Web, o contêiner de *applets* chama o método `start` do *applet* para instanciar e iniciar três novas *threads*.

Dica de desempenho 15.5



Parar threads de applet ao deixar uma página da Web é uma prática de programação polida porque impede que seu applet utilize tempo de processador (o que pode reduzir o desempenho) na máquina do navegador quando o applet não está sendo visualizado. As threads podem ser reiniciadas a partir do método `start` do applet, que é invocado pelo navegador quando a página da Web é visitada novamente pelo usuário.

15.11 Grupos de *threads*

Às vezes é útil identificar várias *threads* como pertencentes a um *grupo de threads*; a classe `ThreadGroup` contém métodos para criar e manipular grupos de *threads*. Durante a construção, o grupo recebe um nome único através de um argumento `String`.

As *threads* em um grupo podem ser tratadas como grupo. Pode ser desejável, por exemplo, interromper (com `interrupt`) todas as *threads* em um grupo. O grupo de *threads* pode ser o *grupo de threads-pai* para um *grupo de threads-filhas*. As chamadas de método enviadas para um grupo de *threads-pai* também são enviadas para todas as *threads* nos grupos de *threads-filhas* desse pai.

A classe `ThreadGroup` fornece dois construtores. O construtor

```
public ThreadGroup( String stringName )
constrói um ThreadGroup com o nome stringName. O construtor
```

```
public ThreadGroup( ThreadGroup parentThreadGroup,
                    String stringName )
```

constrói um `ThreadGroup` filho do `parentThreadGroup` denominado `stringName`.

A classe `Thread` fornece três construtores que permitem instanciar uma `Thread` e associá-la a um `ThreadGroup`. O construtor

```
public Thread( ThreadGroup threadGroup, String stringName )
constrói uma Thread que pertence ao threadGroup e tem o nome stringName. Esse construtor normalmente é invocado para as classes derivadas de Thread cujos objetos devem ser associados com um threadGroup.
```

O construtor

```
public Thread( ThreadGroup threadGroup,
    Runnable runnableObject )
```

constrói uma **Thread** que pertence ao **threadGroup** e que invoca o método **run** de **runnableObject** quando a **thread** recebe um processador para iniciar a execução.

O construtor

```
public Thread( ThreadGroup threadGroup,
    Runnable runnableObject, String stringName )
```

constrói uma **Thread** que pertence ao **threadGroup** e que invoca o método **run** de **runnableObject** quando a **thread** recebe um processador para iniciar a execução. O nome dessa **Thread** é indicado por **stringName**.

A classe **ThreadGroup** contém muitos métodos para processar grupos de **threads**. Alguns desses métodos estão resumidos aqui. Para obter mais informações sobre esses métodos, veja a documentação da Java API.

1. O método **activeCount** informa o número de **threads** ativas em um grupo de **threads**, mas o número de **threads** ativas em todos os seus grupos de **threads**-filhas.
2. O método **enumerate** tem quatro versões. Duas versões copiam para um *array* de referências **Thread** as **threads** ativas no **ThreadGroup** (uma dessas também permite que você obtenha recursivamente cópias de todas as **threads** ativas no **ThreadGroup** filho). Duas versões copiam para um *array* de referências **ThreadGroup** os grupos de **threads**-filhas ativas no **ThreadGroup** (uma delas também permite que você obtenha recursivamente cópias de todos os grupos de **threads** ativos em todos os **ThreadGroups** filhos).
3. O método **getMaxPriority** devolve a prioridade máxima de um **ThreadGroup**. O método **setMaxPriority** configura uma nova prioridade máxima para um **ThreadGroup**.
4. O método **getName** devolve o nome do **ThreadGroup** como **String**.
5. O método **getParent** determina o pai de um grupo de **threads**.
6. O método **parentOf** devolve **true** se o **ThreadGroup** para o qual a mensagem é enviada for o pai do, ou o próprio, **ThreadGroup** fornecido como argumento e devolve **false** caso contrário.



Dica de teste e de depuração 15.5

*O método **list** lista o **ThreadGroup**. Isso pode ajudar na depuração.*

15.12 (Estudo de caso opcional) Pensando em objetos: *multithreading*

Os objetos do mundo real executam suas operações independentemente uns dos outros e simultaneamente (em paralelo). Como você aprendeu neste capítulo, Java é uma linguagem de programação com suporte a *multithreading* que facilita a implementação de atividades simultâneas. A UML também contém suporte para projetar modelos simultâneos, como veremos em breve. Nesta seção, discutimos como nossa simulação se beneficia de *multithreading*.

Na Seção 10.25 de “Pensando em objetos”, encontramos um problema com o diagrama de colaborações (Fig. 10.25 – o **waitingPassenger** (a **Person** esperando para andar no **Elevator**) sempre entra no **Elevator** antes que o **ridingPassenger** (a **Person** que está andando no **Elevator**) saia. O uso apropriado de *multithreading* em Java impede este problema, garantindo que o **waitingPassenger** deva esperar que o **ridingPassenger** saia do **Elevator** – como aconteceria na vida real. Em nosso diagrama de colaborações, os objetos passam mensagens para outros objetos chamando métodos destes outros objetos – um objeto remetente pode continuar só depois que um método retorna o controle para ele. Java se refere a uma passagem de mensagem como *chamada síncrona*. Entretanto, a chamada síncrona não é *sincronizada*, porque diversos objetos podem acessar o método ao mesmo tempo – a chamada síncrona não consegue garantir exclusividade sobre um objeto. Isto traz um problema quando modelamos nossa simulação de elevador, porque, de acordo com a Fig. 10.25, o **waitingPassenger** e o **ridingPassenger** podem ocupar o **Elevator** ao mesmo tempo, o que viola o requisito de “capacidade para um” especificado na definição do problema. Neste seção, usamos um método **synchronized** para garantir que somente uma **Person** possa ocupar o **Elevator** de cada vez.

Threads, classes ativas e métodos sincronizados

Java usa *threads* – fluxos de controle do programa independentes de outros fluxos – para representar atividades independentes simultâneas. A UML oferece a noção de *classe ativa* para representar uma *thread*. As classes **Elevator** e **Person** são classes (*threads*) ativas porque seus objetos devem ser capazes de operar simultaneamente e independentemente uns dos outros e de outros objetos no sistema. Por exemplo, o **Elevator** deve ser capaz de se mover entre **Floors** enquanto a **Person** está caminhando num **Floor**. A Fig. 15.18 atualiza o diagrama de colaborações da Fig. 10.25 para suportar classes ativas, as quais são indicadas por uma borda preta espessa no diagrama. Para assegurar que as **Persons** entrem no **Elevator** e saiam dele na ordem apropriada, precisamos observar uma ordem específica no envio de mensagens – o **Elevator** deve enviar a mensagem 3.3.1 (**ridingPassenger** sai) antes de enviar a mensagem 3.2.1.1 (**waitingPassenger** entra no **Elevator**). A UML oferece uma notação para permitir a sincronização em diagramas de colaborações – se temos duas mensagens A e B, a notação B/A indica que a mensagem A precisa esperar que a mensagem B seja completada antes de a mensagem A

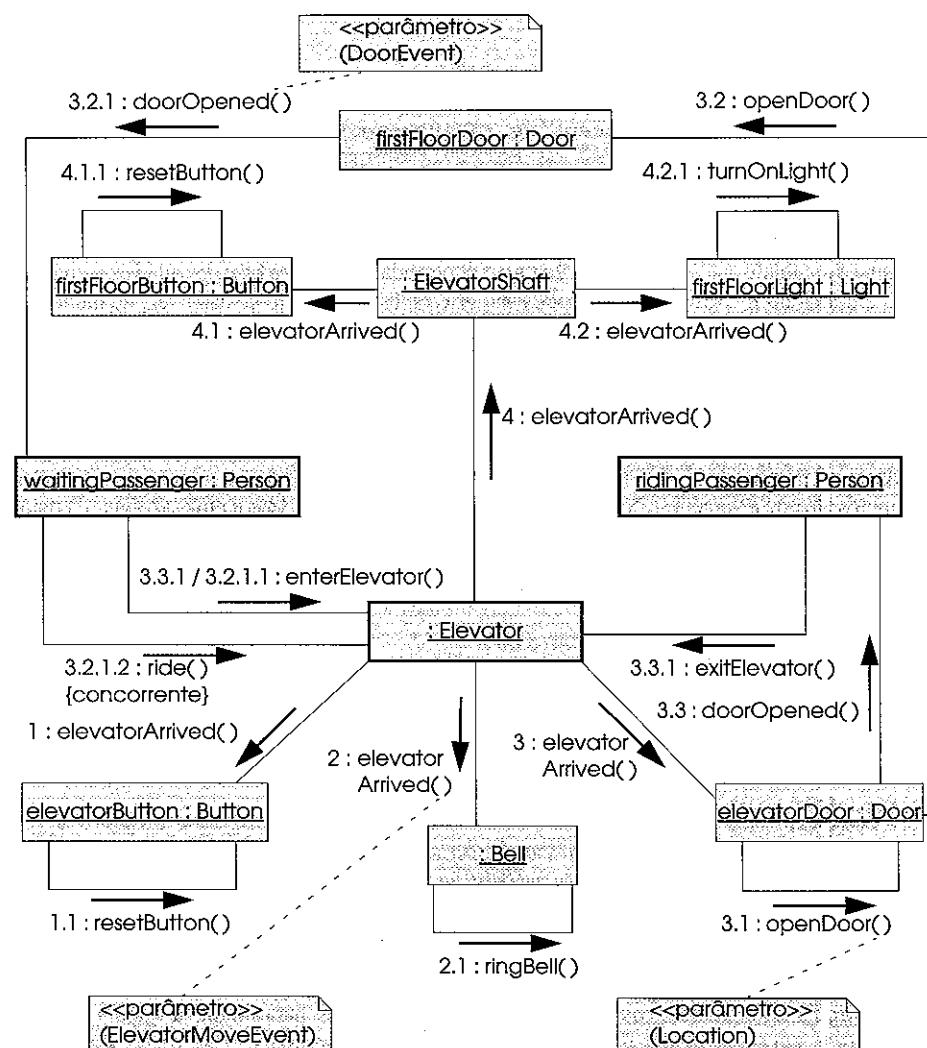


Fig. 15.18 Diagrama de colaborações modificado com as classes ativas para os passageiros que entram no **Elevator** e saem dele.

ocorrer. Por exemplo, a notação 3.3.1/3.2.1.1 antes da mensagem `enterElevator` indica que `waitingPassenger` precisa esperar que o `ridingPassenger` saia do `Elevator` (mensagem 3.3.1) antes de entrar (mensagem 3.2.1.1).

Observação de engenharia de software 15.5



As mensagens em diagramas de colaboração devem ser completadas em ordem (por exemplo, a mensagem 3.1 deve completar antes que seja emitida a mensagem 3.2). Entretanto, as mensagens entre classes ativas podem especificar ordenação diferente conforme necessário para garantir a sincronização entre certas mensagens.

A `Person` deve se sincronizar com o `Elevator` quando andar nele para garantir que só uma `Person` ocupe o `Elevator` de cada vez. Na Fig. 15.18, incluímos a mensagem `ride` (3.2.1.2) para representar a `Person` que está andando no `Elevator` para o outro `Floor`. A palavra-chave `{ simultâneo }` colocada após a mensagem `ride` indica que o método `ride` é `synchronized` quando implementarmos nosso projeto em Java. O `Elevator` contém o método `ride` para que a `Person` permita a sincronização.

```
public synchronized void ride()
{
    try {
        Thread.sleep( maxTravelTime );
    }
    catch ( InterruptedException interruptedException ) {
        // método doorOpened em Person interrompe o método sleep;
        // Person terminou de andar no Elevator
    }
}
```

O método `ride` garante que somente uma `Person` pode andar no `Elevator` de cada vez – como descrito na Seção 15.5, só um método `synchronized` pode estar ativo sobre um objeto de cada vez, de modo que todas as `threads Person` que tentam invocar `ride` devem esperar que a `thread` atual saia do método `ride`.

O método `ride` invoca o método `static sleep` da classe `Thread` para colocar a `thread Person` no estado adormecida, o que representa a `Person` que espera que a viagem seja completada. Precisamos especificar a quantidade máxima de tempo que uma `Person` irá esperar que o `Elevator` complete a viagem – entretanto, tal informação não está especificada na definição do problema. Apresentamos um novo atributo que representa este tempo – `maxTravelTime`, para o qual atribuímos arbitrariamente um valor de 10 minutos (isto é, a `Person` vai esperar 10 minutos até que a viagem se complete). O atributo `maxTravelTime` é uma salvaguarda para o caso do `Elevator` – por qualquer razão – nunca chegar ao outro `Floor`. A `Person` nunca deveria esperar tanto tempo – se esperar 10 minutos, o `Elevator` está quebrado e supomos que nossa `Person` engatinhe para fora do `Elevator` e saia da simulação.

Observação de engenharia de software 15.6



Em um processo de desenvolvimento de software, a fase de análise produz um documento de requisitos (por exemplo, nossa definição do problema). À medida que continuamos a fase de projeto e implementação, descobrimos questões adicionais que não estavam aparentes para nós na fase de análise. Como projetistas, precisamos prever tais questões e lidar com elas adequadamente.

Observação de engenharia de software 15.7



Uma suposição que é falsa é que os requisitos do sistema permanecem estáveis (isto é, fornecem todas as informações necessárias para construir o sistema) ao longo da fase de análise e projeto. Em sistemas grandes, que têm fases longas de implementação, os requisitos podem, e freqüentemente o fazem, mudar para acomodar aquelas questões que não estavam visíveis durante a análise.

Se nosso `Elevator` funciona corretamente, o `Elevator` viaja por cinco segundos – especificamente, invocar o método `sleep` faz a `thread` do `Elevator` parar por cinco segundos, para simular a viagem. Quando a `thread` do `Elevator` acorda, ela envia eventos `elevatorArrived` como descrito na Seção 10.22. A `elevatorDoor` recebe este evento e invoca o método `doorOpened` (mensagem 3.3) do `ridingPassenger`, como em:

```

public void doorOpened( DoorEvent doorEvent )
{
    // coloca Person no Floor em que a Door abriu
    setLocation( doorEvent.getLocation() );
    // interrompe o método sleep de Person no método run
    // e no método ride de Elevator
    interrupt();
}

```

O método `doorOpened` configura a `Location` do `ridingPassenger` para o `Floor` no qual o `Elevator` chegou, depois chama o método `interrupt` da `thread ridingPassenger`. O método `interrupt` termina o método `sleep` invocado no método `ride`, o método `ride` termina e o `ridingPassenger` sai do `Elevator`, e o `ridingPassenger` libera o monitor sobre o objeto `Elevator`, o que permite que o `waitingPassenger` invoque o método `ride` e obtenha o monitor. Agora, o `waitingPassenger` pode invocar o método `ride` para andar no `Elevator`.

O `ridingPassenger` não precisa enviar a mensagem `exitElevator` (mensagem 3.3.1) para o `Elevator`, porque o `waitingPassenger` não pode invocar `ride` até que o `ridingPassenger` libere o monitor do `Elevator` – o `ridingPassenger` libera o monitor quando a `thread ridingPassenger` sai do método `ride` depois de chamar seu método `interrupt`. Portanto, o método `interrupt` da `thread Person` é equivalente ao método `exitElevator` (exceto pelo fato de que a `Person` envia a si mesma a mensagem `interrupt`) e podemos substituir o método `exitElevator` pelo método `interrupt`. Além disso, podemos combinar os métodos `ride` e `enterElevator` para tratar tanto de entrar quanto de andar no `Elevator` – nosso sistema precisa só do método `ride`, que permite a uma `Person` obter um monitor sobre o `Elevator`. À medida que implementarmos nosso modelo, no Apêndice H, usamos nosso diagrama de colaborações para ajudar a gerar código em Java – entretanto, faremos alguns ajustes “específicos para Java” sutis em nosso código, para garantir que as `Persons` entrem no `Elevator` e saiam dele corretamente.

Diagramas de seqüência

Apresentamos agora o outro tipo de diagrama de interação, chamado *diagrama de seqüência*. Como o diagrama de colaborações, o diagrama de seqüência mostra interações entre objetos; entretanto, o diagrama de seqüência enfatiza como as mensagens são enviadas entre objetos *ao longo do tempo*. Os dois diagramas modelam interações em um sistema. Os diagramas de colaboração enfatizam que os objetos interagem no sistema e os diagramas de seqüência enfatizam quando ocorrem estas interações.

A Fig. 15.19 é o diagrama de seqüência para uma `Person` mudando de andar. O retângulo que delimita o nome de um objeto representa aquele objeto. Escrevemos nomes de objetos em diagramas de seqüência usando a mesma convenção que temos usado nos diagramas de colaboração. A linha tracejada para baixo a partir do retângulo de um objeto é a *linha de vida* daquele objeto, que representa a progressão do tempo. As ações ocorrem ao longo da linha de vida de um objeto em ordem cronológica, de cima para baixo – uma ação próxima ao topo de uma linha de vida acontece antes de uma ação próxima à parte de baixo. Fornecemos diversas anotações neste diagrama para esclarecer onde a `Person` sai na simulação. Observe que o diagrama inclui diversas setas tracejadas. Estas setas representam “mensagens de retorno”, ou o retorno do controle para o objeto remetente. Todas as mensagens basicamente produzem uma mensagem de retorno. Mostrar mensagens de retorno não é obrigatório em um diagrama de seqüência – mostramos mensagens de retorno para ficar mais claro.

A passagem de mensagens em diagramas de seqüência é semelhante àquela nos diagramas de colaborações. A seta que sai do objeto que envia a mensagem para o objeto que recebe a mensagem representa uma mensagem entre dois objetos. A ponta da seta aponta para o retângulo na linha de vida do objeto receptor. Como mencionado anteriormente, quando um objeto devolve o controle, uma mensagem de retorno – representada como uma linha tracejada com uma ponta de seta – se estende do objeto que está devolvendo o controle até o objeto que inicialmente enviou a mensagem.

A seqüência na Fig. 15.19 começa quando a `Person` pressiona o `Button` no `Floor` enviando a mensagem `pressButton` para aquele `Button`. O `Button` depois chama o `Elevator` enviando a mensagem `requestElevator` para o `Elevator`.

A **Person** precisa esperar que o **Elevator** processe esta mensagem antes de continuar. Entretanto, a **Person** não precisa esperar pela chegada do **Elevator** antes de prosseguir com outras ações. Em nossa simulação, forçamos a **Person** a esperar, mas poderíamos ter feito a **Person** executar outras ações, como ler jornal, se balançar ou fazer uma ligação com o celular enquanto o **Elevator** vai até o **Floor** da **Person**.

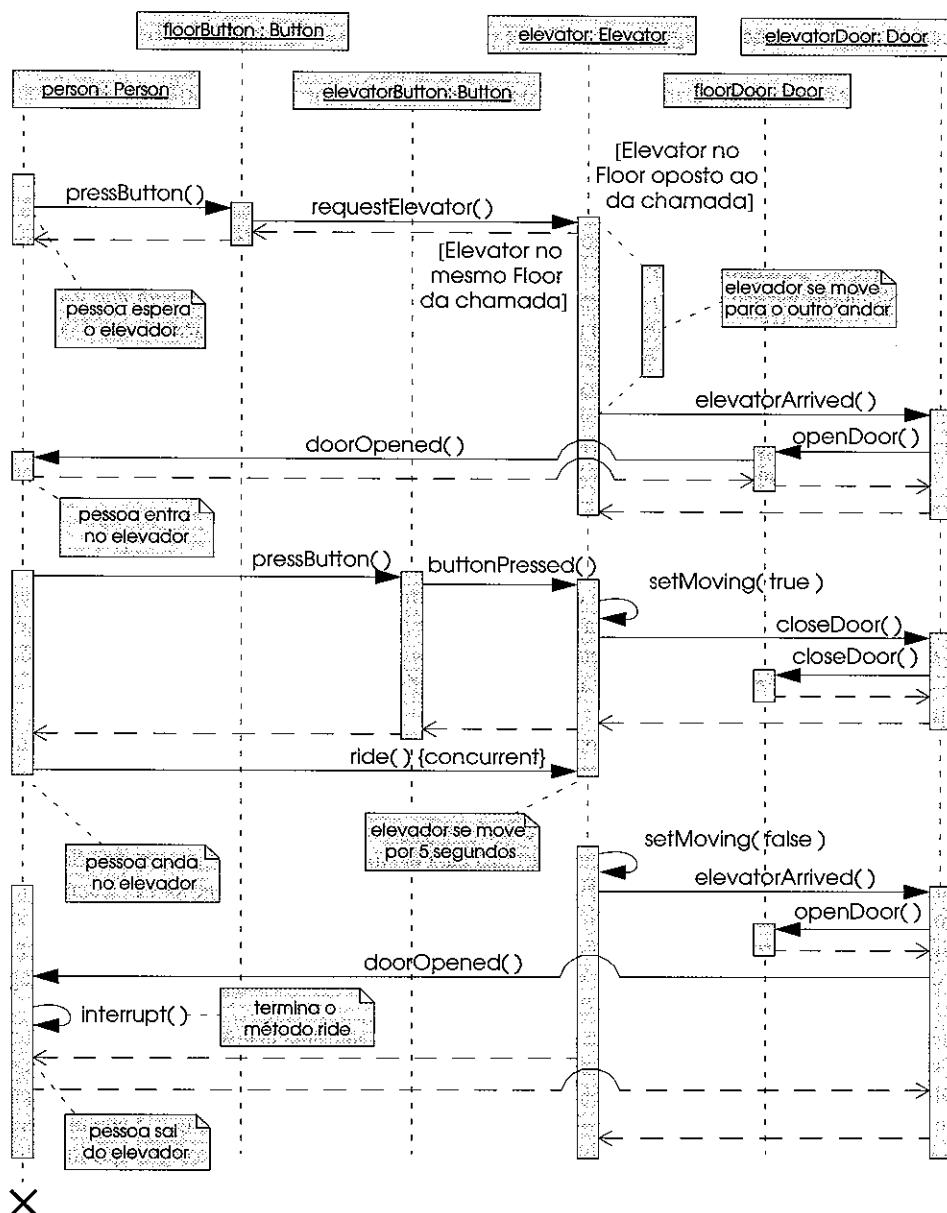


Fig. 15.19 Diagrama de seqüência para uma única **Person** trocando de andar no sistema.

Note a separação no fluxo para o **Elevator** depois de ter sido chamado; o fluxo de execução depende de qual **Floor** gerou a chamada. Se o **Elevator** está em um **Floor** diferente daquele da chamada, ele deve se mover para o **Floor** no qual a **Person** está esperando. Para economizar espaço, a nota informando que o **Elevator** se move para o outro **Floor** representa esta seqüência (iremos construir esta seqüência em seguida, quando discutirmos como a **Person** anda no **Elevator** para o outro **Floor**). Quando o **Elevator** se move para o outro **Floor**, a seqüência alternativa se une com a seqüência original na linha de vida do **Elevator**, e o **Elevator** envia uma mensagem **elevatorArrived** para a **elevatorDoor** ao chegar.

Se o **Elevator** estiver no mesmo **Floor** da chamada, a **elevatorDoor** envia imediatamente uma mensagem **elevatorArrived** para a **elevatorDoor** na chegada. A **elevatorDoor** recebe a mensagem de chegada e abre a **Door** no **Floor** de chegada. Esta porta envia uma mensagem **doorOpened** para a **Person** – aquela **Person** depois entra no **Elevator**.

A **Person** então pressiona o **elevatorButton**, o qual envia um evento **buttonPressed** para o **Elevator**. O **Elevator** fica pronto para sair chamando seu método **private setMoving**, que muda o atributo boolean **moving** do **Elevator** para **true**. O **Elevator** fecha a **elevatorDoor**, que fecha a **Door** naquele **Floor**. A **Person** então anda no **Elevator** invocando o método **synchronized ride** do **Elevator**. Como no diagrama de colaborações, a palavra-chave **{ simultânea }** colocada após o método **ride** indica que o método é **synchronized** ao ser implementado em Java.

O restante do diagrama mostra a seqüência após a chegada do **Elevator** ao **Floor** de destino (também descrita na Fig. 15.19). Na chegada, o **Elevator** pára de se mover chamando o método **private setMoving**, que configura o atributo **moving** para **false**. O **Elevator** então envia uma mensagem **elevatorArrived** para a **elevatorDoor**, que abre a **Door** naquele **Floor** e envia para a **Person** uma mensagem **doorOpened**. O método **doorOpened** acorda a **thread Person** adormecida invocando o método **interrupt**, fazendo o método **ride** terminar e passar a **thread Person** novamente para o estado “pronta”. A **Person** sai da simulação logo em seguida. Note o grande “x” na base da linha de vida da **Person**. Em um diagrama de seqüência, este “x” indica que o objeto associado destrói a si mesmo (em Java, o objeto **Person** é marcado para coleta de lixo).

Nosso diagrama de classes final

A integração de *multithreading* em nosso modelo de elevador conclui o projeto do modelo. Implementamos este modelo no Apêndice H. A Fig. 15.20 apresenta o diagrama de classes completo que usamos ao implementarmos o modelo. Observe que a principal diferença entre os diagramas de classes da Fig. 15.20 e da Fig. 10.30 é a presença de classes ativas na Fig. 15.20. Estabelecemos que as classes **Elevator** e **Person** são classes ativas. Entretanto, a definição do problema mencionou que, “se uma pessoa não entra no elevador nem o chama, o elevador fecha a porta”. Tendo discutido *multithreading*, acreditamos que um requisito melhor seria que as **Doors** se fechassem automaticamente (usando uma **thread**) se elas tiverem ficado abertas por mais do que um breve período (por exemplo, três segundos). Além disso, embora não mencionadas na definição do problema, as **Lights** se desligam só quando o **Elevator** parte de um **Floor**. Marcamos as **Doors** e **Lights** como classes ativas para lidar com esta mudança. Implementamos esta mudança no Apêndice H.

A Fig. 15.21 apresenta os atributos e as operações para todas as classes na Fig. 15.20. Usamos os dois diagramas para implementar o modelo. Omitimos os métodos **enterElevator** e **exitElevator** da classe **Elevator**, porque, como foi discutido no diagrama de interações, o método **ride** aparece para tratar da entrada, da viagem e da saída da **Person** do **Elevator**. Além disso, substituímos o método **departElevator** pelo método **private setMoving**, porque, de acordo com a Fig. 15.19, o método **setMoving** fornece o serviço que permite ao **Elevator** partir de um **Floor**. Também incluímos na classe **Person** o atributo **private maxTravelTime**, que representa o tempo máximo que a **Person** irá esperar para andar no **Elevator**. Atribuímos um valor de 10 minutos a **maxTravelTime** ($10 * 60$ segundos). Iremos usar estes diagramas de classes para implementar nosso código em Java no Apêndice H, mas continuaremos a fazer ajustamentos sutis “específicos de Java” no código. Nos apêndices, para cada classe criamos métodos que acessam referências a objetos e implementamos métodos de interface.

Na Seção 22.9 de “Pensando em objetos”, projetamos a visão – a exibição de um modelo. Quando implementarmos esta exibição no Apêndice I, teremos uma simulação de elevador totalmente funcional, com 3.594 linhas de código.

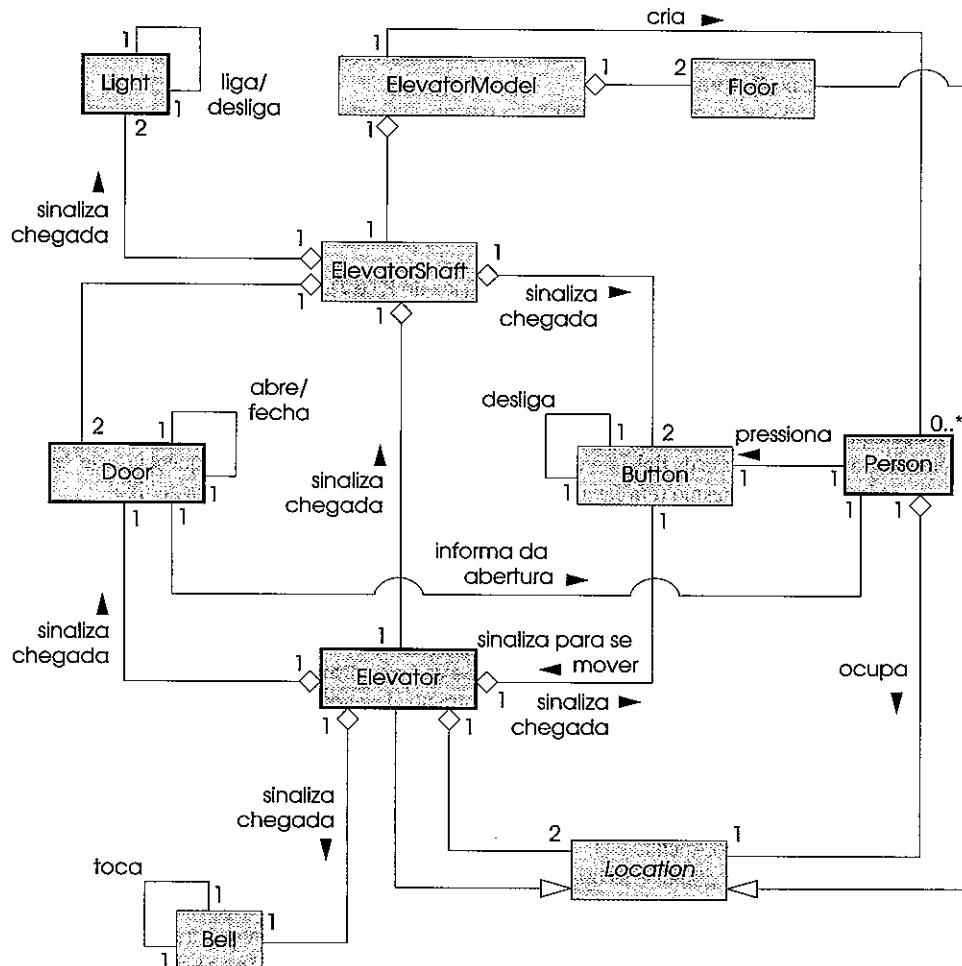


Fig. 15.20 Diagrama de classes final da simulação de elevador.

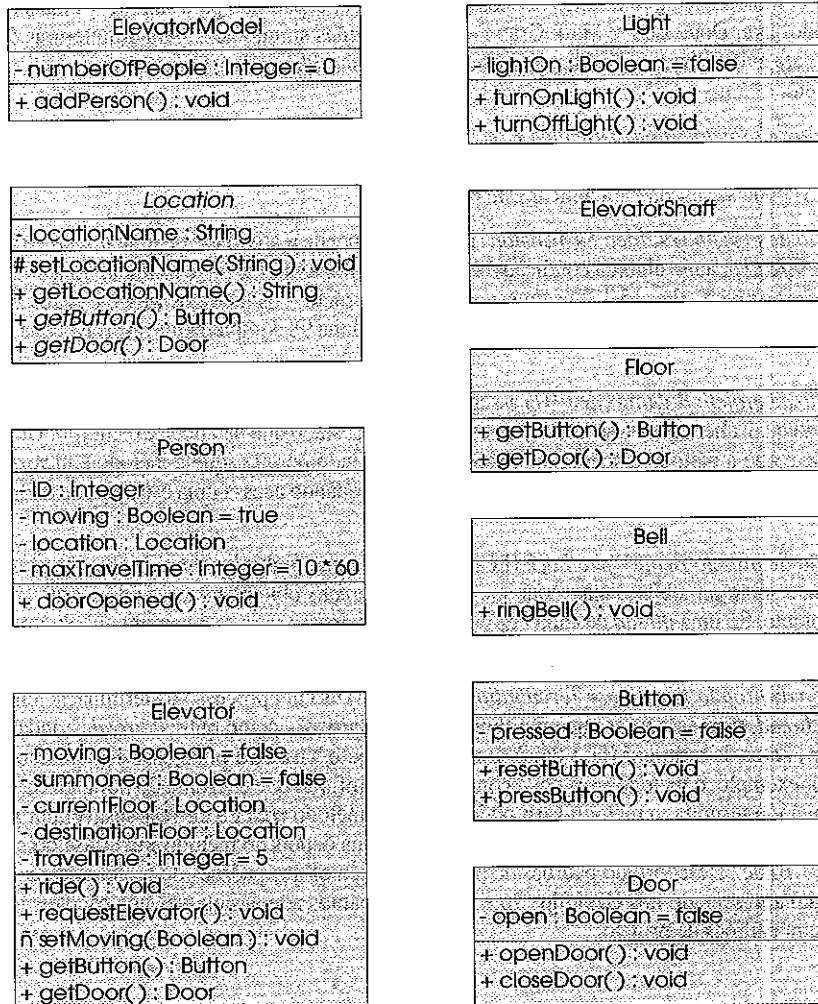


Fig. 15.21 Diagrama de classes final com atributos e operações.

15.13 (Opcional) Descobrindo padrões de projeto: padrões de projeto simultâneos

Criaram-se muitos padrões de projeto adicionais desde a publicação do livro da gangue dos quatro, que apresentou padrões envolvendo sistemas orientados a objetos. Alguns destes novos padrões envolvem sistemas orientados a objetos específicos, como sistemas simultâneos, distribuídos ou paralelos. Nesta seção, discutimos padrões de simultaneidade para concluir nossa discussão de programação com *multithreading*.

Padrões de projeto simultâneos

As linguagens de programação com *multithreading* como Java permitem especificar atividades simultâneas – isto é, aquelas que operam em paralelo umas com as outras. Projetar sistemas simultâneos de forma inadequada pode tra-

zer problemas na simultaneidade. Por exemplo, dois objetos tentando alterar dados compartilhados ao mesmo tempo poderiam corromper os dados. Além disso, se dois objetos esperam um pelo outro para terminar tarefas e se nenhum consegue completar sua tarefa, estes objetos poderiam ficar esperando para sempre – uma situação chamada *deadlock*. Usando Java, Doug Lea¹ e Mark Grand² criaram padrões simultâneos para arquiteturas de projetos com *multithreading* para evitar diversos problemas associados com *multithreading*. Fornecemos uma lista parcial destes padrões de projeto:

- O *padrão de projeto Single-Threaded Execution* (Grand, 98) evita que diversas *threads* invoquem o mesmo método de um outro objeto simultaneamente. Em Java, a palavra-chave **synchronized** (discutida no Capítulo 15) pode ser usada para aplicar este padrão.
- O *padrão de projeto Guarded Suspension* (Lea, 97) suspende a atividade de uma *thread* e retorna a atividade de desta *thread* quando alguma condição é satisfeita. As linhas 137 a 147 e as linhas 95 a 109 da classe **RandomCharacters** (Fig. 15.17) usam este padrão de projeto – os métodos **wait** e **notify** suspendem e retomam, respectivamente, as *threads* do programa, e a linha 98 complementa a variável que a condição avalia.
- O *padrão de projeto Balking* (Lea, 97) assegura que o método irá “empacar” (*balk*) – isto é, retornar sem ter executado quaisquer ações – se um objeto estiver em um estado que não pode executá-lo. Uma variação deste padrão é que o método dispara uma exceção descrevendo porque aquele método é incapaz de ser executado – por exemplo, o método que dispara uma exceção quando acessa uma estrutura de dados que não existe.
- O *padrão de projeto Read/Write Lock* (Lea, 97) permite que múltiplas *threads* obtenham acesso de leitura simultâneo sobre um objeto, mas evita que múltiplas *threads* obtenham acesso de escrita simultâneo sobre aquele objeto. Só uma *thread* de cada vez pode obter um acesso para escrever em um objeto – quando aquela *thread* obtém acesso de escrita, o objeto fica *bloqueado* para outras *threads*.
- O *padrão de projeto Two-Phase Termination* (Grand, 98) usa um processo de terminação em duas fases para uma *thread*, para assegurar que uma *thread* libere recursos – como outras *threads* criadas – na memória (fase um) antes de terminar (fase dois). Em Java, o objeto **Thread** pode usar este padrão para o método **run**. Por exemplo, o método **run** pode conter um laço infinito que é terminado por alguma mudança de estado – quando termina, o método **run** pode invocar o método **private** responsável por parar quaisquer outras *threads* criadas por ele (fase um). A *thread* termina depois do método **run** (fase dois).

Na Seção 17.10 de “Descobrindo padrões de projeto”, voltamos aos padrões de projeto da gangue dos quatro. Usando o material apresentado nos Capítulos 16 e 17, identificamos aquelas classes nos pacotes **java.io** e **java.net** que usam padrões de projeto.

Resumo

- Os computadores realizam operações de forma simultânea, como compilar um programa, imprimir um arquivo e receber mensagens de correio eletrônico através de uma rede.
- As linguagens de programação geralmente fornecem apenas um conjunto simples de estruturas de controle que permitem que os programadores realizem uma ação por vez e, então, prossigam para a próxima ação depois que a anterior é terminada.
- A simultaneidade que os computadores realizam hoje é normalmente implementada como “primitivas” de sistemas operacionais disponíveis somente para os “programadores de sistemas” altamente experientes.
- Java disponibiliza as primitivas de simultaneidade para o programador.
- Os aplicativos contêm *threads* de execução, cada *thread* designando uma parte de um programa que pode ser executada de forma simultânea com outras *threads*. Essa capacidade é chamada de *multithreading*.

¹ D. Lea, *Concurrent Programming in Java, Second Edition: Design Principles and Patterns*. Massachussets: Addison-Wesley. November 1999.

² M. Grand, *Patterns in Java; A Catalog of Reusable Design Patterns Illustrated with UML*. New York: John Wiley and Sons, 1998.

- Java fornece uma *thread* coletor de lixo de baixa prioridade que recupera memória alocada dinamicamente que não é mais necessária. O coletor de lixo é executado quando o tempo de processador está disponível e não há *threads* executáveis de prioridade mais alta. O coletor de lixo é executado imediatamente quando o sistema está sem memória, para tentar recuperar memória.
- O método **run** contém o código que controla a execução de uma *thread*.
- O programa dispara a execução de uma *thread* chamando o método **start**, que, por sua vez, chama o método **run**.
- O método **interrupt** é chamado para interromper uma *thread*.
- O método **isAlive** devolve **true** se **start** tiver sido chamado por uma determinada *thread* e a *thread* não estiver morta (isto é, seu método **stop** não foi chamado e seu método de controle **run** não completou a execução).
- O método **setName** configura o nome da *Thread*. O método **getName** devolve o nome da *Thread*. O método **toString** devolve um **String** que consiste no nome, na prioridade e no grupo da *thread*.
- O método **static currentThread** de *Thread* devolve uma referência à *Thread* que está sendo executada.
- O método **join** espera que a *Thread* sobre a qual **join** foi chamado morra, antes que a *Thread* atual possa prosseguir.
- A espera pode ser perigosa; ela pode levar a dois problemas sérios, denominados *deadlock* e adiamento indefinido; o adiamento indefinido também é chamado de inanição.
- A *thread* que acabou de ser criada está no estado de nascimento. Ela permanece nesse estado até o seu método **start** ser chamado; isso faz com que a *thread* passe para o estado pronta.
- A *thread* pronta de prioridade mais alta entra no estado em execução quando o sistema aloca um processador para ela.
- A *thread* entra no estado morta quando seu método **run** completa ou termina por alguma razão. O sistema, em algum momento, vai se desfazer de uma *thread* morta.
- A *thread* em execução entra no estado bloqueada quando ela emite uma solicitação de entrada/saída. A *thread* bloqueada torna-se pronta quando a E/S que ela está esperando se completa. A *thread* bloqueada não pode utilizar o processador, mesmo se houver algum disponível.
- Quando um método em execução chama **wait**, a *thread* entra em um estado de espera pelo objeto particular em que a *thread* estava em execução. A *thread* no estado de espera por um objeto particular torna-se pronta com uma chamada para **notify** emitida por outra *thread* associada àquele objeto.
- Toda *thread* no estado de espera por um objeto dado torna-se pronta quando se faz uma chamada para **notifyAll** por outra *thread* associada com esse objeto.
- Toda *thread* Java tem uma prioridade no intervalo **Thread.MIN_PRIORITY** (uma constante igual a 1) e **Thread.MAX_PRIORITY** (uma constante igual a 10). Por *default*, cada *thread* recebe prioridade **Thread.NORM_PRIORITY** (uma constante igual a 5).
- Algumas plataformas Java suportam um conceito denominado fracionamento de tempo e algumas não. Sem fracionamento de tempo, as *threads* de prioridade igual são executadas até sua conclusão antes de seus pares obterem uma chance de ser executados. Com fracionamento de tempo, cada *thread* recebe um breve período de tempo do processador, chamado de quantum, durante o qual essa *thread* pode ser executada. Na conclusão do quantum, mesmo se a execução dessa *thread* não terminou, o processador é tirado dessa *thread* e dado à próxima *thread* de prioridade igual, se houver alguma disponível.
- O trabalho do *scheduler* de Java é manter uma *thread* de prioridade mais alta que está sendo executada o tempo todo e, se o fracionamento de tempo estiver disponível, assegurar que várias *threads* de prioridade igualmente alta sejam executadas por um quantum de tempo, no modo de rodízio.
- A prioridade de uma *thread* pode ser ajustada com o método **setPriority**. O método **getPriority** devolve a prioridade da *thread*.
- A *thread* pode chamar o método **yield** para dar a outra *thread* uma chance de ser executada.
- Todo objeto que tenha métodos **synchronized** tem um monitor. O monitor permite que apenas uma *thread* por vez execute o método **synchronized** sobre o objeto.
- A *thread* que está sendo executada sobre um método **synchronized** pode determinar que ela não pode prosseguir; assim, a *thread* chama voluntariamente **wait**. Isso tira a *thread* da disputa pelo processador e da disputa pelo objeto.
- A *thread* que chamou **wait** é acordada por uma *thread* que chama **notify**. O **notify** atua como um sinal, para a *thread* em espera, de que a condição pela qual a *thread* em espera estava esperando está (ou poderia estar) satisfeita agora, de modo que é aceitável que essa *thread* entre novamente no monitor.
- A *thread* *daemon* serve outras *threads*. Quando somente as *threads* *daemon* permanecem em um programa, Java termina. Se a *thread* deve ser *daemon*, ela deve ser configurada como tal antes de seu método **start** ser chamado.

- Para suportar *multithreading* em uma classe derivada de alguma classe diferente de **Thread**, implemente a interface **Runnable** nessa classe.
- A implementação da interface **Runnable** nos dá a capacidade de tratar a nova classe como um objeto **Runnable** (exatamente como herdar de uma classe permite tratar nossa subclasse como um objeto de sua superclasse). Assim como ocorre quando derivamos da classe **Thread**, o código que controla a *thread* é colocado no método **run**.
- A *thread* com uma classe **Runnable** é criada passando-se para o construtor da classe **Thread** uma referência para um objeto da classe que implementa a interface **Runnable**. O construtor **Thread** registra o método **run** do objeto **Runnable** como o método a ser invocado quando a *thread* começar a ser executada.
- A classe **ThreadGroup** contém os métodos para criar e manipular grupos de *threads* relacionadas em um programa.

Terminologia

<i>adiamento indefinido</i>	método getName
<i>bloqueada (estado de uma thread)</i>	método getParent da classe ThreadGroup
<i>bloqueada por E/S</i>	método interrupt
<i>buffer circular</i>	método interrupted
<i>classe Error ThreadDeath</i>	método isAlive
<i>classe InterruptedException</i>	método isDaemon
<i>classe Thread (no pacote <code>java.lang</code>)</i>	método isInterrupted
<i>classe ThreadGroup</i>	método join
<i>coleta de lixo por uma thread de prioridade baixa</i>	método notify
<i>comunicação entre threads</i>	método notifyAll
<i>consumidor</i>	método run
<i>contexto</i>	método setDaemon
<i>contexto de execução</i>	método setName
<i>deadlock</i>	método setPriority
<i>eliminar uma thread</i>	método sleep da classe Thread
<i>escalonamento</i>	método start
<i>escalonamento de prioridade fixa</i>	método synchronized
<i>escalonamento de rodízio</i>	método wait
<i>escalonamento não-preemptivo</i>	método yield
<i>escalonamento preemptivo</i>	MIN_PRIORITY(1)
<i>escalonar uma thread</i>	<i>monitor</i>
<i>espera ocupada</i>	<i>morta (estado de uma thread)</i>
<i>estado adormecido (de uma thread)</i>	<i>multiprocessamento</i>
<i>estado executável (de uma thread)</i>	<i>multithreading</i>
<i>estados de threads</i>	NORM_PRIORITY(5)
<i>exceção ThreadDeath</i>	<i>nova (estado de uma thread)</i>
<i>execução concorrente de threads</i>	<i>objetos compartilhados</i>
<i>executável (estado de thread)</i>	<i>paralelismo</i>
<i>fracionamento de tempo</i>	<i>perda de memória</i>
<i>grupo de threads</i>	<i>prioridade de threads</i>
<i>grupo de threads-filhas</i>	<i>prioridade de uma thread</i>
<i>grupo de threads-pai</i>	<i>programa de uma única thread</i>
<i>herança múltipla</i>	<i>programa multithreaded</i>
<i>herdar a prioridade da thread</i>	<i>quantum</i>
<i>IllegalArgumentException</i>	<i>relacionamento produtor/consumidor</i>
<i>IllegalMonitorStateException</i>	<i>segurança de threads</i>
<i>IllegalThreadStateException</i>	<i>servidor multithreaded</i>
<i>inanição</i>	<i>sincronização</i>
<i>interface Runnable (no pacote <code>java.lang</code>)</i>	<i>sincronização de threads</i>
<i>InterruptedException</i>	string BUFFER FULL/EMPTY
<i>linguagem de uma única thread</i>	<i>thread</i>
MAX_PRIORITY(10)	<i>thread consumidora</i>
<i>método currentThread</i>	<i>thread daemon</i>
<i>método destroy</i>	<i>thread definida pelo programador</i>
<i>método dumpStack</i>	<i>thread executável de prioridade mais alta</i>

thread produtora

`Thread.MAX_PRIORITY`
`Thread.MIN_PRIORITY`
`Thread.NORM_PRIORITY`

`Thread.sleep()`

thread-pai
threads assíncronas
variável de condição

Exercícios de auto-revisão

- 15.1** Preencha as lacunas em cada uma das frases seguintes:
- C e C++ são linguagens de *thread* _____, ao passo que Java é uma linguagem com _____ *threads*.
 - Java fornece uma *thread* _____ que recupera automaticamente a memória alocada dinamicamente.
 - Java elimina a maioria dos erros de _____ que em geral ocorrem em linguagens como C e C++ quando a memória dinamicamente alocada não é explicitamente recuperada pelo programa.
 - Três razões pelas quais uma *thread* que está viva poderia ser não-executável (isto é, bloqueada) são: ela estar _____, estar _____ e estar _____.
 - A *thread* entra no estado morta quando _____.
 - A prioridade da *thread* pode ser alterada com o método _____.
 - A *thread* pode desistir do processador em favor de uma *thread* de mesma prioridade chamando o método _____.
 - Para esperar um número designado de milissegundos e depois retomar a execução, a *thread* deve chamar o método _____.
 - O método _____ move uma *thread* do estado de *espera* do objeto para o estado pronta.
- 15.2** Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- A *thread* não é executável se estiver morta.
 - Em Java, a *thread* executável de prioridade mais alta fará preempção da *thread* de prioridade mais baixa.
 - Os sistemas Java Windows e Windows NT utilizam fracionamento de tempo. Portanto, eles podem permitir às *threads* fazer preempção de *threads* da mesma prioridade.
 - As *threads* podem ceder seu tempo de processamento (`yield`) às *threads* de prioridade mais baixa.

Respostas aos exercícios de auto-revisão

- 15.1** a) única, múltiplas. b) coletor de lixo. c) perda de memória. d) em espera, adormecida, bloqueada para entrada/saída. e) seu método `run` termina. f) `setPriority`. g) `yield`. h) `sleep`. i) `notify`.
- 15.2** a) Verdadeira.
b) Verdadeira.
c) Falsa. O fracionamento de tempo permite que uma *thread* seja executada até que sua fração de tempo (ou quantum) expire. Assim, outras *threads* de prioridade igual podem ser executadas.
d) Falsa. As *threads* só podem ceder seu tempo de processamento a *threads* de igual prioridade.

Exercícios

- 15.3** Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- O método `sleep` não consome tempo de processador enquanto uma *thread* dorme.
 - Declarar um método `synchronized` garante que não ocorra *deadlock*.
 - Java oferece um recurso poderoso denominado herança múltipla.
 - Os métodos `suspend` e `resume` da classe `Thread` são obsoletos.
- 15.4** Defina cada um dos seguintes termos.
- thread*
 - multithreading*
 - estado pronta
 - estado bloqueada
 - escalonamento preemptivo
 - interface `Runnable`
 - monitor

- h) método **notify**
 i) relacionamento produtor/consumidor
- 15.5** a) Liste cada uma das razões declaradas neste capítulo para utilizar *multithreading*.
 b) Liste razões adicionais para utilizar *multithreading*.
- 15.6** Liste cada uma das três razões dadas no texto para entrar no estado bloqueado. Para cada uma delas, descreva como o programa normalmente deixará o estado bloqueado e entrará no estado executável.
- 15.7** Diferencie escalonamento preemptivo de escalonamento não-preemptivo. Qual deles Java utiliza?
- 15.8** O que é fracionamento de tempo? Dê uma diferença fundamental de como o escalonamento é realizado em sistemas Java que suportam fracionamento de tempo em oposição aos sistemas Java que não suportam fracionamento de tempo.
- 15.9** Por que uma *thread* iria alguma vez querer chamar **yield**?
- 15.10** Que aspectos do desenvolvimento de *applets* Java para a World Wide Web incentivam os projetistas de *applet* a utilizar **yield** e **sleep** abundantemente?
- 15.11** Se você quiser escrever seu próprio método **start**, o que você deve se certificar de fazer para se assegurar de que suas *threads* iniciarão adequadamente?
- 15.12** Distinga cada uma das seguintes maneiras de parar *threads* temporariamente:
 a) espera ocupada
 b) adormecimento
 c) bloqueio de E/S
- 15.13** Escreva uma instrução Java que testa se uma *thread* está viva.
- 15.14** a) O que é herança múltipla?
 b) Explique por que Java não oferece herança múltipla.
 c) Que recurso Java oferece em vez da herança múltipla?
 d) Explique a utilização típica desse recurso.
 e) Em que esse recurso difere das classes **abstract**?
- 15.15** Distinga as noções de **extends** e **implements**.
- 15.16** Discuta cada um dos seguintes termos no contexto dos monitores:
 a) monitor
 b) produtor
 c) consumidor
 d) **wait**
 e) **notify**
 f) **InterruptedException**
 g) **synchronized**
- 15.17** (*A Lebre e a Tartaruga*) Nos exercícios do Capítulo 7, pedimos para você simular a lendária corrida da lebre e tartaruga. Implemente uma nova versão dessa simulação, dessa vez colocando cada um dos animais em *threads* separadas. No início da corrida, chame os métodos **start** para cada uma das *threads*. Utilize **wait**, **notify** e **notifyAll** para sincronizar as atividades dos animais.
- 15.18** (*Aplicativos colaboradores multithreaded em Rede*) No Capítulo 21, vamos tratar das redes em Java. Um aplicativo Java *multithreaded* pode comunicar-se simultaneamente com vários computadores *host*. Isso cria a possibilidade de construirmos alguns tipos interessantes de aplicativos colaboradores. Antecipando o estudo de redes no Capítulo 17, desenvolva propostas para vários aplicativos com múltiplas *threads* em rede. Depois de estudar o Capítulo 17, implemente alguns desses aplicativos.
- 15.19** Escreva um programa Java para demonstrar que, quando uma *thread* de alta prioridade é executada, ela retarda a execução de todas as *threads* de prioridade mais baixa.
- 15.20** Se seu sistema suporta fracionamento de tempo, escreva um programa Java que demonstra o fracionamento de tempo entre várias *threads* de igual prioridade. Mostre que a execução de uma *thread* de prioridade mais baixa é adiada pelo fracionamento de tempo das *threads* de prioridade mais alta.
- 15.21** Escreva um programa Java que demonstra uma *thread* de alta prioridade que utiliza **sleep** para dar uma oportunidade para as *threads* de prioridade mais baixa serem executadas.

15.22 Se seu sistema não suporta fracionamento de tempo, escreva um programa Java que demonstra duas *threads* que utilizam `yield` para permitir que as duas sejam executadas.

15.23 Dois problemas que podem ocorrer em sistemas como Java, que permitem que as *threads* esperem, são os *deadlocks*, em que uma ou mais *threads* esperarão eternamente um evento que não pode ocorrer, e o adiamento indefinido, em que uma ou mais *threads* serão retardados por algum tempo imprevisivelmente longo. Dê um exemplo de como cada um desses problemas pode ocorrer em um programa Java com múltiplas *threads*.

15.24 (*Leitores e gravadores*) Este exercício pede para você desenvolver um monitor Java para resolver um problema famoso no controle de simultaneidade. Esse problema foi discutido e resolvido pela primeira vez por P. J. Courtois, F. Heymans e D. L. Parnas no seu trabalho de pesquisa, “Concurrent Control with Readers and Writers”, *Communications of the ACM*, Vol. 14, No. 10, October 1971, pp. 667–668. Um aluno interessado talvez também queira ler o trabalho de pesquisa de C. A. R. Hoare sobre monitores, “Monitors: An Operating System Structuring Concept,” *Communications of the ACM*, Vol. 17, No. 10, October 1974, pp. 549–557. Corrigendum, *Communications of the ACM*, Vol. 18, No. 2, February 1975, p. 95. [O problema dos Leitores e Gravadores é discutido demoradamente no Capítulo 5 do livro do autor: Deitel, H. M., *Operating Systems*, Reading, MA: Addison-Wesley, 1990.]

- a) Com *multithreading*, muitas *threads* podem acessar dados compartilhados; como vimos, o acesso aos dados compartilhados precisa ser cuidadosamente sincronizado para não se corromper os dados.
- b) Imagine um sistema de reservas de passagens aéreas em que muitos clientes estão tentando reservar assentos em vôos específicos entre cidades específicas. Todas as informações sobre vôos e assentos estão armazenadas em um banco de dados comum na memória. O banco de dados consiste em muitas entradas, cada uma representando um assento em um vôo em particular durante um dia em particular entre cidades em particulares. Em um cenário típico de reservas de companhias aéreas, o cliente pesquisará o banco de dados procurando o vôo “ótimo” para atender às suas necessidades. Depois ele pode pesquisar o banco de dados muitas vezes antes de decidir reservar um vôo particular. O assento que estava disponível durante a fase de pesquisa poderia facilmente ser reservado por outra pessoa antes de o cliente ter a chance de reservá-lo depois de se decidir. Nesse caso, quando o cliente tentar fazer a reserva, ele descobrirá que os dados foram alterados e o vôo não está mais disponível.
- c) O cliente que pesquisa o banco de dados é denominado *leitor*. O cliente que tenta reservar o vôo é denominado *gravador*. Evidentemente, um número qualquer de leitores pode estar pesquisando dados compartilhados ao mesmo tempo, mas cada gravador precisa de acesso exclusivo aos dados compartilhados, para evitar que os dados sejam corrompidos.
- d) Escreva um programa Java que dispara múltiplas *threads* de leitor e múltiplas *threads* de gravador, cada uma tentando acessar um único registro de reserva. A *thread* de gravador tem duas transações possíveis, `makeReservation` e `cancelReservation`. A de leitor tem uma transação possível, `queryReservation`.
- e) Primeiro implemente uma versão do programa que permita o acesso não-sincronizado ao registro de reserva. Mostre como a integridade do banco de dados pode ser corrompida. Em seguida, implemente uma versão de seu programa que utiliza sincronização com monitor de Java com `wait` e `notify` para impor um protocolo disciplinado para os leitores e gravadores que acessam os dados de reserva compartilhados. Em particular, o programa deve permitir que vários leitores acessem os dados compartilhados simultaneamente quando nenhum gravador estiver ativo. Mas se houver um gravador ativo, nenhum leitor deve ter permissão de acessar os dados compartilhados.
- f) Tenha cuidado. Esse problema tem muitas sutilezas. Por exemplo, o que acontece quando há vários leitores ativos e um gravador quer gravar? Se permitirmos que um fluxo contínuo de leitores chegue e compartilhe os dados, eles poderiam adiar indefinidamente o gravador (que poderia ficar cansado de esperar e ir fazer negócios em outra parte). Para resolver esse problema, você pode decidir favorecer os gravadores em detrimento dos leitores. Mas aqui também há uma armadilha, uma vez que um fluxo contínuo de gravadores poderia então adiar indefinidamente os leitores em espera e estes talvez escolham fazer seu negócio em outra parte! Implemente o monitor com os seguintes métodos: `startReading`, que é chamado por qualquer leitor que queira começar o acesso a uma reserva, `stopReading`, para ser chamado por qualquer leitor que terminou de ler uma reserva, `startWriting`, para ser chamado por qualquer gravador que queira fazer uma reserva, e `stopWriting`, para ser chamado por qualquer gravador que terminou de fazer uma reserva.

15.25 Escreva um programa que faz uma bola azul pular dentro de um *applet*. A bola deve ser iniciada com um evento `mousePressed`. Quando a bola atingir a borda do *applet*, a bola deve rebater na borda e continuar na direção oposta.

15.26 Modifique o programa do Exercício 15.25 para adicionar uma nova bola toda vez que o usuário clicar o mouse. Permita um mínimo de 20 bolas. Escolha a cor para cada bola nova aleatoriamente.

15.27 Modifique o programa do Exercício 15.26 para adicionar sombras. Enquanto a bola se move, desenhe uma elipse sólida preta na parte inferior do *applet*. Você pode pensar em pôr também um efeito de 3D, aumentando ou diminuindo o tamanho de cada bola quando ela atinge a borda do *applet*.

15.28 Modifique o programa do Exercício 15.25 ou 15.26 para fazer as bolas rebaterem quando colidirem entre si.

16

Arquivos e fluxos

Objetivos

- Ser capaz de criar, ler, gravar e atualizar arquivos.
- Entender a hierarquia de classes de fluxos de Java.
- Ser capaz de utilizar as classes `FileInputStream` e `OutputStream`.
- Ser capaz de utilizar as classes `ObjectInputStream` e `ObjectOutputStream`.
- Ser capaz de utilizar a classe `RandomAccessFile`.
- Ser capaz de utilizar o diálogo `JFileChooser` para acessar arquivos e diretórios.
- Familiarizar-se com o processamento de arquivos de acesso seqüencial e de acesso aleatório.
- Ser capaz de utilizar a classe `File`.

Só posso supor que um documento marcado como “Não Arquivar” está arquivado em um documento marcado como “Não Arquivar”.

Senador Frank Church

Depoimento ao Subcomitê de Inteligência do Senado, 1975

A consciência ... em si não aparece dividida em pedaços [bits]. ... Um “rio” ou um “fluxo” são as metáforas utilizadas para descrevê-la mais naturalmente.*

William James

Li parte dele até o fim.

Samuel Goldwyn

*It is quite a three-pipe problem!***

Sir Arthur Conan Doyle



* N. de R. *Stream*, no original.

**N. R. Tradução literal: “É um problema que dá para resolver com três caximbadas”, aludindo aos *pipes* que são abordados no texto.

Sumário do capítulo

- 16.1 Introdução**
- 16.2 Hierarquia de dados**
- 16.3 Arquivos e fluxos**
- 16.4 Criando um arquivo de acesso sequencial**
- 16.5 Lendo dados de um arquivo de acesso sequencial**
- 16.6 Atualizando arquivos de acesso sequencial**
- 16.7 Arquivos de acesso aleatório**
- 16.8 Criando um arquivo de acesso aleatório**
- 16.9 Gravando dados aleatoriamente em um arquivo de acesso aleatório**
- 16.10 Lendo dados sequencialmente de um arquivo de acesso aleatório**
- 16.11 Exemplo: um programa de processamento de transações**
- 16.12 A classe `File`**

Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão • Exercícios

16.1 Introdução

O armazenamento de dados em variáveis e *arrays* é temporário – os dados se perdem quando uma variável local “sai do escopo” ou quando o programa termina. Os *arquivos* são utilizados para retenção a longo prazo de grandes quantidades de dados, mesmo depois de terminar o programa que criou os dados. Os dados mantidos em arquivos são frequentemente chamados de *dados persistentes*, porque os dados existem por mais tempo que a duração da execução do programa. Os computadores armazenam os arquivos em *dispositivos secundários de armazenamento*, como discos magnéticos, discos ópticos e fitas magnéticas. Neste capítulo, explicamos como os arquivos de dados são criados, atualizados e processados por programas Java. Analisamos tanto os arquivos de “acesso seqüencial” como arquivos de “acesso aleatório” e discutimos aplicações típicas para cada um deles.

O processamento de arquivos é um dos recursos mais importantes que uma linguagem deve ter para suportar os aplicativos comerciais que, em geral, processam quantidades maciças de dados persistentes. Neste capítulo, discutiremos os poderosos e abundantes recursos de processamento de arquivos e de fluxos de entrada/saída de Java. O processamento de arquivos é um subconjunto dos recursos de processamento de fluxos de Java que permite a um programa ler e escrever *bytes* da memória em arquivos e através de conexões por redes. Temos dois objetivos neste capítulo – apresentar o paradigma de processamento de arquivos e fornecer ao leitor recursos suficientes de processamento de fluxos para suportar as características de redes apresentadas no Capítulo 17.



Observação de engenharia de software 16.1

Seria perigoso permitir que os applets que chegam de qualquer lugar da World Wide Web fossem capazes de ler e gravar arquivos no sistema do cliente. Por default, a maioria dos navegadores da Web impede que os applets realizem o processamento de arquivos no sistema do cliente. Portanto, os programas de processamento de arquivos geralmente são implementados como aplicativos Java.

16.2 Hierarquia de dados

Em última instância, o computador processa todos os itens de dados como combinações de zeros e uns, porque é simples e econômico construir dispositivos eletrônicos que podem assumir dois estados estáveis – um estado representa 0 e o outro estado representa 1. É notável que as impressionantes funções realizadas pelos computadores envolvam só as manipulações mais fundamentais de 0s e 1s.

O menor item de dados em um computador pode assumir o valor 0 ou o valor 1. Esse item de dados chama-se *bit* (abreviação de “*binary digit*” – um dígito que pode assumir um de dois valores). Os circuitos de computador realizam várias manipulações simples de *bits*, como examinar o valor de um *bit*, configurar-lhe o valor e invertê-lo (de 1 para 0 ou de 0 para 1).

É incômodo para os programadores trabalhar com dados na forma de baixo nível de *bits*. Em vez disso, os programadores preferem trabalhar com dados em formas como *dígitos decimais* (0 a 9), *letras* (A a Z e a a z) e *símbolos especiais* (isto é, \$, @, %, &, *, (,), –, +, “, ;, ?, / e muitos outros). Os dígitos, as letras e os símbolos especiais são chamados de *caracteres*. O conjunto de todos os caracteres utilizados para escrever programas e representar itens de dados em um computador particular chama-se *conjunto de caracteres* desse computador. Como os computadores podem processar somente 1s e 0s, cada caractere em um conjunto de caracteres do computador é representado como um padrão de 1s e 0s (os caracteres em Java são caracteres *Unicode* compostos de 2 *bytes*). Os *bytes* são compostos por oito *bits*. Os programadores criam programas e dados com caracteres; os computadores manipulam e processam esses caracteres como padrões de *bits*. Veja o Apêndice K para obter mais informações sobre o Único-de.

Assim como os caracteres são compostos por *bits*, os *campos* são compostos de caracteres ou *bytes*. O campo é um grupo de caracteres ou *bytes* que possui um significado. Por exemplo, o campo que consiste em letras minúsculas e maiúsculas pode ser utilizado para representar um nome de pessoa.

Os dados processados por computadores formam uma *hierarquia de dados* em que os itens de dados tornam-se maiores e mais complexos, em termos de estrutura, à medida que progredimos de *bits*, para caracteres, para campos; etc.

Geralmente, vários campos (chamados de variáveis de instância em Java) compõem um *registro* (implementando como uma classe em Java). Em um sistema de folha de pagamento, por exemplo, um registro para um empregado em particular talvez consista nos seguintes campos (possíveis tipos de dados para esses campos são mostrados entre parênteses após cada campo):

- Número de identificação do empregado (**int**)
- Nome (**String**)
- Endereço (**String**)
- Salário/hora (**double**)
- Número de isenções reivindicadas (**int**)
- Rendimentos no ano até a data atual (**int** ou **double**)
- Total de impostos retidos (**int** ou **double**)

Portanto, o registro é um grupo de campos relacionados. No exemplo precedente, cada um dos campos pertence ao mesmo empregado. Naturalmente, uma empresa em particular pode ter muitos empregados e terá um registro de folha de pagamento para cada empregado. O *arquivo* é um grupo de registros relacionados¹. O arquivo de folha de pagamento de uma empresa normalmente contém um registro para cada empregado. Portanto, um arquivo de folha de pagamento de uma empresa pequena talvez contenha apenas 22 registros, enquanto um arquivo de folha de pagamento de uma empresa grande talvez contenha 100.000 registros. Não é incomum uma empresa ter muitos arquivos, que contêm alguns milhões, ou mesmo bilhões, de caracteres de informações. A Fig. 16.1 ilustra a *hierarquia de dados*.

Para facilitar a recuperação de registros específicos de um arquivo, pelo menos um campo em cada registro é escolhido como *chave de registro*. A chave de registro identifica um registro como pertencente a uma pessoa ou entidade em particular que é única dentre todos os outros registros. No registro de folha de pagamento descrito anteriormente, o número de identificação do empregado normalmente seria escolhido como a chave de registro.

Há muitas maneiras de organizar registros em um arquivo. O tipo de organização mais comum é chamado de *arquivo seqüencial*, no qual os registros são geralmente armazenados em ordem pelo campo-chave de registro. Em um arquivo de folha de pagamento, os registros são colocados em ordem pelo número de identificação do empregado. O primeiro registro de empregado no arquivo contém o número mais baixo de identificação de empregado e os registros subsequentes contêm números de identificação de empregado cada vez mais altos.

A maioria das empresas utiliza muitos arquivos diferentes para armazenar dados. Por exemplo, as empresas podem ter arquivos de folha de pagamento, arquivos de contas a receber (listagem de valores devidos por clientes), arquivo de contas a pagar (listagem de valores devidos aos fornecedores), arquivo de inventário (listagem de fatos so-

¹ Generalizando mais, o arquivo pode conter dados arbitrários em formatos arbitrários. Em alguns sistemas operacionais, o arquivo é visto como nada mais do que um conjunto de *bytes*. Em um sistema operacional assim, qualquer organização dos *bytes* em um arquivo (como organizar os dados em registros) é uma visão criada pelo programador de aplicativos.

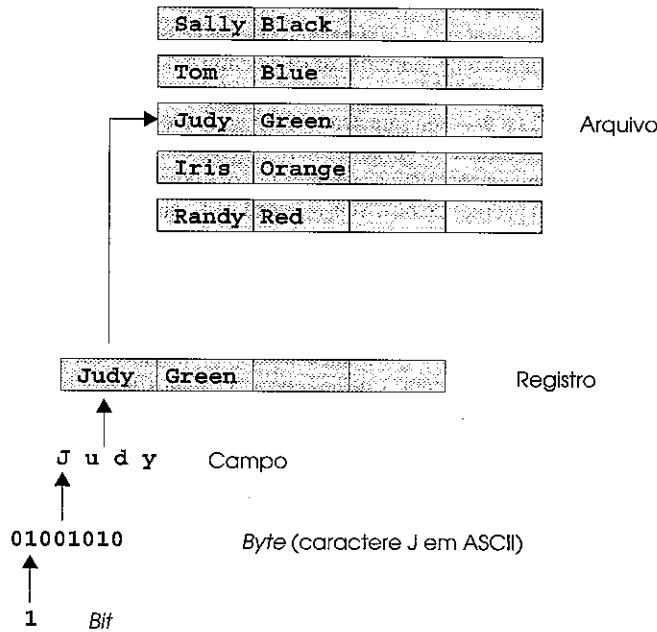


Fig. 16.1 A hierarquia de dados.

bre todos os itens abrangidos pelo negócio) e muitos outros tipos de arquivo. O grupo de arquivos relacionados às vezes se chama *banco de dados*. A coleção de programas projetados para criar e gerenciar os bancos de dados se chama *sistema de gerenciamento de bancos de dados* (DBMS – *database management system*).

16.3 Arquivos e fluxos

Java vê cada arquivo como um *fluxo* seqüencial de *bytes* (Fig. 16.2). Cada arquivo acaba com um *marcador de fim do arquivo* ou em um número específico de *bytes* registrado em uma estrutura administrativa de dados mantida pelo sistema. Java esconde este conceito do programador. Um programa Java que processa um fluxo de *bytes* simplesmente recebe uma indicação do sistema quando o programa alcança o final do fluxo – o programa não precisa saber como a plataforma subjacente representa arquivos ou fluxos. Em alguns casos, a indicação de fim de arquivo ocorre como uma exceção. Em outros casos, a indicação é um valor devolvido por um método invocado sobre um objeto que processa um fluxo. Ambos os casos são demonstrados neste capítulo.

Um programa Java *abre* um arquivo através da criação de um objetos e a associação de um fluxo de *bytes* a este objeto. Java também pode associar fluxos de *bytes* associados com dispositivos. Na verdade, Java cria três objetos de fluxo que são associados com dispositivos quando a execução de um programa Java é iniciada – `System.in`, `System.out` e `System.err`. Os fluxos associados com esses objetos fornecem canais de comunicação entre um programa e um dispositivo particular. Por exemplo, o objeto `System.in` (*objeto de fluxo de entrada padrão*) normalmente permite que um programa insira *bytes* pelo teclado, o objeto `System.out` (*objeto de fluxo de saída padrão*) normalmente permite a um programa gerar como saída dados na tela e o objeto `System.err` (*objeto de fluxo de erro padrão*) permite a um programa gerar como saída mensagens de erro na tela. Cada um desses fluxos pode ser *redirecionado*. Para `System.in` isso permite ao programa ler *bytes* de uma fonte diferente. Para `System.out` e `System.err` isso permite enviar a saída para um destino diferente, como um arquivo em um disco. A classe `System` fornece métodos `setIn`, `setOut` e `setErr` para redirecionar os fluxos de entrada, saída e erro padrão.

Os programas Java fazem o processamento de arquivos usando classes do pacote `java.io`. Esse pacote inclui definições para as classes de fluxo como `FileInputStream` (para entrada baseada em *bytes* de um arquivo), `FileOutputStream` (para saída baseada em *bytes* para um arquivo), `FileReader` (para entrada baseada em carac-

teres de um arquivo) e **FileWriter** (para saída baseada em caracteres para um arquivo). Os arquivos são abertos criando-se objetos dessas classes de fluxo, que herdam das classes **InputStream**, **OutputStream**, **Reader** e **Writer**, respectivamente. Portanto, os métodos dessas classes de fluxo também podem ser aplicados a fluxos de arquivo. Para realizar entrada e saída de tipos de dados, os objetos das classes **ObjectInputStream**, **DataInputStream**, **ObjectOutputStream** e **DataOutputStream** serão utilizados junto com as classes de fluxo de arquivos baseadas em *bytes*, **FileInputStream** e **FileOutputStream**. Os relacionamentos de herança de muitas das classes de E/S de Java estão resumidos na Fig. 16.3 (as classes **abstract** são mostradas em fonte itálico).

Java oferece muitas classes para realizar entrada/saída. Nesta seção, fornecemos uma breve visão geral de cada uma e explicamos como elas se relacionam entre si. No restante do capítulo, usamos várias dessas classes de fluxo à medida que implementamos diversos programas de processamento de arquivos que criam, manipulam e destroem arquivos de acesso sequencial e arquivos de acesso aleatório. Também incluímos um exemplo detalhado da classe **File** que é útil para obter as informações sobre arquivos e diretórios. No Capítulo 17, utilizamos extensamente as classes de fluxo para implementar aplicativos de redes.

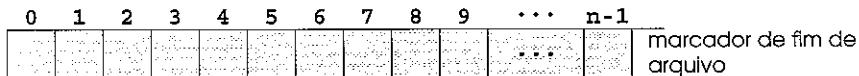


Fig. 16.2 Visão de Java de um arquivo de *n* bytes.

Uma parte da hierarquia de classes do pacote `java.io`

```

java.lang.Object
  File
    FileDescriptor
  InputStream
    ByteArrayInputStream
    FileInputStream
    FilterInputStream
      BufferedInputStream
      DataInputStream
      PushbackInputStream
    ObjectInputStream
    PipedInputStream
    SequenceInputStream
  OutputStream
    ByteArrayOutputStream
    FileOutputStream
    FilterOutputStream
      BufferedOutputStream
      DataOutputStream
      PrintStream
    ObjectOutputStream
  
```

Fig. 16.3 Parte da hierarquia de classes do pacote `java.io` (parte 1 de 2).

Uma parte da hierarquia de classes do pacote `java.io`

```

PipedOutputStream
RandomAccessFile
Reader
    BufferedReader
        LineNumberReader
    CharArrayReader
    FilterReader
        PushbackReader
    InputStreamReader
        FileReader
    PipedReader
    StringReader
Writer
    BufferedWriter
    CharArrayWriter
    FilterWriter
    OutputStreamWriter
        FileWriter
    PipedWriter
    PrintWriter
    StringWriter

```

Fig. 16.3 Parte da hierarquia de classes do pacote `java.io` (parte 2 de 2).

`InputStream` e `OutputStream` (subclasses de `Object`) são classes `abstract` que definem métodos para realizar entrada e saída baseada em `bytes` respectivamente.

A entrada/saída de arquivos é feita com `FileInputStream` (uma subclasse de `InputStream`) e `FileOutputStream` (uma subclasse de `OutputStream`). Utilizamos essas classes extensamente nos exemplos deste capítulo.

Os *pipes* são canais de comunicação sincronizados entre *threads* ou processos. Java oferece `PipedInputStream` (uma subclasse de `InputStream`) `PipedOutputStream` (uma subclasse de `OutputStream`) para estabelecer *pipes* entre duas *threads*. Uma *thread* envia dados para outra escrevendo em um `PipedOutputStream`. A *thread* de destino lê as informações do *pipe* através de um `PipedInputStream`.

Uma `PrintStream` (uma subclasse de `FilterOutputStream`) realiza a saída de texto para o fluxo especificado. Na verdade, temos utilizado a saída `PrintStream` por todo o texto até este ponto; `System.out` é um `PrintStream`, assim como `System.err`.

O `FilterInputStream` filtra um `InputStream` e um `FilterOutputStream` filtra um `OutputStream`; filtrar significa simplesmente que o fluxo de filtro fornece funcionalidade adicional como armazenamento em *buffer*, monitoramento de números de linha ou agregação de `bytes` de dados em unidades que formam um tipo primitivo de dados. `FilterInputStream` e `FilterOutputStream` são classes `abstract`, de modo que a funcionalidade adicional é fornecida por suas subclasses.

Ler dados diretamente como `bytes` é rápido, mas grosseiro. Normalmente os programas lêem dados como agregados de `bytes` que formam um `int`, um `float`, um `double` e assim por diante. Os programas Java podem usar várias classes para dar entrada e saída a dados de forma agregada.

O `RandomAccessFile` é útil para *aplicativos de acesso direto* e para os *aplicativos de processamento de transações*, como sistemas de reservas de passagens aéreas e sistemas de ponto de venda. Com um *arquivo de aces-*

so seqüencial, cada solicitação de entrada/saída sucessiva lê ou grava o próximo conjunto de dados consecutivo no arquivo. Com um *arquivo de acesso aleatório*, cada solicitação sucessiva de entrada/saída pode ser dirigida para qualquer parte do arquivo, talvez bem distante da parte do arquivo a que se faz referência na solicitação anterior. Aplicativos de acesso direto fornecem acesso rápido a itens de dados específicos em arquivos grandes; esses aplicativos são freqüentemente utilizados quando as pessoas estão esperando respostas – essas respostas devem ser disponibilizadas rapidamente ou as pessoas podem ficar impacientes e “levar seus negócios para outro lugar”.

A interface **DataInput** é implementada pelas classes **DataInputStream** e **RandomAccessFile** (discutidas mais adiante neste capítulo); cada um precisa ler tipos primitivos de dados de um fluxo. **DataInputStreams** permitem a um programa ler dados binários de um **InputStream**. A interface **DataInput** inclui os métodos **read** (para os arrays de **byte**), **readBoolean**, **readByte**, **readChar**, **readDouble**, **readFloat**, **readFully** (para arrays de **byte**), **readInt**, **readLong**, **readShort**, **readUnsignedByte**, **readUnsignedShort**, **readUTF** (para strings) e **skipBytes**.

A interface **DataOutput** é implementada pelas classes **DataOutputStream** (uma subclasse de **FilterOutputStream**) e **RandomAccessFile**; cada uma precisa escrever tipos primitivos de dados em um **OutputStream**. **DataOutputStreams** permitem a um programa gravar dados binários em um **OutputStream**. A interface **DataOutput** inclui os métodos **flush**, **size**, **write** (para um **byte**), **write** (para um array de **byte**), **writeBoolean**, **writeByte**, **writeBytes**, **writeChar**, **writeChars** (para **Strings** Unicode), **writeDouble**, **writeFloat**, **writeInt**, **writeLong**, **writeShort** e **writeUTF**.

O *armazenamento em buffers (buffering)* é uma técnica de aprimoramento do desempenho de E/S. Com um **BufferedOutputStream** (uma subclasse de classe **FilterOutputStream**), cada instrução de saída não resulta necessariamente em uma transferência física real de dados para o dispositivo de saída. Em vez disso, cada operação de saída é dirigida para uma região na memória, chamada *buffer*, que é suficientemente grande para armazenar os dados de muitas operações de saída. Assim, a transferência real para o dispositivo de saída é realizada em uma grande *operação física de saída* toda vez que o *buffer* se enche. As operações de saída dirigidas para o *buffer* de saída na memória são freqüentemente chamadas de *operações lógicas de saída*. Com um **BufferedOutputStream**, um *buffer* parcialmente preenchido pode ser forçado a dar saída no dispositivo a qualquer momento através da invocação do método **flush** sobre o objeto de fluxo.



Dica de desempenho 16.1

Como as operações físicas de saída são extremamente lentas, comparadas às velocidades de acesso à memória, colocar a saída em um buffer oferece melhorias significativas de desempenho em relação às saídas sem usar buffer.

Com um **BufferedInputStream** (uma subclasse da classe **FilterInputStream**), muitos pedaços ou trechos “lógicos” de dados de um arquivo são lidos como uma grande *operação física de entrada* para um *buffer* na memória. A medida que o programa solicita cada novo trecho dos dados, é feita uma busca no *buffer* (trata-se de uma *operação lógica de entrada*). Quando o *buffer* está vazio, a operação física de entrada do dispositivo de entrada é realizada lendo o próximo grupo de trechos “lógicos” de dados. Portanto, o número de operações físicas reais de entrada é pequeno, comparado com o número de solicitações de leitura emitido pelo programa.



Dica de desempenho 16.2

Como as operações físicas de entrada são extremamente lentas, comparadas à velocidade de acesso à memória, usar buffers de entrada normalmente oferece melhorias significativas de desempenho em relação a não usar buffers de entrada.

O **PushBackInputStream** (uma subclasse da classe **FilterInputStream**) é utilizada para aplicações mais exóticas do que a maioria dos usuários precisa. Essencialmente, o aplicativo que lê um **PushBackInputStream** lê *bytes* do fluxo e forma agregados formados por vários *bytes*. Às vezes, para determinar que o agregado está completo, o aplicativo deve ler o primeiro caractere além do fim do primeiro agregado. Uma vez que o programa determinou que o agregado atual está completo, o caractere extra é reinserido (“pushed back”) no fluxo. **PushBackInputStreams** são utilizados por programas (como compiladores) que *analisam sintaticamente (parse)* suas entradas, isto é, eles as dividem em unidades significativas (como palavras-chave, identificadores e operadores que o compilador Java deve reconhecer).

Quando as variáveis de instância de objetos são enviadas para a saída em um arquivo de disco, em certo sentido perdemos as informações do tipo do objeto. Em um disco, temos apenas dados, não as informações de tipo. Se o pro-

grama que lê esses dados conhece o tipo de objeto a que eles correspondem, esses dados são simplesmente lidos para objetos desse tipo. Às vezes, queremos ler ou gravar um objeto inteiro em um arquivo. As classes **ObjectInputStream** e **ObjectOutputStream**, que implementam respectivamente as interfaces **ObjectInput** e **ObjectOutput**, permitem que um objeto inteiro seja lido de um arquivo ou gravado nele (ou outro tipo de fluxo). Frequentemente encadeamos **ObjectInputStreams** com **FileInputStreams** (também encadeamos **ObjectOutputStreams** com **FileOutputStreams**.) A interface **ObjectOutput** tem um método **writeObject**, que recebe como argumento um **Object** que implementa a interface **Serializable**, e grava suas informações na **OutputStream**. Correspondentemente, a interface **ObjectInput** requer o método **readObject**, que lê e devolve um **Object** de um **InputStream**. Depois da leitura de um **Object**, ele deve ser convertido para o tipo desejado. Além disso, essas interfaces incluem outros métodos centrados em **Object**, bem como os mesmos métodos que **DataInput** e **DataOutput** para leitura e gravação de tipos de dados primitivos.

O fluxo de E/S de Java inclui capacidades para entrada de *arrays* de **byte** na memória e saída de *arrays* de **byte** na memória. O **ByteArrayInputStream** (uma subclasse de **InputStream**) realiza suas entradas a partir de um *array* de **byte** na memória. O **ByteArrayOutputStream** (uma subclasse de **OutputStream**) coloca a saída em um *array* de **byte** na memória. Uma aplicação da E/S de *array* de **byte** é a validação de dados. O programa pode inserir uma linha inteira do fluxo de entrada por vez em um *array* de **byte**. Então uma rotina de validação pode escrutinar o conteúdo do *array* de **byte** e corrigir os dados, se necessário. O programa agora pode continuar a entrada a partir do *array* de **byte**, sabendo que os dados de entrada estão no formato adequado. Dar saída para um *array* de **byte** é uma boa maneira de tirar proveito das poderosas capacidades de formatação de fluxos de saída de Java. Por exemplo, os dados podem ser preparados em um *array* de **byte**, usando a mesma formatação que será exibida posteriormente, então gravar em um disco para preservar a imagem de tela.

O **SequenceInputStream** (uma subclasse de **InputStream**) permite que vários **InputStreams** sejam concatenados de modo que o programa veja o grupo como um único **InputStream**. À medida que o fim de cada fluxo de entrada é alcançado, o fluxo é fechado e o próximo fluxo na seqüência é aberto.

Além dos fluxos baseados em *bytes*, Java fornece as classes **Reader** e **Writer**, que são fluxos baseados em caracteres Unicode dois *bytes*. A maioria dos fluxos baseados em *bytes* possui classes correspondentes **Reader** ou **Writer** baseadas em caracteres.

A classe **BufferedReader** (uma subclasse da classe **abstract Reader**) e a classe **BufferedWriter** (uma subclasse da classe **abstract Writer**) permitem armazenamento eficiente em *buffer* para fluxos baseados em caracteres. Os fluxos baseados em caracteres utilizam caracteres Unicode – esses fluxos podem processar dados em qualquer linguagem que seja representada pelo conjunto de caracteres Unicode.

A classe **CharArrayReader** e a classe **CharArrayWriter** lêem e gravam um fluxo de caracteres em um *array* de caracteres.

A **PushbackReader** (subclasse da classe **abstract FilterReader**) permite que os caracteres sejam colocados de volta em um fluxo de caracteres. A **LineNumberReader** (uma subclasse de **BufferedReader**) fornece um fluxo de caracteres com *buffer* que monitora números de linha (isto é, uma nova linha, um retorno de carro ou uma combinação de quebra de linha e de retorno de carro).

As classes **FileReader** (uma subclasse de **InputStreamReader**) e **FileWriter** (uma subclasse de **OutputStreamWriter**) lêem e gravam caracteres em um arquivo, respectivamente. As classes **PipedReader** e **PipedWriter** fornecem fluxos de caracteres colocados em *pipes*. As classes **StringReader** e **StringWriter** lêem e gravam caracteres em **Strings**. Uma **PrintWriter** grava caracteres em um fluxo.

A classe **File** permite que os programas obtenham informações sobre um arquivo ou diretório. Discutimos a classe **File** extensamente na Seção 16.12

16.4 Criando um arquivo de acesso seqüencial

Java não impõe nenhuma estrutura a um arquivo. Logo, noções como “registro” não existem em arquivos Java. Portanto, o programador deve estruturar arquivos para atender aos requisitos de aplicativos. No exemplo a seguir, vemos como o programador impõe uma estrutura simples de registro a um arquivo. Primeiro, apresentamos o programa, depois o analisaremos em detalhes.

O programa das Figs. 16.4 a 16.6 cria um arquivo simples de acesso seqüencial que poderia ser utilizado em um sistema de contas a receber para ajudar a gerenciar o valor devido por um cliente da empresa. Para cada cliente, o programa obtém um número de conta, o primeiro nome, o sobrenome e o saldo do cliente (isto é, a quantia que o cliente ainda deve à empresa pelas mercadorias e pelos serviços recebidos no passado). Os dados obtidos para cada

cliente constituem um registro desse cliente. O programa usa o número da conta como chave de registro; isto é, o arquivo será criado e mantido ordenado pelo número de conta. [Nota: esse programa parte do princípio de que o usuário insere os registros em ordem de número de conta. Em um sistema abrangente de contas a receber, um recurso de classificação é fornecido para o usuário poder inserir o registro em qualquer ordem – os registros então seriam classificados e gravados no arquivo.]

A maioria dos programas neste capítulo tem uma GUI semelhante, por isso este programa define a classe **BankUI** (Fig. 16.4) para encapsular essa GUI (veja a segunda tela na Fig. 16.6). O programa também define a classe **BankAccountRecord** (Fig. 16.5) para encapsular as informações de registro dos clientes (isto é, conta, nome etc.) utilizadas pelos exemplos deste capítulo. Para reutilização, as classes **BankUI** e **AccountRecord** são definidas no pacote **com.deitel.jhttp4.ch16**.

[Nota: a maior parte dos programas neste capítulo usa as classes **BankUI** e **AccountRecord**. Quando você compilar estas classes, ou quaisquer outras que serão reutilizadas neste capítulo, você deve colocar as classes em um diretório comum. Quando você compilar as classes que usam **BankUI** e **AccountRecord**, certifique-se de especificar o argumento de linha de comando – **-classpath** em **javac** e **java** como em:

```
javac -classpath .;localizaçãoDoPacote NomeDaClasse.java
java -classpath .;localizaçãoDoPacote NomeDaClasse
```

onde *localizaçãoDoPacote* representa o diretório comum no qual se encontram as classes do pacote **com.deitel.jhttp4.ch16** e *NomeDaClasse* representa a classe a ser compilada ou executada. Certifique-se de incluir o diretório atual (especificado por **.**) no caminho da classe. [Nota: se as classes do pacote estiverem em um arquivo JAR, a localização do pacote deve incluir a localização e o nome do arquivo JAR atual.] Ademais, o separador de caminho mostrado (**,**, que é usado no Microsoft Windows) deve ser apropriado para sua plataforma (como: no UNIX/Linux).]

A classe **BankUI** (Fig. 16.4) contém dois **JButtons** e os arrays de **JLabels** e **JTextFields**. O número de **JLabels** e **JTextFields** é estabelecido pelo construtor nas linhas 34 a 75. Os métodos **getFieldValues** (linhas 116 a 124), **setFieldValues** (linhas 104 a 113) e **clearFields** (linhas 96 a 100) são utilizados para manipular o texto dos **JTextFields**. Os métodos **getFields** (linhas 90 a 93), **getDoTask1Button** (linhas 78 a 81) e **getDoTask2Button** (linhas 84 a 87) devolvem componentes GUI individuais para que um programa cliente possa adicionar **ActionListeners** (por exemplo).

```

1 // Fig. 16.4: BankUI.java
2 // Uma GUI reutilizável para os exemplos deste capítulo.
3 package com.deitel.jhttp4.ch16;
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class BankUI extends JPanel {
12
13     // texto de rótulos para a GUI
14     protected final static String names[] = { "Account number",
15         "First name", "Last name", "Balance",
16         "Transaction Amount" };
17
18     // componentes GUI; protected para acesso por futuras subclasses
19     protected JLabel labels[];
20     protected JTextField fields[];
21     protected JButton doTask1, doTask2;
22     protected JPanel innerPanelCenter, innerPanelSouth;
23
24     // número de campos de texto na GUI
25     protected int size;
26
27     // constantes que representam campos de texto na GUI

```

Fig. 16.4 BankUI contém uma GUI reutilizável para diversos programas (parte 1 de 3).

```

28     public static final int ACCOUNT = 0, FIRSTNAME = 1,
29         LASTNAME = 2, BALANCE = 3, TRANSACTION = 4;
30
31     // Configura a GUI. Argumento 4 para o construtor cria quatro linhas
32     // de componentes GUI. Argumento 5 para o construtor (usado em outro
33     // programa, adiante) cria cinco linhas de componentes GUI.
34     public BankUI( int mySize )
35     {
36         size = mySize;
37         labels = new JLabel[ size ];
38         fields = new JTextField[ size ];
39
40         // cria rótulos
41         for ( int count = 0; count < labels.length; count++ )
42             labels[ count ] = new JLabel( names[ count ] );
43
44         // cria campos de texto
45         for ( int count = 0; count < fields.length; count++ )
46             fields[ count ] = new JTextField();
47
48         // cria painel para dispor os rótulos e os campos de texto
49         innerPanelCenter = new JPanel();
50         innerPanelCenter.setLayout( new GridLayout( size, 2 ) );
51
52         // anexa rótulos e campos de texto a innerPanelCenter
53         for ( int count = 0; count < size; count++ ) {
54             innerPanelCenter.add( labels[ count ] );
55             innerPanelCenter.add( fields[ count ] );
56         }
57
58         // cria botões genéricos; sem rótulos ou tratadores de eventos
59         doTask1 = new JButton();
60         doTask2 = new JButton();
61
62         // cria painel para dispor os botões e anexa os botões
63         innerPanelSouth = new JPanel();
64         innerPanelSouth.add( doTask1 );
65         innerPanelSouth.add( doTask2 );
66
67         // configura leiaute deste contêiner e anexa painéis a ele
68         setLayout( new BorderLayout() );
69         add( innerPanelCenter, BorderLayout.CENTER );
70         add( innerPanelSouth, BorderLayout.SOUTH );
71
72         // valida o leiaute
73         validate();
74
75     } // fim do construtor
76
77     // devolve referência para o botão genérico de tarefa doTask1
78     public JButton getDoTask1Button()
79     {
80         return doTask1;
81     }
82
83     // devolve referência para o botão genérico de tarefa doTask2
84     public JButton getDoTask2Button()
85     {
86         return doTask2;
87     }

```

Fig. 16.4 BankUI contém uma GUI reutilizável para diversos programas (parte 2 de 3).

```

88
89     // devolve referência fields para um array de JTextFields
90     public JTextField[] getFields()
91     {
92         return fields;
93     }
94
95     // limpa o conteúdo dos campos de texto
96     public void clearFields()
97     {
98         for ( int count = 0; count < size; count++ )
99             fields[ count ].setText( "" );
100    }
101
102    // configura valores dos campos de texto; dispara IllegalArgumentException
103    // se o argumento contiver um número incorreto de Strings
104    public void setFieldValues( String strings[] )
105        throws IllegalArgumentException
106    {
107        if ( strings.length != size )
108            throw new IllegalArgumentException( "There must be " +
109                size + " Strings in the array" );
110
111        for ( int count = 0; count < size; count++ )
112            fields[ count ].setText( strings[ count ] );
113    }
114
115    // obtém array de Strings com o conteúdo atual dos campos de texto
116    public String[] getFieldValues()
117    {
118        String values[] = new String[ size ];
119
120        for ( int count = 0; count < size; count++ )
121            values[ count ] = fields[ count ].getText();
122
123        return values;
124    }
125
126 } // fim da classe BankUI

```

Fig. 16.4 BankUI contém uma GUI reutilizável para diversos programas (parte 3 de 3).

A classe **AccountRecord** (Fig. 16.5) implementa a interface **Serializable** que permite que os objetos **AccountRecord** sejam utilizados com **ObjectInputStreams** e **ObjectOutputStreams**. A interface **Serializable** é conhecida como *interface de marcação*. Tal interface não contém métodos. A classe que implementa esta interface é *marcada* como objeto **Serializable**, o que é importante porque um **ObjectOutputStream** não gerará como saída um objeto a menos que seja um objeto **Serializable**. Em uma classe que implementa **Serializable**, o programador deve assegurar que cada variável de instância da classe seja um tipo **Serializable**, ou deve declarar variáveis particulares de instância como **transient** para indicar que essas variáveis não são **Serializable** e que devem ser ignoradas durante o processo de serialização. Por *default*, todas as variáveis de tipos primitivos são transientes. Para tipos não-primitivos, você deve verificar a definição da classe (e possivelmente suas superclasses) para assegurar que o tipo é **Serializable**. A classe **AccountRecord** contém os membros de dados **private account**, **firstName**, **lastName** e **balance**. Essa classe também fornece os métodos **public “get”** e **“set”** para acessar os membros de dados **private**.

```

1 // Fig. 16.5: AccountRecord.java
2 // Uma classe que representa um registro de informações.

```

Fig. 16.5 Classe **AccountRecord** mantém as informações sobre uma conta (parte 1 de 3).

```
3 package com.deitel.jhttp4.ch16;
4
5 // Pacotes do núcleo de Java
6 import java.io.Serializable;
7
8 public class AccountRecord implements Serializable {
9     private int account;
10    private String firstName;
11    private String lastName;
12    private double balance;
13
14    // construtor sem argumentos chama outro construtor
15    // com valores default
16    public AccountRecord()
17    {
18        this( 0, "", "", 0.0 );
19    }
20
21    // inicializa um registro
22    public AccountRecord( int acct, String first,
23                          String last, double bal )
24    {
25        setAccount( acct );
26        setFirstName( first );
27        setLastName( last );
28        setBalance( bal );
29    }
30
31    // configura o número da conta
32    public void setAccount( int acct )
33    {
34        account = acct;
35    }
36
37    // obtém o número da conta
38    public int getAccount()
39    {
40        return account;
41    }
42
43    // configura primeiro nome
44    public void setFirstName( String first )
45    {
46        firstName = first;
47    }
48
49    // obtém primeiro nome
50    public String getFirstName()
51    {
52        return firstName;
53    }
54
55    // configura sobrenome
56    public void setLastName( String last )
57    {
58        lastName = last;
59    }
60
61    // obtém sobrenome
62    public String getLastname()
```

Fig. 16.5 Classe AccountRecord mantém as informações sobre uma conta (parte 2 de 3).

```

63     {
64         return lastName;
65     }
66
67     // configura o saldo
68     public void setBalance( double bal )
69     {
70         balance = bal;
71     }
72
73     // obtém o saldo
74     public double getBalance()
75     {
76         return balance;
77     }
78
79 } // fim da classe AccountRecord

```

Fig. 16.5 Classe `AccountRecord` mantém as informações sobre uma conta (parte 3 de 3).

Agora vamos discutir o código que cria o arquivo de acesso seqüencial (Fig. 16.6). Nesse exemplo, introduzimos a classe `JFileChooser` (pacote `javax.swing`) para selecionar arquivos (veja a primeira tela na Fig. 16.6). A linha 103 constrói uma instância de `JFileChooser` e a atribui à referência `fileChooser`. As linhas 104 e 105, chamam o método `setFileSelectionMode` para especificar o que o usuário pode selecionar a partir de `fileChooser`. Nesse programa, utilizamos a constante `static FILES_ONLY` de `JFileChooser` para indicar que somente arquivos podem ser selecionados. Outras constantes `static` são `FILES_AND_DIRECTORIES` e `DIRECTORIES_ONLY`.

```

1 // Fig. 16.6: CreateSequentialFile.java
2 // Demonstrando saída de objetos com a classe ObjectOutputStream.
3 // Os objetos são gravados seqüencialmente em um arquivo.
4
5 // Pacotes do núcleo de Java
6 import java.io.*;
7 import java.awt.*;
8 import java.awt.event.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 // Pacotes Deitel
14 import com.deitel.jhtp4.ch16.BankUI;
15 import com.deitel.jhtp4.ch16.AccountRecord;
16
17 public class CreateSequentialFile extends JFrame {
18     private ObjectOutputStream output;
19     private BankUI userInterface;
20     private JButton enterButton, openButton;
21
22     // configura a GUI
23     public CreateSequentialFile()
24     {
25         super( "Creating a Sequential File of Objects" );
26
27         // cria instância de interface com o usuário reusável
28         userInterface = new BankUI( 4 );    // quatro campos de texto
29         getContentPane().add(

```

Fig. 16.6 Criando um arquivo seqüencial (parte 1 de 5).

```

30         userInterface, BorderLayout.CENTER );
31
32     // obtém referência para o botão de tarefa genérica doTask1 em BankUI
33     // e configura o botão para uso neste programa
34     openButton = userInterface.getDoTask1Button();
35     openButton.setText( "Save into File ..." );
36
37     // registra ouvinte para chamar openFile quando o botão é pressionado
38     openButton.addActionListener(
39
40         // classe interna anônima para tratar eventos de openButton
41         new ActionListener() {
42
43             // chama openFile quando o botão é pressionado
44             public void actionPerformed( ActionEvent event )
45             {
46                 openFile();
47             }
48
49         } // fim da classe interna anônima
50
51     ); // fim da chamada para addActionListener
52
53     // obtém referência para o botão de tarefa genérica doTask2 em BankUI
54     // e configura o botão para uso neste programa
55     enterButton = userInterface.getDoTask2Button();
56     enterButton.setText( "Enter" );
57     enterButton.setEnabled( false ); // desabilita o botão
58
59     // registra ouvinte para chamar addRecord quando o botão é pressionado
60     enterButton.addActionListener(
61
62         // classe interna anônima para tratar eventos de enterButton
63         new ActionListener() {
64
65             // chama addRecord quando o botão é pressionado
66             public void actionPerformed( ActionEvent event )
67             {
68                 addRecord();
69             }
70
71         } // fim da classe interna anônima
72
73     ); // fim da chamada para addActionListener
74
75     // registra ouvinte da janela para tratar o evento de fechamento da janela
76     addWindowListener(
77
78         // classe interna anônima para tratar do evento windowClosing
79         new WindowAdapter() {
80
81             // adiciona o registro corrente na GUI ao arquivo, depois fecha o arquivo
82             public void windowClosing( WindowEvent event )
83             {
84                 if ( output != null )
85                     addRecord();
86
87                 closeFile();
88             }
89

```

Fig. 16.6 Criando um arquivo seqüencial (parte 2 de 5).

```

90         } // fim da classe interna anônima
91
92     ); // fim da chamada para addWindowListener
93
94     setSize( 300, 200 );
95     show();
96
97 } // fim do construtor de CreateSequentialFile
98
99 // permite que o usuário especifique o nome do arquivo
100 private void openFile()
101 {
102     // exibe diálogo de arquivo, para que o usuário escolha o arquivo a ser aberto
103     JFileChooser fileChooser = new JFileChooser();
104     fileChooser.setFileSelectionMode(
105         JFileChooser.FILES_ONLY );
106
107     int result = fileChooser.showSaveDialog( this );
108
109     // se o usuário clicou no botão Cancel no diálogo, retorna
110     if ( result == JFileChooser.CANCEL_OPTION )
111         return;
112
113     // obtém arquivo selecionado
114     File fileName = fileChooser.getSelectedFile();
115
116     // exibe erro se inválido
117     if ( fileName == null ||
118         fileName.getName().equals( "" ) )
119         JOptionPane.showMessageDialog( this,
120             "Invalid File Name", "Invalid File Name",
121             JOptionPane.ERROR_MESSAGE );
122
123     else {
124
125         // abre o arquivo
126         try {
127             output = new ObjectOutputStream(
128                 new FileOutputStream( fileName ) );
129
130             openButton.setEnabled( false );
131             enterButton.setEnabled( true );
132         }
133
134         // processa exceções decorrentes da abertura do arquivo
135         catch ( IOException ioException ) {
136             JOptionPane.showMessageDialog( this,
137                 "Error Opening File", "Error",
138                 JOptionPane.ERROR_MESSAGE );
139         }
140     }
141
142 } // fim do método openFile
143
144 // fecha o arquivo e termina o aplicativo
145 private void closeFile()
146 {
147     // fecha o arquivo
148     try {
149         output.close();

```

Fig. 16.6 Criando um arquivo seqüencial (parte 3 de 5).

```

150         System.exit( 0 );
151     }
152 }
153
154 // processa exceções decorrentes do fechamento do arquivo
155 catch( IOException ioException ) {
156     JOptionPane.showMessageDialog( this,
157         "Error closing file", "Error",
158         JOptionPane.ERROR_MESSAGE );
159     System.exit( 1 );
160 }
161 }
162
163 // adiciona registro ao arquivo
164 public void addRecord()
165 {
166     int accountNumber = 0;
167     AccountRecord record;
168     String fieldValues[] = userInterface.getFieldValues();
169
170     // se o valor do campo de conta não é vazio
171     if ( ! fieldValues[ BankUI.ACCOUNT ].equals( "" ) ) {
172
173         // grava os valores no arquivo
174         try {
175             accountNumber = Integer.parseInt(
176                 fieldValues[ BankUI.ACCOUNT ] );
177
178             if ( accountNumber > 0 ) {
179
180                 // cria novo registro
181                 record = new AccountRecord( accountNumber,
182                     fieldValues[ BankUI.FIRSTNAME ],
183                     fieldValues[ BankUI.LASTNAME ],
184                     Double.parseDouble(
185                         fieldValues[ BankUI.BALANCE ] ) );
186
187                 // envia registro para a saída e descarrega o buffer
188                 output.writeObject( record );
189                 output.flush();
190             }
191
192             // limpa os campos de texto
193             userInterface.clearFields();
194         }
195
196         // processa número de conta ou formato do saldo inválidos
197         catch ( NumberFormatException formatException ) {
198             JOptionPane.showMessageDialog( this,
199                 "Bad account number or balance",
200                 "Invalid Number Format",
201                 JOptionPane.ERROR_MESSAGE );
202         }
203
204         // processa exceções decorrentes da operação de saída no arquivo
205         catch ( IOException ioException ) {
206             closeFile();
207         }
208     } // fim do if

```

Fig. 16.6 Criando um arquivo seqüencial (parte 4 de 5).

```

210
211     } // fim do método addRecord
212
213     // executa o aplicativo;
214     // o construtor CreateSequentialFile exibe a janela
215     public static void main( String args[] )
216     {
217         new CreateSequentialFile();
218     }
219
220 } // fim da classe CreateSequentialFile

```

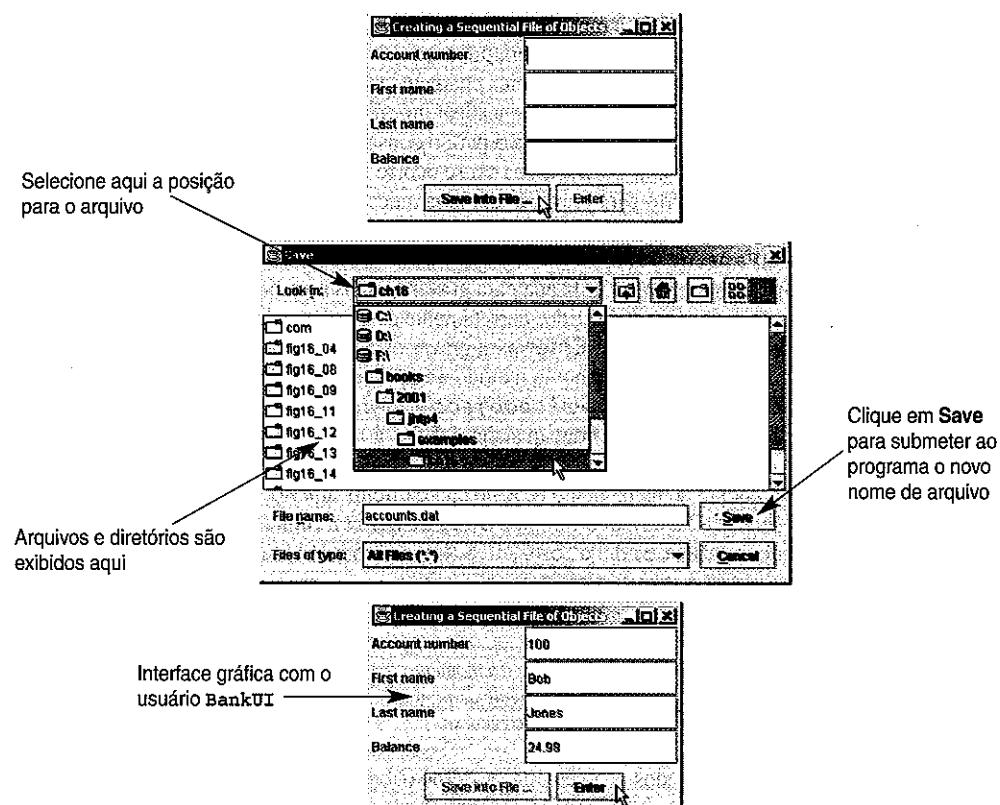


Fig. 16.6 Criando um arquivo seqüencial (parte 5 de 5).

A linha 107, chama o método `showSaveDialog` para exibir o diálogo de `JFileChooser` intitulado `Save`. O argumento `this` especifica que a janela de diálogo *pai* de `JFileChooser` é utilizado para determinar a posição do diálogo na tela. Se `null` for passado, o diálogo é exibido no centro da janela; caso contrário, o diálogo é centrado sobre a janela da aplicação. Quando exibido, o diálogo `JFileChooser` não permite ao usuário interagir com qualquer outra janela de programa até o diálogo `JFileChooser` ser fechado, clicando em `Save` ou `Cancel`. Os diálogos que se comportam dessa maneira chamam-se diálogos *modais*. O usuário seleciona a unidade, o diretório e o nome do arquivo e clica em `Save`. O método `showSaveDialog` devolve um inteiro que especifica o botão (`Save` ou `Cancel`) que foi clicado pelo usuário para fechar o diálogo. A linha 110 testa se se clicou em `Cancel` comparando `result` com a constante `static CANCEL_OPTION`. Se foi, o método é retornado.

A linha 114 recupera o arquivo que o usuário selecionou com o método `getSelectedFile`, que devolve um objeto do tipo `File` que encapsula as informações sobre o arquivo (por exemplo, nome e localização), mas não re-

presenta o conteúdo do arquivo. Esse objeto `File` não abre o arquivo. Atribuímos esse objeto `File` à referência `fileName`.

Como afirmado antes, o programa abre um arquivo criando um objeto das classes de fluxo `FileInputStream` ou `FileOutputStream`. Nesse exemplo, o arquivo será aberto para saída, assim o programa cria um `FileOutputStream`. Um argumento é passado para o construtor de `FileOutputStream` – um objeto `File`. Os arquivos já existentes abertos para saída são *truncados* – todos os dados no arquivo são descartados.



Erro comum de programação 16.1

É um erro de lógica abrir um arquivo existente para saída quando, na realidade, o usuário quer preservar o arquivo. O conteúdo do arquivo é descartado sem aviso.

A classe `FileOutputStream` fornece métodos para gravar *arrays* de `byte` e `bytes` individuais em um arquivo. Para esse programa, precisamos gravar objetos em um arquivo – um recurso não fornecido por `FileOutputStream`. A solução para esse problema é uma técnica chamada *encadeamento de objetos de fluxo* – a capacidade de adicionar os serviços de um fluxo a outro. Para encadear um `ObjectOutputStream` com o `FileOutputStream`, passamos o objeto `FileOutputStream` para o construtor de `ObjectOutputStream` (linhas 127 e 128). O construtor pode disparar uma `IOException` se ocorre um problema durante a abertura do arquivo (por exemplo, quando um arquivo é aberto para gravação em uma unidade com espaço insuficiente, quando um arquivo de leitura é aberto para gravação, quando um arquivo inexistente é aberto para leitura). Se isso acontecer, o programa exibe um `JOptionPane`. Se a construção dos dois fluxos não disparar uma `IOException`, o arquivo é aberto. A referência `output` então pode ser utilizada para gravar objetos no arquivo.

O programa assume que os dados foram inseridos corretamente e na ordem adequada de número de registro. O usuário preenche os `JTextFields` e clica em `Enter` para gravar os dados no arquivo. O método `actionPerformed` do botão `Enter` (linhas 66 a 69) chama nosso método `addRecord` (linhas 164 a 211) para realizar a operação de gravação. A linha 188 chama o método `writeObject` para gravar o objeto `record` nesse arquivo. A linha 189 chama o método `flush` para assegurar que quaisquer dados armazenados na memória sejam gravados no arquivo imediatamente.

Quando o usuário clica no botão de fechamento (o `X` no canto direito superior da janela), o programa chama o método `windowClosing` (linhas 82 a 88), que compara `output` a `null` (linha 84). Se `output` não for `null`, o fluxo está aberto e os métodos `addRecord` e `closeFile` (linhas 145 a 161) são chamados. O método `closeFile` chama o método `close` para `output` para fechar o arquivo.



Dica de desempenho 16.3

Sempre libere recursos explicitamente e no primeiro momento possível em que se determinou que o recurso não é mais necessário. Isso torna o recurso imediatamente disponível para ser reutilizado por seu programa ou por outro programa, melhorando, assim, a utilização do recurso.



Ao utilizar objetos de fluxo encadeados, o objeto mais externo (`ObjectOutputStream`, nesse exemplo) deve ser utilizado para fechar o arquivo.

Dica de desempenho 16.4

Feche explicitamente cada arquivo assim que souber que o programa não fará referência ao arquivo novamente. Isso pode reduzir o uso de recursos em um programa que continuará rodando muito tempo depois que não for mais necessário fazer referência a um arquivo em particular. Essa prática também melhora a clareza do programa.

No exemplo de execução para o programa da Fig. 16.6, inserimos informações para cinco contas (veja a Fig. 16.7). O programa não mostra como os registros de dados realmente aparecem no arquivo. Para verificar se o arquivo foi criado com sucesso, na próxima seção criamos um programa para ler o arquivo.

16.5 Lendo dados de um arquivo de acesso seqüencial

Os dados são armazenados em arquivos para que possam ser recuperados para processamento quando necessário. A seção anterior demonstrou como criar um arquivo de acesso seqüencial. Nesta seção, discutimos como ler dados seqüencialmente de um arquivo.

O programa da Fig. 16.8 lê registros de um arquivo criado pelo programa da Fig. 16.6 e exibe o conteúdo dos registros. O programa abre o arquivo para entrada criando um objeto `FileInputStream`. O programa especifica o nome do arquivo a ser aberto como argumento para o construtor `FileInputStream`. Na Fig. 16.6, gravamos objetos no arquivo utilizando um objeto `ObjectOutputStream`. Os dados devem ser lidos do arquivo no mesmo formato em que foram gravados no arquivo. Portanto, utilizamos um `ObjectInputStream` encadeado com um `FileInputStream` nesse programa. Observe que a terceira captura de tela do exemplo mostra a GUI exibindo o último registro no arquivo.

Exemplo de dados

100	Bob	Jones	24.98
200	Steve	Doe	-345.67
300	Pam	White	0.00
400	Sam	Stone	-42.16
500	Sue	Rich	224.62

Fig. 16.5 Exemplo de dados para o programa da Fig. 16.4.

```

1 // Fig. 16.8: ReadSequentialFile.java
2 // Este programa lê um arquivo de objetos seqüencialmente
3 // e exibe cada registro.
4
5 // Pacotes do núcleo de Java
6 import java.io.*;
7 import java.awt.*;
8 import java.awt.event.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 // Pacotes Deitel
14 import com.deitel.jhttp4.ch16.*;
15
16 public class ReadSequentialFile extends JFrame {
17     private ObjectInputStream input;
18     private BankUI userInterface;
19     private JButton nextButton, openButton;
20
21     // Construtor — inicializa a Frame
22     public ReadSequentialFile()
23     {
24         super( "Reading a Sequential File of Objects" );
25
26         // cria uma instância da interface com o usuário reusável
27         userInterface = new BankUI( 4 );    // quatro campos de texto textfields
28         getContentPane().add(
29             userInterface, BorderLayout.CENTER );
30
31         // obtém referência para botão de tarefa genérico doTask1 de BankUI
32         openButton = userInterface.getDoTask1Button();
33         openButton.setText( "Open File" );
34

```

Fig. 16.8 Lendo um arquivo seqüencial (parte 1 de 5).

```

35      // registra ouvinte para chamar openFile quando o botão é pressionado
36      openButton.addActionListener(
37
38          // classe interna anônima para tratar eventos de openButton
39          new ActionListener() {
40
41              // fecha arquivo e termina aplicativo
42              public void actionPerformed( ActionEvent event ) {
43
44                  openFile();
45              }
46
47          } // fim da classe interna anônima
48
49      ); // fim da chamada para addActionListener
50
51      // registra ouvinte da janela para o evento de fechamento da janela
52      addWindowListener(
53
54          // classe interna anônima para tratar evento windowClosing
55          new WindowAdapter() {
56
57              // fecha o arquivo e termina o aplicativo
58              public void windowClosing( WindowEvent event ) {
59
60                  if ( input != null )
61                      closeFile();
62
63                  System.exit( 0 );
64              }
65
66          } // fim da classe interna anônima
67
68      ); // fim da chamada para addWindowListener
69
70      // obtém referência para o botão de tarefa genérico doTask2 de BankUI
71      nextButton = userInterface.getDoTask2Button();
72      nextButton.setText( "Next Record" );
73      nextButton.setEnabled( false );
74
75      // registra ouvinte para chamar readRecord quando o botão é pressionado
76      nextButton.addActionListener(
77
78          // classe interna anônima para tratar eventos de nextRecord
79          new ActionListener() {
80
81              // chama readRecord quando o usuário clica em nextRecord
82              public void actionPerformed( ActionEvent event ) {
83
84                  readRecord();
85              }
86
87          } // fim da classe interna anônima
88
89      ); // fim da chamada para addActionListener
90
91      pack();
92      setSize( 300, 200 );
93      show();
94

```

Fig. 16.8 Lendo um arquivo seqüencial (parte 2 de 5).

```

95     } // fim do construtor ReadSequentialFile
96
97     // permite que o usuário selecione o arquivo a ser aberto
98     private void openFile()
99     {
100         // exibe o diálogo de arquivo para o usuário escolher o arquivo a ser aberto
101         JFileChooser fileChooser = new JFileChooser();
102         fileChooser.setFileSelectionMode(
103             JFileChooser.FILES_ONLY );
104
105         int result = fileChooser.showOpenDialog( this );
106
107         // se o usuário clicou no botão Cancel do diálogo, retorna
108         if ( result == JFileChooser.CANCEL_OPTION )
109             return;
110
111         // obtém arquivo selecionado
112         File fileName = fileChooser.getSelectedFile();
113
114         // exibe erro se o nome do arquivo for inválido
115         if ( fileName == null ||
116             fileName.getName().equals( "" ) )
117             JOptionPane.showMessageDialog( this,
118                 "Invalid File Name", "Invalid File Name",
119                 JOptionPane.ERROR_MESSAGE );
120
121     else {
122
123         // abre o arquivo
124         try {
125             input = new ObjectInputStream(
126                 new FileInputStream( fileName ) );
127
128             openButton.setEnabled( false );
129             nextButton.setEnabled( true );
130         }
131
132         // processa exceções decorrentes da abertura do arquivo
133         catch ( IOException ioException ) {
134             JOptionPane.showMessageDialog( this,
135                 "Error Opening File", "Error",
136                 JOptionPane.ERROR_MESSAGE );
137         }
138
139     } // fim do else
140
141 } // fim do método openFile
142
143 // lê um registro do arquivo
144 public void readRecord()
145 {
146     AccountRecord record;
147
148     // lê os valores do arquivo
149     try {
150         record = ( AccountRecord ) input.readObject();
151
152         // cria array de Strings para exibir na GUI
153         String values[] = {
154             String.valueOf( record.getAccount() ),

```

Fig. 16.8 Lendo um arquivo seqüencial (parte 3 de 5).

```

155         record.getFirstName(),
156         record.getLastName(),
157         String.valueOf( record.getBalance() ) );
158
159         // exibe o conteúdo do registro
160         userInterface.setFieldValues( values );
161     }
162
163     // exibe mensagem quando o fim do arquivo é encontrado
164     catch ( EOFException endOfFileException ) {
165         nextButton.setEnabled( false );
166
167         JOptionPane.showMessageDialog( this,
168             "No more records in file",
169             "End of File", JOptionPane.ERROR_MESSAGE );
170     }
171
172     // exibe mensagem de erro se não consegue ler o objeto
173     // porque a classe não foi encontrada
174     catch ( ClassNotFoundException classNotFoundException ) {
175         JOptionPane.showMessageDialog( this,
176             "Unable to create object",
177             "Class Not Found", JOptionPane.ERROR_MESSAGE );
178     }
179
180     // exibe mensagem de erro se não consegue ler
181     // devido a um problema com o arquivo
182     catch ( IOException ioException ) {
183         JOptionPane.showMessageDialog( this,
184             "Error during read from file",
185             "Read Error", JOptionPane.ERROR_MESSAGE );
186     }
187 }
188
189 // fecha o arquivo e termina o aplicativo
190 private void closeFile()
191 {
192     // fecha o arquivo e sai
193     try {
194         input.close();
195         System.exit( 0 );
196     }
197
198     // processa exceção durante o fechamento do arquivo
199     catch ( IOException ioException ) {
200         JOptionPane.showMessageDialog( this,
201             "Error closing file",
202             "Error", JOptionPane.ERROR_MESSAGE );
203
204         System.exit( 1 );
205     }
206 }
207
208 // executa o aplicativo;
209 // o construtor ReadSequentialFile exibe a janela
210 public static void main( String args[] )
211 {
212     new ReadSequentialFile();
213 }
214
215 } // fim da classe ReadSequentialFile

```

Fig. 16.8 Lendo um arquivo seqüencial (parte 4 de 5).

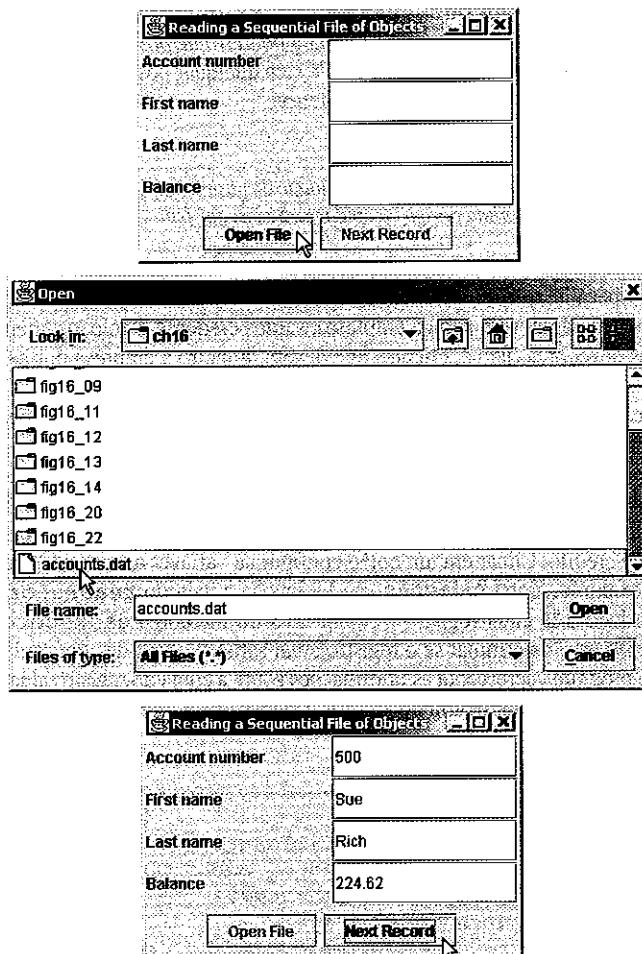


Fig. 16.8 Lendo um arquivo seqüencial (parte 5 de 5).

Como a maior parte do código nesse exemplo é semelhante à Fig. 16.6, discutimos somente as principais linhas do código que são diferentes. A linha 105 chama o método `showOpenDialog` de `JFileChooser` para exibir o diálogo **Open** (primeira captura de tela na Fig. 16.8). O comportamento e a GUI são os mesmos do diálogo exibido por `showSaveDialog`, exceto pelo fato de que o título do diálogo e o botão **Save** são substituídos por **Open**.

As linhas 125 e 126 criam um objeto `ObjectInputStream` e o atribuem a `input`. O `File fileName` é passado para o construtor `FileInputStream` para abrir o arquivo.

O programa lê um registro do arquivo toda vez que o usuário clica no botão **Next Record**. A linha 84 no método `actionPerformed` de **Next Record** chama o método `readRecord` (linhas 144 a 187) para ler um registro do arquivo. A linha 150 chama o método `readObject` para ler um `Object` do `ObjectInputStream`. Para utilizar os métodos específicos de `AccountRecord`, fazemos a coerção do `Object` devolvido para o tipo `AccountRecord`. Se o marcador de fim do arquivo for encontrado durante a leitura, `readObject` dispara uma `EndOfFileException`.

Para recuperar seqüencialmente os dados de um arquivo, os programas normalmente começam a ler a partir do início do arquivo e lêem todos os dados consecutivamente até que os dados desejados sejam encontrados. Pode ser necessário processar o arquivo seqüencialmente várias vezes (a partir do início do arquivo) durante a execução de um programa. A classe `FileInputStream` não fornece a capacidade de voltar para o começo do arquivo para ler o arquivo novamente, a menos que o programa feche o arquivo e o reabra. Os objetos da classe `RandomAccess-`

File podem ser repositionados no início do arquivo. A classe **RandomAccessFile** fornece todos os recursos das classes **FileInputStream**, **FileOutputStream**, **DataInputStream** e **DataOutputStream** e adiciona vários outros métodos, incluindo **seek**, que reposiciona o *ponteiro de posição no arquivo* (o número que indica a posição do próximo *byte* no arquivo a ser lido ou gravado) em qualquer posição no arquivo. Entretanto, a classe **RandomAccessFile** não pode ler e escrever objetos inteiros.



Dica de desempenho 16.5

O processo de fechar e reabrir um arquivo com a finalidade de posicionar o ponteiro de posição de volta no início de um arquivo é uma tarefa demorada para o computador. Se isso for feito com freqüência, pode tornar mais lento o desempenho de seu programa.

O programa da Fig. 16.9 permite que um gerente de crédito exiba as informações das contas dos clientes com saldo zero (isto é, os clientes que não devem nada à empresa), saldos credores (isto é, os clientes para quem a empresa deve dinheiro) e saldos devedores (isto é, clientes que devem dinheiro à empresa por bens e serviços recebidos no passado).

O programa exibe botões que permitem que um gerente de crédito obtenha as informações de crédito. O botão **Credit balances** produz uma lista de contas com saldos credores. O botão **Debit balances** produz uma lista das contas com saldos devedores.

Os registros são exibidos em uma **JTextArea** chamada **recordDisplayArea**. As informações de registro são coletadas lendo-se o arquivo inteiro e determinando-se se cada registro satisfaz o critério para o tipo de conta selecionado pelo gerente de crédito. Clicar em um botão configura a variável **accountType** (linha 274) com o texto do botão clicado (por exemplo, **Zero balances**, etc.) e invoca o método **readRecords** (191 a 238), que percorre o arquivo com um laço e lê cada registro. A linha 218 do método **readRecords** chama o método **shouldDisplay** (241 a 259) para determinar se o registro atual satisfaz o tipo de conta solicitado. Se **shouldDisplay** devolver **true**, o programa acrescenta as informações sobre a conta do registro atual à **JTextArea recordDisplayArea**. Quando o marcador de fim do arquivo é alcançado, a linha 219 chama o método **closeFile** para fechar o arquivo.

```

1 // Fig. 16.9: CreditInquiry.java
2 // Este programa lê um arquivo seqüencialmente e exibe o
3 // conteúdo em uma área de texto de acordo com o tipo de
4 // conta que o usuário solicita (saldo credor,
5 // saldo devedor, ou saldo zero).
6
7 // Pacotes do núcleo de Java
8 import java.io.*;
9 import java.awt.*;
10 import java.awt.event.*;
11 import java.text.DecimalFormat;
12
13 // Pacotes de extensão de Java
14 import javax.swing.*;
15
16 // Pacotes Deitel
17 import com.deitel.jhtp4.ch16.AccountRecord;
18
19 public class CreditInquiry extends JFrame {
20     private JTextArea recordDisplayArea;
21     private JButton openButton,
22         creditButton, debitButton, zeroButton;
23     private JPanel buttonPanel;
24
25     private ObjectInputStream input;
26     private FileInputStream fileInput;
27     private String fileName;
28     private String accountType;
```

Fig. 16.9 Programa de consulta de crédito (parte 1 de 6).

```

29
30    // configura a GUI
31    public CreditInquiry()
32    {
33        super( "Credit Inquiry Program" );
34
35        Container container = getContentPane();
36
37        // configura o painel para os botões
38        buttonPanel = new JPanel();
39
40        // cria e configura o botão para abrir o arquivo
41        openButton = new JButton( "Open File" );
42        buttonPanel.add( openButton );
43
44        // registra o ouvinte para openButton
45        openButton.addActionListener(
46
47            // classe interna anônima para tratar eventos de openButton
48            new ActionListener() {
49
50                // abre o arquivo para processamento
51                public void actionPerformed( ActionEvent event )
52                {
53                    openFile( true );
54                }
55
56            } // fim da classe interna anônima
57
58        ); // fim da chamada para addActionListener
59
60        // cria e configura o botão para obter
61        // as contas com saldo credor
62        creditButton = new JButton( "Credit balances" );
63        buttonPanel.add( creditButton );
64        creditButton.addActionListener( new ButtonHandler() );
65
66        // cria e configura o botão para obter
67        // as contas com saldo devedor
68        debitButton = new JButton( "Debit balances" );
69        buttonPanel.add( debitButton );
70        debitButton.addActionListener( new ButtonHandler() );
71
72        // cria e configura o botão para obter
73        // as contas com saldo zero
74        zeroButton = new JButton( "Zero balances" );
75        buttonPanel.add( zeroButton );
76        zeroButton.addActionListener( new ButtonHandler() );
77
78        // configura a área de exibição
79        recordDisplayArea = new JTextArea();
80        JScrollPane scroller =
81            new JScrollPane( recordDisplayArea );
82
83        // anexa componentes ao painel de conteúdo
84        container.add( scroller, BorderLayout.CENTER );
85        container.add( buttonPanel, BorderLayout.SOUTH );
86
87        // desabilita creditButton, debitButton e zeroButton
88        creditButton.setEnabled( false );

```

Fig. 16.9 Programa de consulta de crédito (parte 2 de 6).

```

89     debitButton.setEnabled( false );
90     zeroButton.setEnabled( false );
91
92     // registra ouvinte para a janela
93     addWindowListener(
94
95         // classe interna anônima para o evento windowClosing
96         new WindowAdapter() {
97
98             // fecha o arquivo e termina o programa
99             public void windowClosing( WindowEvent event )
100            {
101                closeFile();
102                System.exit( 0 );
103            }
104
105        } // fim da classe interna anônima
106
107    ); // fim da chamada para addWindowListener
108
109    // empacota componentes e exibe a janela
110    pack();
111    setSize( 600, 250 );
112    show();
113
114 } // fim do construtor CreditInquiry
115
116 // na primeira vez, permite que o usuário escolha o arquivo a ser aberto;
117 // caso contrário, abre novamente o arquivo escolhido
118 private void openFile( boolean firstTime )
119 {
120     if ( firstTime ) {
121
122         // exibe diálogo para que o usuário possa escolher o arquivo
123         JFileChooser fileChooser = new JFileChooser();
124         fileChooser.setFileSelectionMode(
125             JFileChooser.FILES_ONLY );
126
127         int result = fileChooser.showOpenDialog( this );
128
129         // se o usuário clicou no botão Cancel no diálogo, retorna
130         if ( result == JFileChooser.CANCEL_OPTION )
131             return;
132
133         // obtém arquivo selecionado
134         fileName = fileChooser.getSelectedFile();
135     }
136
137     // exibe erro se o nome de arquivo for inválido
138     if ( fileName == null ||
139         fileName.getName().equals( "" ) )
140         JOptionPane.showMessageDialog( this,
141             "Invalid File Name", "Invalid File Name",
142             JOptionPane.ERROR_MESSAGE );
143
144     else {
145
146         // abre o arquivo
147         try {
148

```

Fig. 16.9 Programa de consulta de crédito (parte 3 de 6).

```

149         // fecha arquivo da operação anterior
150         if ( input != null )
151             input.close();
152
153         fileInput = new FileInputStream( fileName );
154         input = new ObjectInputStream( fileInput );
155         openButton.setEnabled( false );
156         creditButton.setEnabled( true );
157         debitButton.setEnabled( true );
158         zeroButton.setEnabled( true );
159     }
160
161     // captura problemas decorrentes da manipulação do arquivo
162     catch ( IOException ioException ) {
163         JOptionPane.showMessageDialog( this,
164             "File does not exist", "Invalid File Name",
165             JOptionPane.ERROR_MESSAGE );
166     }
167 }
168
169 } // fim do método openFile
170
171 // fecha o arquivo antes que o aplicativo termine
172 private void closeFile()
173 {
174     // fecha o arquivo
175     try {
176         input.close();
177     }
178
179     // processa exceção decorrente do fechamento do arquivo
180     catch ( IOException ioException ) {
181         JOptionPane.showMessageDialog( this,
182             "Error closing file",
183             "Error", JOptionPane.ERROR_MESSAGE );
184
185         System.exit( 1 );
186     }
187 }
188
189 // lê registros do arquivo e exibe só os
190 // registros do tipo apropriado
191 private void readRecords()
192 {
193     AccountRecord record;
194     DecimalFormat twoDigits = new DecimalFormat( "0.00" );
195     openFile( false );
196
197     // lê registros
198     try {
199         recordDisplayArea.setText( "The accounts are:\n" );
200
201         // lê os valores do arquivo
202         while ( true ) {
203
204             // lê um AccountRecord
205             record = ( AccountRecord ) input.readObject();
206
207             // se for o tipo de conta apropriado, exibe o registro
208             if ( shouldDisplay( record.getBalance() ) )

```

Fig. 16.9 Programa de consulta de crédito (parte 4 de 6).

```

209         recordDisplayArea.append( record.getAccount() +
210             "\t" + record.getFirstName() + "\t" +
211             record.getLastName() + "\t" +
212             twoDigits.format( record.getBalance() ) +
213             "\n" );
214     }
215 }
216
217 // fecha o arquivo quando o fim de arquivo é encontrado
218 catch ( EOFException eofException ) {
219     closeFile();
220 }
221
222 // exibe erro se não consegue ler o objeto
223 // porque a classe não foi encontrada
224 catch ( ClassNotFoundException classNotFound ) {
225     JOptionPane.showMessageDialog( this,
226         "Unable to create object",
227         "Class Not Found", JOptionPane.ERROR_MESSAGE );
228 }
229
230 // exibe erro se não consegue ler
231 // devido a um problema com o arquivo
232 catch ( IOException ioException ) {
233     JOptionPane.showMessageDialog( this,
234         "Error reading from file",
235         "Error", JOptionPane.ERROR_MESSAGE );
236 }
237
238 } // fim do método readRecords
239
240 // usa tipo de conta para determinar se um registro deve ser exibido
241 private boolean shouldDisplay( double balance )
242 {
243     if ( accountType.equals( "Credit balances" ) &&
244         balance < 0 )
245
246         return true;
247
248     else if ( accountType.equals( "Debit balances" ) &&
249         balance > 0 )
250
251         return true;
252
253     else if ( accountType.equals( "Zero balances" ) &&
254         balance == 0 )
255
256         return true;
257
258     return false;
259 }
260
261 // executa o aplicativo
262 public static void main( String args[] )
263 {
264     new CreditInquiry();
265 }
266
267 // classe interna privativa para tratamento dos eventos
268 // de creditButton, debitButton e zeroButton

```

Fig. 16.9 Programa de consulta de crédito (parte 5 de 6).

```

269     private class ButtonHandler implements ActionListener {
270
271         // lê registros do arquivo
272         public void actionPerformed( ActionEvent event )
273         {
274             accountType = event.getActionCommand();
275             readRecords();
276         }
277
278     } // fim da classe ButtonHandler
279
280 } // fim da classe CreditInquiry

```

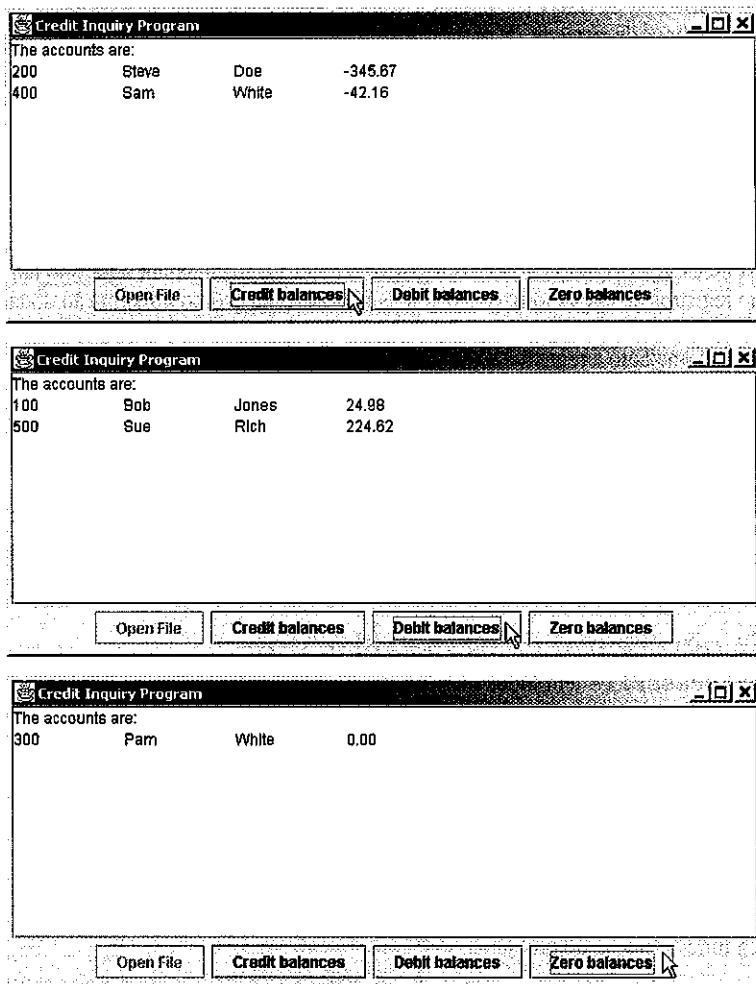


Fig. 16.9 Programa de consulta de crédito (parte 6 de 6).

16.6 Atualizando arquivos de acesso seqüencial

Os dados que são formatados e gravados em um arquivo de acesso seqüencial, como mostrado na Seção 16.4, não podem ser modificados sem se ler e gravar todos os dados do arquivo. Por exemplo, se o nome `White` precisar ser alterado para `Washington`, o nome antigo não pode ser simplesmente sobreescrito. Essa atualização pode ser fei-

ta, mas é algo complicado. Para fazer a alteração de nome precedente, os registros antes de `White` em um arquivo de acesso seqüencial poderiam ser copiados para um novo arquivo, o registro atualizado seria então gravado em um novo arquivo, e os registros depois de `White` seriam copiados para o novo arquivo. Para isso, é preciso processar cada registro no arquivo para atualizar um registro. Se muitos registros estiverem sendo atualizados em uma passagem pelo arquivo, essa técnica pode ser aceitável.

16.7 Arquivos de acesso aleatório

Até agora, vimos como criar arquivos de acesso seqüencial e pesquisar neles para localizar informações em particular. Os arquivos de acesso seqüencial são inadequados para os chamados *aplicativos de “acesso instantâneo”*, em que um registro particular de informações deve ser localizado imediatamente. Entre os aplicativos populares de acesso instantâneo estão os sistemas de reserva de passagens aéreas, sistemas bancários, sistemas de ponto de venda, caixas automáticos e outros tipos de *sistemas de processamento de transações* que exigem acesso rápido a dados específicos. O banco em que você tem sua conta tem centenas de milhares ou até milhões de outros clientes; mesmo quando você utiliza um caixa automático, sua conta é verificada em questão de segundos para determinar se há fundos suficientes para a transação. Esse tipo de acesso instantâneo é possível com os *arquivos de acesso aleatório*. Os registros individuais de um arquivo de acesso aleatório podem ser acessados (rápida e) diretamente sem pesquisar outros registros. Os arquivos de acesso aleatório são às vezes chamados de *arquivos de acesso direto*.

Como dissemos, Java não impõe estrutura a um arquivo. Portanto, o aplicativo que quiser utilizar arquivos de acesso aleatório deve, literalmente, criá-los. Podem-se utilizar várias técnicas para criar arquivos de acesso aleatório. Talvez a mais simples seja exigir que todos os registros em um arquivo tenham o mesmo tamanho fixo.

A utilização de registros de tamanho fixo deixa mais fácil calcular (como uma função do tamanho do registro e da chave de registro) a localização exata de qualquer registro em relação ao início do arquivo. Logo veremos como isso facilita o acesso imediato a registros específicos.

A Fig. 16.10 ilustra a visão de Java de um arquivo de acesso aleatório composto por registros de tamanho fixo (cada registro nessa figura tem 100 bytes de comprimento). O arquivo de acesso aleatório é como um trem de ferrovia com muitos vagões – alguns vazios e alguns com conteúdo.

Os dados podem ser inseridos em um arquivo de acesso aleatório sem destruir outros dados no arquivo. Os dados previamente armazenados também podem ser atualizados ou excluídos sem regravar o arquivo inteiro. Nas seções a seguir, explicamos como criar um arquivo de acesso aleatório, inserir dados, ler os dados tanto no modo seqüencial como no modo aleatório, atualizar os dados e excluir os dados que não são mais necessários.

16.8 Criando um arquivo de acesso aleatório

Os objetos `RandomAccessFile` têm todos os recursos dos objetos `DataInputStream` e `DataOutputStream` discutidos anteriormente. Quando o programa associa um objeto da classe `RandomAccessFile` com um arquivo, o programa lê ou escreve os dados a partir da posição do arquivo especificada pelo ponteiro de posição no arquivo e todos os dados são lidos ou gravados como tipos primitivos de dados. Quando se grava um valor `int`, 4 bytes são enviados para o arquivo de saída. Quando se lê um valor `double`, 8 bytes são recebidos do arquivo de entrada. O tamanho dos tipos de dados é garantido porque Java tem tamanhos fixos para todos os tipos primitivos de dados, independentemente da plataforma de computação.

Os programas de processamento de arquivo de acesso aleatório raramente gravam um único campo em um arquivo. Normalmente, eles gravam um objeto por vez, como mostramos nos seguintes exemplos.

Considere a seguinte definição do problema:

Criar um programa de processamento de transações capaz de armazenar até 100 registros de tamanho fixo, para uma empresa que pode ter até 100 clientes. Cada registro deve consistir em um número de conta, que será utilizado como chave de registro, um sobrenome, um nome e um saldo. O programa deve ser capaz de atualizar uma conta, inserir uma nova e excluir uma conta qualquer.

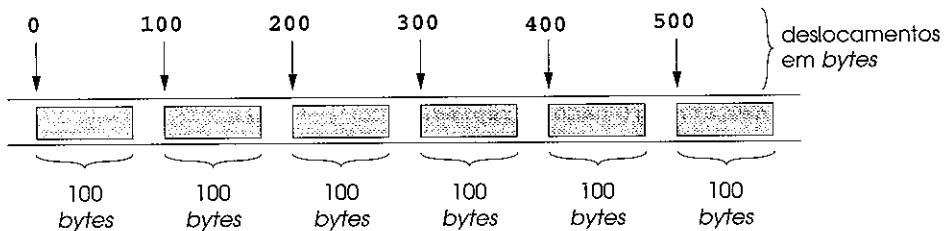


Fig. 16.10 A visão de Java de um arquivo de acesso aleatório.

As seções seguintes apresentam as técnicas necessárias para criar esse programa de processamento de crédito. A Fig. 16.10 contém a classe `RandomAccessAccountRecord` que é utilizada pelos quatro programas seguintes, tanto para ler registros de um arquivo como para gravar registros em um arquivo.

```

1 // Fig. 16.11: RandomAccessAccountRecord.java
2 // Subclasse de AccountRecord para programas com arquivo de acesso aleatório.
3 package com.deitel.jhttp4.ch16;
4
5 // Pacotes do núcleo de Java
6 import java.io.*;
7
8 public class RandomAccessAccountRecord extends AccountRecord {
9
10    // construtor sem argumentos chama outro construtor
11    // com valores default
12    public RandomAccessAccountRecord()
13    {
14        this( 0, "", "", 0.0 );
15    }
16
17    // inicializa um RandomAccessAccountRecord
18    public RandomAccessAccountRecord( int account,
19        String firstName, String lastName, double balance )
20    {
21        super( account, firstName, lastName, balance );
22    }
23
24    // lê um registro do RandomAccessFile especificado
25    public void read( RandomAccessFile file ) throws IOException
26    {
27        setAccount( file.readInt() );
28        setFirstName( padName( file ) );
29        setLastName( padName( file ) );
30        setBalance( file.readDouble() );
31    }
32
33    // assegura que o nome tem o comprimento adequado
34    private String padName( RandomAccessFile file )
35        throws IOException
36    {
37        char name[] = new char[ 15 ], temp;
38

```

Fig. 16.11 Classe `RandomAccessAccountRecord` usada nos programas com arquivo de acesso aleatório (parte 1 de 2).

```

39     for ( int count = 0; count < name.length; count++ ) {
40         temp = file.readChar();
41         name[ count ] = temp;
42     }
43
44     return new String( name ).replace( '\0', ' ' );
45 }
46
47 // grava um registro no RandomAccessFile especificado
48 public void write( RandomAccessFile file ) throws IOException
49 {
50     file.writeInt( getAccount() );
51     writeName( file, getFirstName() );
52     writeName( file, getLastName() );
53     file.writeDouble( getBalance() );
54 }
55
56 // grava um nome no arquivo; máximo de 15 caracteres
57 private void writeName( RandomAccessFile file, String name )
58 throws IOException
59 {
60     StringBuffer buffer = null;
61
62     if ( name != null )
63         buffer = new StringBuffer( name );
64     else
65         buffer = new StringBuffer( 15 );
66
67     buffer.setLength( 15 );
68     file.writeChars( buffer.toString() );
69 }
70
71 // NOTA: Este método contém um valor fixo para o
72 // tamanho de um registro de informações
73 public static int size()
74 {
75     return 72;
76 }
77
78 } // fim da classe RandomAccessAccountRecord

```

Fig. 16.11 Classe `RandomAccessAccountRecord` usada nos programas com arquivo de acesso aleatório (parte 2 de 2).

A classe `RandomAccessAccountRecord` herda a implementação de `AccountRecord` (que inclui as variáveis de instância `private - account, lastName, firstName e balance` – e seus métodos `public get e set`).

O método `read` (linhas 25 a 31) lê um registro do objeto `RandomAccessFile` passado como argumento. Os métodos `readInt` (linha 27) e `readDouble` (linha 30) são utilizados para ler `account` e `balance`, respectivamente. O método `read` faz duas chamadas ao método `private padName` (linhas 34 a 45) para obter o nome e o sobrenome. O método `padName` lê 15 caracteres de `RandomAccessFile` e devolve um `String`. Se um nome é mais curto que 15 caracteres, o programa preenche cada caractere extra com um byte nulo (' \0 '). Os componentes do Swing, como `JTextFields`, não podem exibir caracteres de byte nulo (são exibidos como retângulos). A linha 44 resolve esse problema substituindo `bytes nulos` por espaços.

O método `write` (48 a 54) grava um registro no objeto `RandomAccessFile` passado como argumento. Esse método utiliza o método `writeInt` para gerar como saída o inteiro `account`, o método `writeChars` (chamado pelo método utilitário `writeName`) para dar saída aos `arrays` de caracteres `firstName` e `lastName` e o método `writeDouble` para dar saída ao `double balance`. [Nota: a fim de assegurar que todos os registros no

`RandomAccessFile` tenham o mesmo tamanho, escrevemos exatamente 15 caracteres para o nome e exatamente 15 caracteres para o sobrenome.] O método `writeName` (linhas 57 a 69) realiza as operações de gravação para o nome e o sobrenome.

A Fig. 16.12 ilustra a abertura de um arquivo de acesso aleatório e a gravação dos dados em disco. Esse programa grava 100 `RandomAccessAccountRecords` utilizando o método `write` (Fig. 16.11). Cada objeto `RandomAccessAccountRecord` contém 0 para o número de conta, `null` para o sobrenome, `null` para o nome e 0.0 para o saldo. O arquivo é inicializado para criar a quantidade adequada de espaço “vazio” em que os dados da conta serão armazenados e para permitir que determinemos nos programas subsequentes se cada registro está vazio ou contém dados.

```

1 // Fig. 16.12: CreateRandomFile.java
2 // Este programa cria um arquivo de acesso aleatório seqüencialmente,
3 // escrevendo 100 registros vazios no disco.
4
5 // Pacotes do núcleo de Java
6 import java.io.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 // Pacotes Deitel
12 import com.deitel.jhtp4.ch16.RandomAccessAccountRecord;
13
14 public class CreateRandomFile {
15
16     // permite que o usuário selecione um arquivo a ser aberto
17     private void createFile()
18     {
19         // exibe diálogo para que o usuário possa escolher o arquivo
20         JFileChooser fileChooser = new JFileChooser();
21         fileChooser.setFileSelectionMode(
22             JFileChooser.FILES_ONLY );
23
24         int result = fileChooser.showSaveDialog( null );
25
26         // se o usuário clicou no botão Cancel no diálogo, retorna
27         if ( result == JFileChooser.CANCEL_OPTION )
28             return;
29
30         // obtém o arquivo selecionado
31         File fileName = fileChooser.getSelectedFile();
32
33         // exibe erro se o nome de arquivo for inválido
34         if ( fileName == null ||
35             fileName.getName().equals( "" ) )
36             JOptionPane.showMessageDialog( null,
37                 "Invalid File Name", "Invalid File Name",
38                 JOptionPane.ERROR_MESSAGE );
39
40     else {
41
42         // abre arquivo
43         try {
44             RandomAccessFile file =
45                 new RandomAccessFile( fileName, "rw" );
46
47             RandomAccessAccountRecord blankRecord =
48                 new RandomAccessAccountRecord();

```

Fig. 16.12 Criando um arquivo de acesso aleatório seqüencialmente (parte 1 de 3).

```

49
50     // escreve 100 registros em branco
51     for ( int count = 0; count < 100; count++ )
52         blankRecord.write( file );
53
54     // fecha o arquivo
55     file.close();
56
57     // exibe mensagem dizendo que o arquivo foi criado
58     JOptionPane.showMessageDialog( null,
59         "Created file " + fileName, "Status",
60         JOptionPane.INFORMATION_MESSAGE );
61
62     System.exit( 0 );    // termina o programa
63 }
64
65 // processa exceções durante as operações de
66 // abrir, gravar ou fechar o arquivo
67 catch ( IOException ioException ) {
68     JOptionPane.showMessageDialog( null,
69         "Error processing file", "Error processing file",
70         JOptionPane.ERROR_MESSAGE );
71
72     System.exit( 1 );
73 }
74 }
75
76 } // fim do método openFile
77
78 // executa o aplicativo para criar o arquivo que o usuário especifica
79 public static void main( String args[] )
80 {
81     CreateRandomFile application = new CreateRandomFile();
82
83     application.createFile();
84 }
85
86 } // fim da classe CreateRandomFile

```

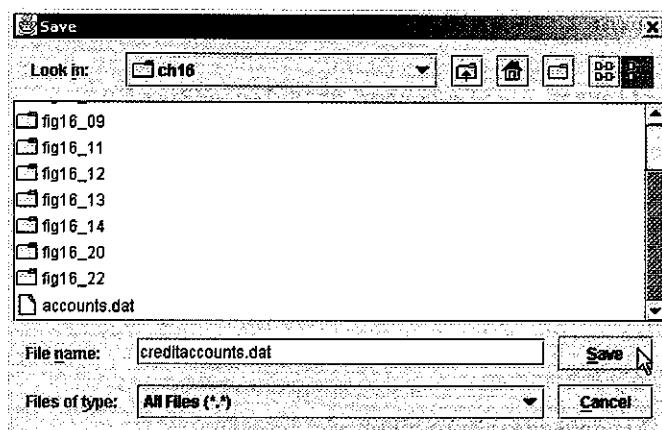


Fig. 16.12 Criando um arquivo de acesso aleatório seqüencialmente (parte 2 de 3).

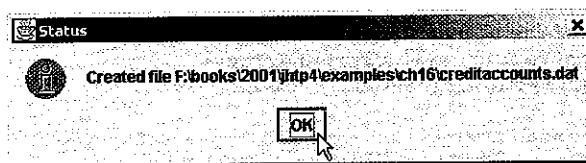


Fig. 16.12 Criando um arquivo de acesso aleatório seqüencialmente (parte 3 de 3).

As linhas 44 e 45 tentam abrir um `RandomAccessFile` para utilização nesse programa. Dois argumentos são passados para o construtor `RandomAccessFile` – o nome do arquivo e o *modo de abertura do arquivo*. O modo de abertura de arquivo para um `RandomAccessFile` é “`r`” para abrir o arquivo para leitura ou “`rw`” para abrir o arquivo para leitura e gravação.

Se ocorrer uma `IOException` durante o processo de abertura, é exibido um diálogo de mensagem e o programa é finalizado. Se o arquivo é aberto adequadamente, o programa utiliza uma estrutura `for` (linhas 51 e 52) para invocar o método `Write` de `RandomAccessAccountRecord` 100 vezes. Essa instrução faz com que os membros de dados do objeto `blankRecord` gravem no arquivo associado com o objeto `file` de `RandomAccessFile` file.

16.9 Gravando dados aleatoriamente em um arquivo de acesso aleatório

A Fig. 16.13 grava dados em um arquivo que é aberto com o modo “`rw`” de leitura e gravação. Utiliza-se o método `seek` de `RandomAccessFile` para determinar a localização exata no arquivo em que um registro de informações está armazenado. O método `seek` configura o ponteiro de posição para uma posição específica no arquivo em relação ao início do arquivo, e o método `write` da classe `RandomAccessFile` envia os dados para saída. Esse programa pressupõe que o usuário não irá digitar números de conta duplicados e que ele digitará dados apropriados em cada `JTextField`.

```

1 // Fig. 16.13: WriteRandomFile.java
2 // Este programa usa campos de texto para obter informações do usuário
3 // que está usando o teclado e escreve as informações em um
4 // arquivo de acesso aleatório.
5
6 // Pacotes do núcleo de Java
7 import java.awt.*;
8 import java.awt.event.*;
9 import java.io.*;
10
11 // Pacotes de extensão de Java
12 import javax.swing.*;
13
14 // Pacotes Deitel
15 import com.deitel.jhtp4.ch16.*;
16
17 public class WriteRandomFile extends JFrame {
18     private RandomAccessFile output;
19     private BankUI userInterface;
20     private JButton enterButton, openButton;
21
22     // configura a GUI
23     public WriteRandomFile()
24     {
25         super( "Write to random access file" );
26

```

Fig. 16.13 Gravando dados aleatoriamente em um arquivo de acesso aleatório (parte 1 de 5).

```

27      // cria instância da interface reutilizável com o usuário BankUI
28      userInterface = new BankUI( 4 );    // quatro campos de texto
29      getContentPane().add( userInterface,
30          BorderLayout.CENTER );
31
32      // obtém referência para o botão de tarefa genérico doTask1 em BankUI
33      openButton = userInterface.getDoTask1Button();
34      openButton.setText( "Open..." );
35
36      // registra ouvinte para chamar openFileDialog quando o botão é pressionado
37      openButton.addActionListener(
38
39          // classe interna anônima para tratar eventos de openButton
40          new ActionListener() {
41
42              // permite que o usuário selecione um arquivo para ser aberto
43              public void actionPerformed( ActionEvent event )
44              {
45                  openFileDialog();
46              }
47
48          } // fim da classe interna anônima
49
50      ); // fim da chamada para addActionListener
51
52      // registra ouvinte da janela para o evento de fechamento de janela
53      addWindowListener(
54
55          // classe interna anônima para tratar do evento windowClosing
56          new WindowAdapter() {
57
58              // adiciona registro à GUI, depois fecha o arquivo
59              public void windowClosing( WindowEvent event )
60              {
61                  if ( output != null )
62                      addRecord();
63
64                  closeFile();
65              }
66
67          } // fim da classe interna anônima
68
69      ); // fim da chamada para addWindowListener
70
71      // obtém referência para o botão de tarefa genérico doTask2 em BankUI
72      enterButton = userInterface.getDoTask2Button();
73      enterButton.setText( "Enter" );
74      enterButton.setEnabled( false );
75
76      // registra ouvinte para chamar addRecord quando o botão é pressionado
77      enterButton.addActionListener(
78
79          // classe interna anônima para tratar eventos de enterButton
80          new ActionListener() {
81
82              // adiciona registro ao arquivo
83              public void actionPerformed( ActionEvent event )
84              {
85                  addRecord();
86              }

```

Fig. 16.13 Gravando dados aleatoriamente em um arquivo de acesso aleatório (parte 2 de 5).

```

87         } // fim da classe interna anônima
88
89     ); // fim da chamada para addActionListener
90
91     setSize( 300, 150 );
92     show();
93 }
94
95 // permite que o usuário escolha o arquivo para ser aberto
96 private void openFile()
97 {
98     // exibe diálogo de arquivo para que o usuário possa selecionar o arquivo
99     JFileChooser fileChooser = new JFileChooser();
100    fileChooser.setFileSelectionMode(
101        JFileChooser.FILES_ONLY );
102
103    int result = fileChooser.showOpenDialog( this );
104
105    // se o usuário clicou no botão Cancel no diálogo, retorna
106    if ( result == JFileChooser.CANCEL_OPTION )
107        return;
108
109    // obtém arquivo selecionado
110    File fileName = fileChooser.getSelectedFile();
111
112    // exibe erro se o nome do arquivo for inválido
113    if ( fileName == null ||
114        fileName.getName().equals( "" ) )
115        JOptionPane.showMessageDialog( this,
116            "Invalid File Name", "Invalid File Name",
117            JOptionPane.ERROR_MESSAGE );
118
119    else {
120
121        // abre arquivo
122        try {
123            output = new RandomAccessFile( fileName, "rw" );
124            enterButton.setEnabled( true );
125            openButton.setEnabled( false );
126        }
127
128        // processa exceção ocorrida enquanto estava abrindo o arquivo
129        catch ( IOException ioException ) {
130            JOptionPane.showMessageDialog( this,
131                "File does not exist",
132                "Invalid File Name",
133                JOptionPane.ERROR_MESSAGE );
134        }
135    }
136
137 }
138 // fim do método openFile
139
140 // fecha o arquivo e termina o aplicativo
141 private void closeFile()
142 {
143     // fecha o arquivo e sai
144     try {
145         if ( output != null )
146             output.close();

```

Fig. 16.13 Gravando dados aleatoriamente em um arquivo de acesso aleatório (parte 3 de 5).

```

147         System.exit( 0 );
148     }
149 }
150
151 // processa exceção ocorrida enquanto fechava o arquivo
152 catch( IOException ioException ) {
153     JOptionPane.showMessageDialog( this,
154         "Error closing file",
155         "Error", JOptionPane.ERROR_MESSAGE );
156
157     System.exit( 1 );
158 }
159 }
160
161 // adiciona um registro ao arquivo
162 public void addRecord()
163 {
164     int accountNumber = 0;
165     String fields[] = userInterface.getFieldValues();
166     RandomAccessAccountRecord record =
167         new RandomAccessAccountRecord();
168
169     // assegura que o campo de conta tem um valor
170     if ( ! fields[ BankUI.ACCOUNT ].equals( "" ) ) {
171
172         // envia valores para saída no arquivo
173         try {
174             accountNumber =
175                 Integer.parseInt( fields[ BankUI.ACCOUNT ] );
176
177             if ( accountNumber > 0 && accountNumber <= 100 ) {
178                 record.setAccount( accountNumber );
179
180                 record.setFirstName( fields[ BankUI.FIRSTNAME ] );
181                 record.setLastName( fields[ BankUI.LASTNAME ] );
182                 record.setBalance( Double.parseDouble(
183                     fields[ BankUI.BALANCE ] ) );
184
185                 output.seek( ( accountNumber - 1 ) *
186                         RandomAccessAccountRecord.size() );
187                 record.write( output );
188             }
189
190             userInterface.clearFields();    // limpa os TextFields
191         }
192
193         // processa número de conta ou formato do saldo inadequados
194         catch ( NumberFormatException formatException ) {
195             JOptionPane.showMessageDialog( this,
196                 "Bad account number or balance",
197                 "Invalid Number Format",
198                 JOptionPane.ERROR_MESSAGE );
199         }
200
201         // processa exceções ocorridas enquanto estava gravando no arquivo
202         catch ( IOException ioException ) {
203             closeFile();
204         }
205     }
206 }
```

Fig. 16.13 Gravando dados aleatoriamente em um arquivo de acesso aleatório (parte 4 de 5).

```

207     } // fim do método addRecord
208
209     // executa o aplicativo
210     public static void main( String args[] )
211     {
212         new WriteRandomFile();
213     }
214
215 } // fim da classe WriteRandomFile

```

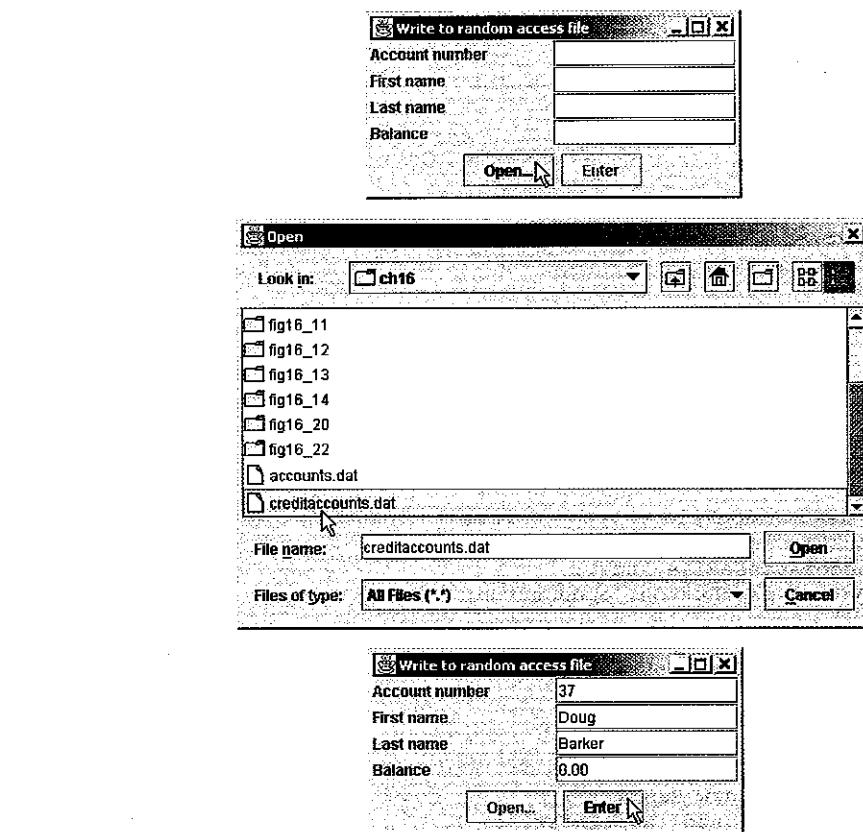


Fig. 16.13 Gravando dados aleatoriamente em um arquivo de acesso aleatório (parte 5 de 5).

O usuário digita valores para o número de conta, o nome, o sobrenome e o saldo. Quando o usuário clica no botão **Enter**, o método `addRecord` da classe `WriteRandomFile` (linhas 162 a 207) é invocado pelo programa para recuperar os dados dos `JTextFields` de `BankAccountUI`, armazenar os dados em um objeto `record` da classe `RandomAccessAccountRecord` e chamar o método `write` da classe `RandomAccessAccountRecord` para enviar os dados para a saída.

As linhas 185 e 186 chamam o método `seek` de `RandomAccessFile` para posicionar o ponteiro de posição no arquivo `output` na posição de `byte` calculada por `(accountNumber - 1) * RandomAccessAccountRecord.size()`. Como o número de conta está entre 1 e 100, 1 é subtraído do número de conta ao se calcular a posição do `byte` de registro. Portanto, para o registro 1, o ponteiro de posição no arquivo é estabelecido como o byte 0 do arquivo.

Quando o usuário fecha a janela, o programa tenta adicionar o último registro ao arquivo (se houver algum esperando para ser enviado para saída), fecha o arquivo e termina.

16.10 Lendo dados seqüencialmente de um arquivo de acesso aleatório

Nas seções anteriores, criamos um arquivo de acesso aleatório e gravamos dados nesse arquivo. Nesta seção, desen- volveremos um programa (Fig. 16.14) que abre um `RandomAccessFile` para leitura com o modo de abertura de arquivo "r", lê o arquivo seqüencialmente e exibe só os registros que contêm dados. Esse programa produz um benefício adicional. Veja se você consegue determinar qual é; nós o revelaremos no final desta seção.



Boa prática de programação 16.1

Se o conteúdo do arquivo não deve ser modificado, abra um arquivo com o modo de abertura de arquivo "r" para entrada. Isso evita modificação não-intencional do conteúdo do arquivo. Esse é outro exemplo do princípio do menor privilégio.

```

1 // Fig. 16.14: ReadRandomFile.java
2 // Este programa lê um arquivo de acesso aleatório seqüencialmente e
3 // exibe o conteúdo de um registro por vez em campos de texto.
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8 import java.io.*;
9 import java.text.DecimalFormat;
10
11 // Pacotes de extensão de Java
12 import javax.swing.*;
13
14 // Pacotes Deitel
15 import com.deitel.jhtp4.ch16.*;
16
17 public class ReadRandomFile extends JFrame {
18     private BankUI userInterface;
19     private RandomAccessFile input;
20     private JButton nextButton, openButton;
21
22     // configura a GUI
23     public ReadRandomFile()
24     {
25         super( "Read Client File" );
26
27         // cria instância reutilizável da interface com o usuário
28         userInterface = new BankUI( 4 );    // quatro campos de texto
29         getContentPane().add( userInterface );
30
31         // configura botão de tarefa genérico doTask1 de BankUI
32         openButton = userInterface.getDoTask1Button();
33         openButton.setText( "Open File for Reading..." );
34
35         // registra ouvinte para chamar openFileDialog quando o botão é pressionado
36         openButton.addActionListener(
37
38             // classe interna anônima para tratar eventos de openButton
39             new ActionListener() {
40
41                 // permite que o usuário selecione o arquivo a ser aberto
42                 public void actionPerformed( ActionEvent event )
43                 {
44                     openFileDialog();
45                 }
46             } // fim da classe interna anônima
47     }

```

Fig. 16.14 Lendo um arquivo de acesso aleatório seqüencialmente (parte 1 de 5).

```

48      ); // fim da chamada para addActionListener
49
50      // configura o botão de tarefa genérico doTask2 de BankUI
51      nextButton = userInterface.getDoTask2Button();
52      nextButton.setText( "Next" );
53      nextButton.setEnabled( false );
54
55      // registra ouvinte para chamar readRecord quando o botão é pressionado
56      nextButton.addActionListener(
57
58          // classe interna anônima para tratar de eventos de nextButton
59          new ActionListener() {
60
61              // lê um registro quando o usuário clica em nextButton
62              public void actionPerformed( ActionEvent event )
63              {
64                  readRecord();
65              }
66
67          } // fim da classe interna anônima
68
69      ); // fim da chamada para addActionListener
70
71      // registra ouvinte para evento de fechamento da janela
72      addWindowListener(
73
74          // classe interna anônima para tratar evento windowClosing
75          new WindowAdapter() {
76
77              // fecha o arquivo e termina o aplicativo
78              public void windowClosing( WindowEvent event )
79              {
80                  closeFile();
81              }
82
83          } // fim da classe interna anônima
84
85      ); // fim da chamada para addWindowListener
86
87      setSize( 300, 150 );
88      show();
89  }
90
91      // permite que o usuário selecione o arquivo a ser aberto
92      private void openFile()
93  {
94
95          // exibe o diálogo de arquivo para o usuário selecionar o arquivo
96          JFileChooser fileChooser = new JFileChooser();
97          fileChooser.setFileSelectionMode(
98              JFileChooser.FILES_ONLY );
99
100         int result = fileChooser.showOpenDialog( this );
101
102         // se o usuário clicou no botão Cancel no diálogo, retorna
103         if ( result == JFileChooser.CANCEL_OPTION )
104             return;
105
106         // obtém arquivo selecionado
107         File fileName = fileChooser.getSelectedFile();

```

Fig. 16.14 Lendo um arquivo de acesso aleatório seqüencialmente (parte 2 de 5).

```

108
109     // exibe erro se o nome do arquivo for inválido
110    if ( fileName == null ||
111        fileName.getName().equals( "" ) )
112        JOptionPane.showMessageDialog( this,
113            "Invalid File Name", "Invalid File Name",
114            JOptionPane.ERROR_MESSAGE );
115
116    else {
117
118        // abre o arquivo
119        try {
120            input = new RandomAccessFile( fileName, "r" );
121            nextButton.setEnabled( true );
122            openButton.setEnabled( false );
123        }
124
125        // captura exceção ocorrida durante a abertura do arquivo
126        catch ( IOException ioException ) {
127            JOptionPane.showMessageDialog( this,
128                "File does not exist", "Invalid File Name",
129                JOptionPane.ERROR_MESSAGE );
130        }
131    }
132
133 } // fim do método openFile
134
135 // lê um registro
136 public void readRecord()
137 {
138     DecimalFormat twoDigits = new DecimalFormat( "0.00" );
139     RandomAccessAccountRecord record =
140         new RandomAccessAccountRecord();
141
142     // lê um registro e exibe
143     try {
144
145         do {
146             record.read( input );
147         } while ( record.getAccount() == 0 );
148
149         String values[] = {
150             String.valueOf( record.getAccount() ),
151             record.getFirstName(),
152             record.getLastName(),
153             String.valueOf( record.getBalance() ) };
154         userInterface.setFieldValues( values );
155     }
156
157     // fecha o arquivo quando é encontrado o fim do arquivo
158     catch ( EOFException eofException ) {
159         JOptionPane.showMessageDialog( this, "No more records",
160             "End-of-file reached",
161             JOptionPane.INFORMATION_MESSAGE );
162         closeFile();
163     }
164
165     // processa exceções ocorridas por problemas com o arquivo
166     catch ( IOException ioException ) {
167         JOptionPane.showMessageDialog( this,
168             "Error Reading File", "Error",

```

Fig. 16.14 Lendo um arquivo de acesso aleatório seqüencialmente (parte 3 de 5).

```

169         JOptionPane.ERROR_MESSAGE );
170
171     System.exit( 1 );
172 }
173
174 } // fim do método readRecord
175
176 // fecha o arquivo e termina o aplicativo
177 private void closeFile()
178 {
179     // fecha o arquivo e sai
180     try {
181         if ( input != null )
182             input.close();
183
184         System.exit( 0 );
185     }
186
187     // processa exceções ocorridas enquanto está fechando o arquivo
188     catch( IOException ioException ) {
189         JOptionPane.showMessageDialog( this,
190             "Error closing file",
191             "Error", JOptionPane.ERROR_MESSAGE );
192
193         System.exit( 1 );
194     }
195 }
196
197 // executa o aplicativo
198 public static void main( String args[] )
199 {
200     new ReadRandomFile();
201 }
202
203 } // fim da classe ReadRandomFile

```

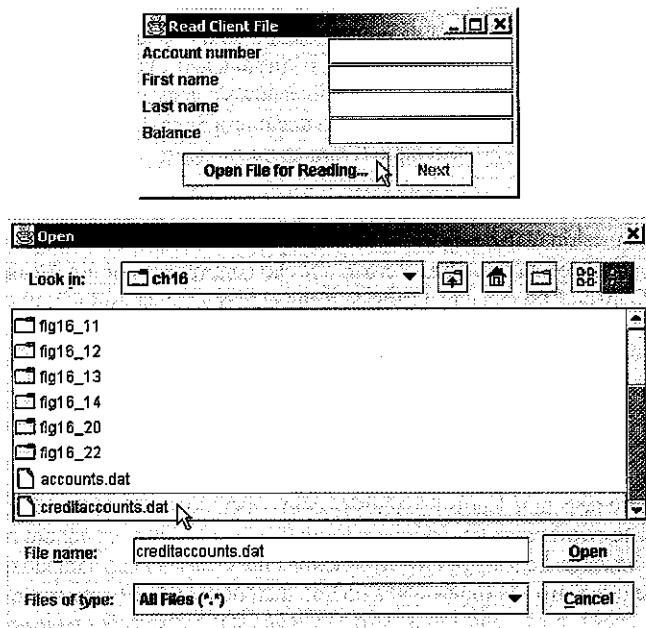


Fig. 16.14 Lendo um arquivo de acesso aleatório seqüencialmente (parte 4 de 5).

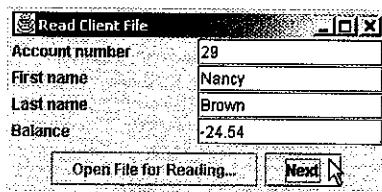


Fig. 16.14 Lendo um arquivo de acesso aleatório seqüencialmente (parte 4 de 5).

Quando o usuário clica no botão **Next** para ler o próximo registro no arquivo, o método `readRecord` da classe `ReadRandomFile` (linhas 136 a 174) é invocado. Esse método invoca o método `read` da classe `RandomAccessAccountRecord`. O método `readRecord` lê o arquivo até encontrar um registro com um número de conta diferente de zero (0 é o valor inicial de todas as contas). Quando um número de conta válido (isto é, um valor não zero) é lido, o laço termina e `readRecord` exibe os dados do registro nos campos de texto. Quando o usuário clica no botão **Done**, ou quando o marcador de fim do arquivo é encontrado durante a leitura, o método `closeFile` é invocado para fechar o arquivo e terminar o programa.

E quanto àquele benefício adicional que prometemos? Se você examinar a GUI ao executar o programa, perceberá que os registros são exibidos em ordem de classificação (por número de conta)! Isso é uma simples consequência da maneira como armazenamos esses registros no arquivo, utilizando técnicas de acesso direto. Comparado à classificação de bolhas (*bubble sort*) que vimos (Capítulo 7), classificar com técnicas de acesso direto é extremamente rápido. A velocidade é alcançada tornando-se o arquivo suficientemente grande para armazenar cada registro possível que vier a ser criado, o que permite a um programa inserir um registro entre outros registros sem ter que reorganizar o arquivo. Isso, naturalmente, significa que o arquivo poderia estar esparsamente ocupado a maior parte do tempo, o que é um desperdício do espaço de armazenamento. Então, eis ainda outro exemplo do compromisso entre tempo e espaço. Por utilizar grandes quantidades de espaço, somos capazes de desenvolver um algoritmo de classificação muito mais rápido.

16.11 Exemplo: um programa de processamento de transações

Apresentamos agora um programa substancial de processamento de transações (Fig. 16.20) utilizando um arquivo de acesso aleatório para realizar o processamento de acesso “instantâneo”. O programa mantém as informações de contas de um banco. Ele atualiza as contas existentes, adiciona novas contas e exclui outras. Pressupomos que o programa da Fig. 16.12 foi executado para criar um arquivo e que o programa da Fig. 16.13 foi executado para inserir os dados iniciais. As técnicas usadas neste exemplo foram apresentadas nos exemplos anteriores de `RandomAccessFile`.

A GUI desse programa consiste em uma janela com uma barra de menu que contém um menu **File** e *frames* internas que permitem a um usuário realizar operações de inserção, atualização e eliminação de registros no arquivo. As *frames* internas são subclasses `JInternalFrame` que são controladas por um `JDesktopPane` (como discutido no Capítulo 13). O menu **File** tem cinco itens de menu para selecionar as diferentes tarefas, como mostrado na Fig. 16.15.

Quando o usuário seleciona **Update Record** do menu **File**, a *frame* interna de **Update Record** (Fig. 16.16) permite ao usuário atualizar uma conta existente. O código que implementa a *frame* interna de **Update Record** está na classe `UpdateDialog` (linhas 238 a 258 da Fig. 16.20). Na primeira captura de tela da Fig. 16.16, o usuário digita o número da conta e pressiona *Enter* para invocar o método `actionPerformed` (linhas 345 a 360 da Fig. 16.20) do campo de texto de conta. Isto lê a conta do arquivo com o método `getRecord` (linhas 371 a 418 da Fig. 16.20), que valida o número de conta, depois lê o registro com o método `read` de `RandomAccessAccountRecord`. Em seguida, `getRecord` compara o número de conta com zero (isto é, nenhum registro) para determinar se o registro contém informações. Se não, `getRecord` exibe uma mensagem declarando que o registro não existe; do contrário, ele devolve o registro. As linhas 350 a 357 do método `actionPerformed` do campo de texto da conta extraem as informações sobre a conta do registro e exibem estas informações na *frame* interna (como mostrado na segunda captura de tela da Fig. 16.16). O campo de texto **Transaction Amount** inicialmente contém o

string charge (+) or payment (-). O usuário deve selecionar esse texto e digitar a quantia da transação (um valor positivo para um débito ou um valor negativo para um pagamento) e pressionar *Enter* para invocar o método `actionPerformed` do campo de texto de transação (linhas 295 a 330 da Fig. 16.20). O método `addRecord` (linhas 421 a 456 da Fig. 16.20) recebe a quantia da transação, adiciona essa quantia ao saldo atual e chama o método `setBalance` de `RandomAccessAccountRecord` para atualizar a exibição. Clicar em **Save Changes** grava o registro atualizado em disco; clicar em **Cancel** fecha a *frame* interna sem gravar o registro em disco. A janela da Fig. 16.7 mostra um exemplo de entrada de uma transação.

Quando o usuário seleciona **New Record** do menu **File** a *frame* interna de **New Record** na Fig. 16.18 permite acrescentar um novo registro. O código que implementa a *frame* interna de **New Record** está na classe `NewDialog` (linhas 461 a 615 da Fig. 16.20). O usuário insere dados nos `JTextFields` e clica em **Save Changes** para gravar o registro em disco. Se o número de conta já existir, o programa exibe uma mensagem de erro e não tenta gravar o registro. Clicar em **Cancel** fecha a *frame* interna sem tentar gravar o registro.

Selecionar **Delete Record** do menu **File** exibe a *frame* interna **Delete Record** na Fig. 16.19, que permite excluir um registro do arquivo. O código que implementa a *frame* interna de **Delete Record** está na classe `DeleteDialog` (linhas 618 a 774 da Fig. 16.20). O usuário insere o número de conta no `JTextField` e pressiona *Enter*. Somente um registro existente pode ser excluído; se a conta especificada estiver vazia, o programa exibe uma mensagem de erro. Clicar no botão **Delete Record** na *frame* interna configura o número da conta de registro como 0 (que esse aplicativo considera um registro vazio). Clicar em **Cancel** fecha a *frame* interna sem tentar excluir o registro.

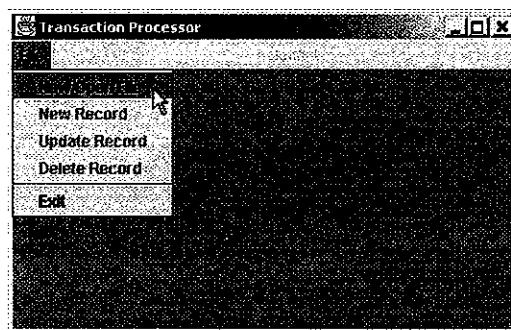


Fig. 16.15 A janela inicial do Transaction Processor.

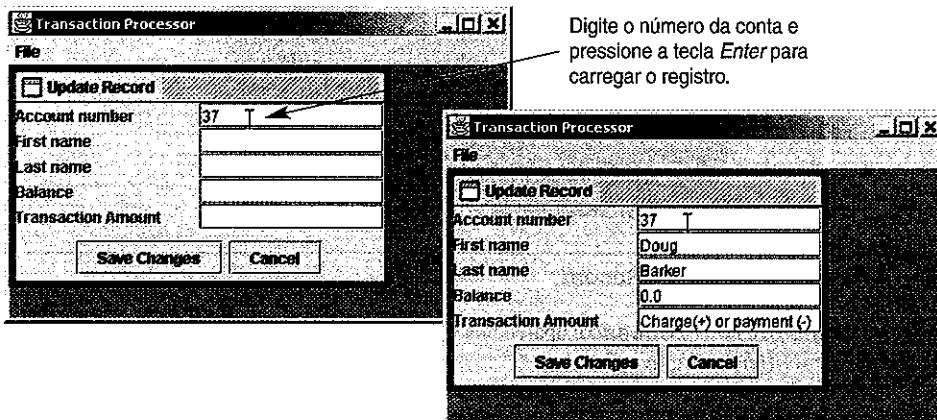
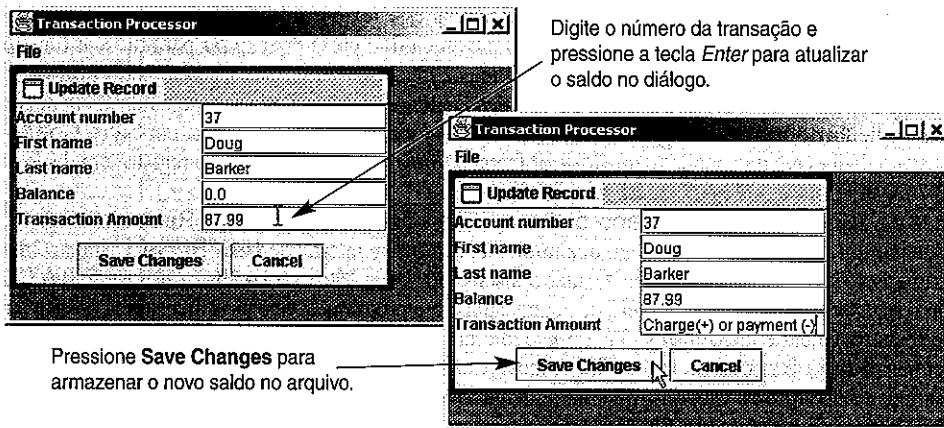
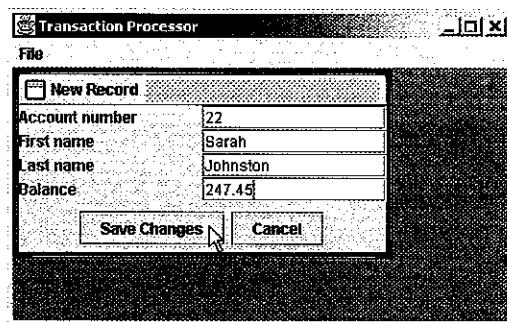
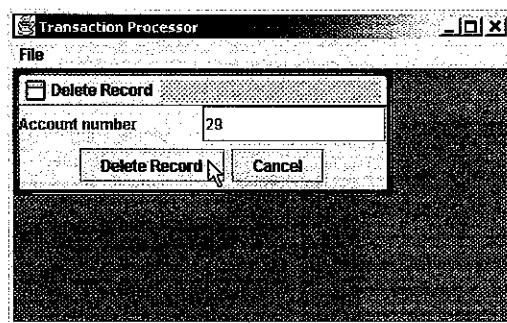


Fig. 16.16 Carregando um registro para a *frame* interna **Update Record**.

Fig. 16.17 Lendo uma transação na *frame* interna **Update Record**.Fig. 16.18 A *frame* interna **New Record**.Fig. 16.19 A *frame* interna **Delete Record**.

O programa **TransactionProcessor** aparece na Fig. 16.20. O programa abre o arquivo com o modo de abertura de arquivo "rw" (leitura e gravação).

```

1 // Programa de processamento de transações usando RandomAccessFiles.
2 // Este programa lê um arquivo de acesso aleatório seqüencialmente,
```

Fig. 16.20 Programa de processamento de transações (parte 1 de 14).

```

3 // atualiza registros já gravados no arquivo, cria novos
4 // registros a serem colocados no arquivo e apaga os dados que
5 // já estão no arquivo.
6
7 // Pacotes do núcleo de Java
8 import java.awt.*;
9 import java.awt.event.*;
10 import java.io.*;
11 import java.text.DecimalFormat;
12
13 // Pacotes de extensão de Java
14 import javax.swing.*;
15
16 // Pacotes Deitel
17 import com.deitel.jhtp4.ch16.*;
18
19 public class TransactionProcessor extends JFrame {
20     private UpdateDialog updateDialog;
21     private NewDialog newDialog;
22     private DeleteDialog deleteDialog;
23     private JMenuItem newItem, updateItem, deleteItem,
24         openItem, exitItem;
25     private JDesktopPane desktop;
26     private RandomAccessFile file;
27     private RandomAccessAccountRecord record;
28
29     // configura a GUI
30     public TransactionProcessor()
31     {
32         super( "Transaction Processor" );
33
34         // configura a área de trabalho, a barra de menus e o menu File
35         desktop = new JDesktopPane();
36         getContentPane().add( desktop );
37
38         JMenuBar menuBar = new JMenuBar();
39         setJMenuBar( menuBar );
40
41         JMenu fileMenu = new JMenu( "File" );
42         menuBar.add( fileMenu );
43
44         // configura o item de menu para adicionar um registro
45         newItem = new JMenuItem( "New Record" );
46         newItem.setEnabled( false );
47
48         // exibe um diálogo para novo registro quando o usuário seleciona New Record
49         newItem.addActionListener(
50
51             new ActionListener() {
52
53                 public void actionPerformed( ActionEvent event )
54                 {
55                     newDialog.setVisible( true );
56                 }
57             }
58         );
59
60         // configura item de menu para atualizar um registro
61         updateItem = new JMenuItem( "Update Record" );
62         updateItem.setEnabled( false );

```

Fig. 16.20 Programa de processamento de transações (parte 2 de 14).

```
63
64     // exibe um diálogo de atualização quando o usuário seleciona Update Record
65     updateItem.addActionListener(
66
67         new ActionListener() {
68
69             public void actionPerformed( ActionEvent event )
70             {
71                 updateDialog.setVisible( true );
72             }
73         };
74     );
75
76     // configura item de menu para apagar um registro
77     deleteItem = new JMenuItem( "Delete Record" );
78     deleteItem.setEnabled( false );
79
80     // exibe diálogo de apagar registro quando usuário seleciona Delete Record
81     deleteItem.addActionListener(
82
83         new ActionListener() {
84
85             public void actionPerformed( ActionEvent event )
86             {
87                 deleteDialog.setVisible( true );
88             }
89         };
90     );
91
92     // configura botão para abertura do arquivo
93     openItem = new JMenuItem( "New/Open File" );
94
95     // permite que o usuário selecione o arquivo a ser aberto,
96     // depois configura caixas de diálogo
97     openItem.addActionListener(
98
99         new ActionListener() {
100
101            public void actionPerformed( ActionEvent event )
102            {
103                boolean opened = openFileDialog();
104
105                if ( !opened )
106                    return;
107
108                openItem.setEnabled( false );
109
110                // configura frames internas para processamento de registro
111                updateDialog = new UpdateDialog( file );
112                desktop.add( updateDialog );
113
114                deleteDialog = new DeleteDialog( file );
115                desktop.add ( deleteDialog );
116
117                newDialog = new NewDialog( file );
118                desktop.add( newDialog );
119            }
120
121        } // fim da classe interna anônima
122
```

Fig. 16.20 Programa de processamento de transações (parte 3 de 14).

```

123      ); // fim da chamada para addActionListener
124
125      // configura item de menu para sair do programa
126      exitItem = new JMenuItem( "Exit" );
127      exitItem.setEnabled( true );
128
129      // termina aplicativo
130      exitItem.addActionListener(
131
132          new ActionListener() {
133
134              public void actionPerformed( ActionEvent event ) {
135                  {
136                      closeFile();
137                  }
138              }
139          );
140
141      // anexa itens de menu ao menu File
142      fileMenu.add( openItem );
143      fileMenu.add( newItem );
144      fileMenu.add( updateItem );
145      fileMenu.add( deleteItem );
146      fileMenu.addSeparator();
147      fileMenu.add( exitItem );
148
149      // configura a janela
150      setDefaultCloseOperation(
151          WindowConstants.DO NOTHING ON CLOSE );
152
153      setSize( 400, 250 );
154      setVisible( true );
155
156 } // fim do construtor TransactionProcessor
157
158 // permite que o usuário selecione o arquivo a ser aberto
159 private boolean openFileDialog()
160 {
161     // exibe diálogo para que o usuário escolha o arquivo
162     JFileChooser fileChooser = new JFileChooser();
163     fileChooser.setFileSelectionMode(
164         JFileChooser.FILES_ONLY );
165
166     int result = fileChooser.showOpenDialog( this );
167
168     // se o usuário clicou no botão Cancel no diálogo, retorna
169     if ( result == JFileChooser.CANCEL_OPTION )
170         return false;
171
172     // obtém arquivo selecionado
173     File fileName = fileChooser.getSelectedFile();
174
175     // exibe erro se o nome de arquivo for inválido
176     if ( fileName == null ||
177         fileName.getName().equals( "" ) ) {
178         JOptionPane.showMessageDialog( this,
179             "Invalid File Name", "Invalid File Name",
180             JOptionPane.ERROR_MESSAGE );
181
182         return false;

```

Fig. 16.20 Programa de processamento de transações (parte 4 de 14).

```

183     }
184
185     else {
186
187         // abre o arquivo
188         try {
189             file = new RandomAccessFile( fileName, "rw" );
190             openItem.setEnabled( false );
191             newItem.setEnabled( true );
192             updateItem.setEnabled( true );
193             deleteItem.setEnabled( true );
194         }
195
196         // processa os problemas durante a abertura do arquivo
197         catch ( IOException ioException ) {
198             JOptionPane.showMessageDialog( this,
199                 "File does not exist", "Invalid File Name",
200                 JOptionPane.ERROR_MESSAGE );
201
202             return false;
203         }
204     }
205
206     return true;    // arquivo aberto
207 }
208
209 // fecha o arquivo e termina o aplicativo
210 private void closeFile()
211 {
212     // fecha o arquivo e sai
213     try {
214         if ( file != null )
215             file.close();
216
217         System.exit( 0 );
218     }
219
220     // processa as exceções durante o fechamento do arquivo
221     catch( IOException ioException ) {
222         JOptionPane.showMessageDialog( this,
223             "Error closing file",
224             "Error", JOptionPane.ERROR_MESSAGE );
225         System.exit( 1 );
226     }
227 }
228
229 // executa o aplicativo
230 public static void main( String args[] )
231 {
232     new TransactionProcessor();
233 }
234
235 } // fim da classe TransactionProcessor
236
237 // classe para atualizar registros
238 class UpdateDialog extends JInternalFrame {
239     private RandomAccessFile file;
240     private BankUI userInterface;
241
242     // configura a GUI

```

Fig. 16.20 Programa de processamento de transações (parte 5 de 14).

```

243     public UpdateDialog( RandomAccessFile updateFile )
244     {
245         super( "Update Record" );
246         file = updateFile;
247         // configura os componentes da GUI
248         userInterface = new BankUI( 5 );
249         getContentPane().add( userInterface,
250             BorderLayout.CENTER );
251
252         // configura o botão Save Changes e registra ouvinte
253         JButton saveButton = userInterface.getDoTask1Button();
254         saveButton.setText( "Save Changes" );
255         saveButton.addActionListener(
256
257             new ActionListener() {
258
259                 public void actionPerformed( ActionEvent event )
260                 {
261                     addRecord( getRecord() );
262                     setVisible( false );
263                     userInterface.clearFields();
264                 }
265             }
266         );
267
268         // configura o botão Cancel e registra ouvinte
269         JButton cancelButton = userInterface.getDoTask2Button();
270         cancelButton.setText( "Cancel" );
271         cancelButton.addActionListener(
272
273             new ActionListener() {
274
275                 public void actionPerformed( ActionEvent event )
276                 {
277                     setVisible( false );
278                     userInterface.clearFields();
279                 }
280             }
281         );
282
283         // configura ouvinte para o campo de texto de transação
284         JTextField transactionField =
285             userInterface.getFields()[ BankUI.TRANSACTION ];
286
287         transactionField.addActionListener(
288
289             new ActionListener() {
290
291                 public void actionPerformed( ActionEvent event )
292                 {
293                     public void actionPerformed( ActionEvent event )
294                     {
295                         try {
296                             // adiciona o valor da transação ao saldo
297                             RandomAccessAccountRecord record = getRecord();
298
299                             // obtém valores dos campos de texto de userInterface
300                             String fieldValues[] =
301
302

```

Fig. 16.20 Programa de processamento de transações (parte 6 de 14).

```

303         userInterface.getFieldValues();
304
305         // obtém valor da transação
306         double change = Double.parseDouble(
307             fieldValues[ BankUI.TRANSACTION ] );
308
309         // especifica Strings a exibir na GUI
310         String[] values = {
311             String.valueOf( record.getAccount() ),
312             record.getFirstName(),
313             record.getLastName(),
314             String.valueOf( record.getBalance()
315                 + change ),
316             "Charge(+) or payment (-)" };
317
318         // exibe Strings na GUI
319         userInterface.setFieldValues( values );
320     }
321
322     // processa número inválido no campo de transação
323     catch ( NumberFormatException numberFormat ) {
324         JOptionPane.showMessageDialog( null,
325             "Invalid Transaction",
326             "Invalid Number Format",
327             JOptionPane.ERROR_MESSAGE );
328     }
329
330 } // fim do método actionPerformed
331
332 } // fim da classe interna anônima
333
334 }; // fim da chamada para addActionListener
335
336 // configura ouvinte para campo de texto da conta
337 JTextField accountField =
338     userInterface.getFields()[ BankUI.ACCOUNT ];
339
340 accountField.addActionListener(
341
342     new ActionListener() {
343
344         // obtém registro e exibe conteúdo na GUI
345         public void actionPerformed( ActionEvent event )
346         {
347             RandomAccessAccountRecord record = getRecord();
348
349             if ( record.getAccount() != 0 ) {
350                 String values[] = {
351                     String.valueOf( record.getAccount() ),
352                     record.getFirstName(),
353                     record.getLastName(),
354                     String.valueOf( record.getBalance() ),
355                     "Charge(+) or payment (-)" };
356
357                 userInterface.setFieldValues( values );
358             }
359
360         } // fim do método actionPerformed
361
362     } // fim da classe interna anônima

```

Fig. 16.20 Programa de processamento de transações (parte 7 de 14).

```

363     ); // fim da chamada para addActionListener
364
365     setSize( 300, 175 );
366     setVisible( false );
367 }
368
369 // obtém registro do arquivo
370 private RandomAccessAccountRecord getRecord()
371 {
372     RandomAccessAccountRecord record =
373         new RandomAccessAccountRecord();
374
375     // obtém registro do arquivo
376     try {
377         JTextField accountField =
378             userInterface.getFields() [ BankUI.ACCOUNT ];
379
380         int accountNumber =
381             Integer.parseInt( accountField.getText() );
382
383         if ( accountNumber < 1 || accountNumber > 100 ) {
384             JOptionPane.showMessageDialog( this,
385                 "Account Does Not Exist",
386                 "Error", JOptionPane.ERROR_MESSAGE );
387             return record;
388         }
389
390         // avança até a posição do registro apropriado no arquivo
391         file.seek( ( accountNumber - 1 ) *
392             RandomAccessAccountRecord.size() );
393         record.read( file );
394
395         if ( record.getAccount() == 0 )
396             JOptionPane.showMessageDialog( this,
397                 "Account Does Not Exist",
398                 "Error", JOptionPane.ERROR_MESSAGE );
399     }
400
401     // processa formato de número de conta inválido
402     catch ( NumberFormatException numberFormat ) {
403         JOptionPane.showMessageDialog( this,
404             "Invalid Account", "Invalid Number Format",
405             JOptionPane.ERROR_MESSAGE );
406     }
407
408     // processa problemas no processamento de arquivo
409     catch ( IOException ioException ) {
410         JOptionPane.showMessageDialog( this,
411             "Error Reading File",
412             "Error", JOptionPane.ERROR_MESSAGE );
413     }
414
415     return record;
416 }
417
418 } // fim do método getRecord
419
420 // adiciona registro ao arquivo
421 public void addRecord( RandomAccessAccountRecord record )
422 {

```

Fig. 16.20 Programa de processamento de transações (parte 8 de 14).

```

423     // atualiza registro no arquivo
424     try {
425         int accountNumber = record.getAccount();
426
427         file.seek( ( accountNumber - 1 ) *
428             RandomAccessAccountRecord.size() );
429
430         String[] values = userInterface.getFieldValues();
431
432         // configura nome, sobrenome e saldo no registro
433         record.setFirstName( values[ BankUI.FIRSTNAME ] );
434         record.setLastName( values[ BankUI.LASTNAME ] );
435         record.setBalance(
436             Double.parseDouble( values[ BankUI.BALANCE ] ) );
437
438         // rescreve registro no arquivo
439         record.write( file );
440     }
441
442     // processa problemas no processamento do arquivo
443     catch ( IOException ioException ) {
444         JOptionPane.showMessageDialog( this,
445             "Error Writing To File",
446             "Error", JOptionPane.ERROR_MESSAGE );
447     }
448
449     // processa valor de saldo inválido
450     catch ( NumberFormatException numberFormat ) {
451         JOptionPane.showMessageDialog( this,
452             "Bad Balance", "Invalid Number Format",
453             JOptionPane.ERROR_MESSAGE );
454     }
455
456 } // fim do método addRecord
457
458 } // fim da classe UpdateDialog
459
460 // classe para criar novos registros
461 class NewDialog extends JInternalFrame {
462     private RandomAccessFile file;
463     private BankUI userInterface;
464
465     // configura a GUI
466     public NewDialog( RandomAccessFile newFile )
467     {
468         super( "New Record" );
469
470         file = newFile;
471
472         // anexa a interface com o usuário ao diálogo
473         userInterface = new BankUI( 4 );
474         getContentPane().add( userInterface,
475             BorderLayout.CENTER );
476
477         // configura o botão Save Changes e registra ouvinte
478         JButton saveButton = userInterface.getDoTask1Button();
479         saveButton.setText( "Save Changes" );
480
481         saveButton.addActionListener(
482

```

Fig. 16.20 Programa de processamento de transações (parte 9 de 14).

```

483     new ActionListener() {
484
485         // adiciona novo registro ao arquivo
486         public void actionPerformed( ActionEvent event ) {
487             {
488                 addRecord( getRecord() );
489                 setVisible( false );
490                 userInterface.clearFields();
491             }
492
493         } // fim da classe interna anônima
494
495     }; // fim da chamada para addActionListener
496
497     JButton cancelButton = userInterface.getDoTask2Button();
498     cancelButton.setText( "Cancel" );
499
500     cancelButton.addActionListener(
501
502         new ActionListener() {
503
504             // fecha o diálogo sem armazenar novo registro
505             public void actionPerformed( ActionEvent event ) {
506                 {
507                     setVisible( false );
508                     userInterface.clearFields();
509                 }
510
511             } // fim da classe interna anônima
512
513         }; // fim da chamada para addActionListener
514
515         setSize( 300, 150 );
516         setVisible( false );
517
518     } // fim do construtor
519
520     // obtém registro do arquivo
521     private RandomAccessAccountRecord getRecord()
522     {
523         RandomAccessAccountRecord record =
524             new RandomAccessAccountRecord();
525
526         // obtém registro do arquivo
527         try {
528             JTextField accountField =
529                 userInterface.getFields()[ BankUI.ACCOUNT ];
530
531             int accountNumber =
532                 Integer.parseInt( accountField.getText() );
533
534             if ( accountNumber < 1 || accountNumber > 100 ) {
535                 JOptionPane.showMessageDialog( this,
536                     "Account Does Not Exist",
537                     "Error", JOptionPane.ERROR_MESSAGE );
538                 return record;
539             }
540
541             // avança até a posição do registro
542             file.seek( accountNumber - 1 ) *

```

Fig. 16.20 Programa de processamento de transações (parte 10 de 14).

```

543         RandomAccessAccountRecord.size() );
544
545     // lê registro do arquivo
546     record.read( file );
547 }
548
549     // processa formato de número da conta inválido
550     catch ( NumberFormatException numberFormat ) {
551         JOptionPane.showMessageDialog( this,
552             "Account Does Not Exist", "Invalid Number Format",
553             JOptionPane.ERROR_MESSAGE );
554     }
555
556     // processa problemas no processamento do arquivo
557     catch ( IOException ioException ) {
558         JOptionPane.showMessageDialog( this,
559             "Error Reading File",
560             "Error", JOptionPane.ERROR_MESSAGE );
561     }
562
563     return record;
564
565 } // fim do método getRecord
566
567 // adiciona registro ao arquivo
568 public void addRecord( RandomAccessAccountRecord record )
569 {
570     String[] fields = userInterface.getFieldValues();
571
572     if ( record.getAccount() != 0 ) {
573         JOptionPane.showMessageDialog( this,
574             "Record Already Exists",
575             "Error", JOptionPane.ERROR_MESSAGE );
576         return;
577     }
578
579     // envia os valores para gravação no arquivo
580     try {
581
582         // configura conta, nome, sobrenome e saldo
583         // para o registro
584         record.setAccount( Integer.parseInt(
585             fields[ BankUI.ACCOUNT ] ) );
586         record.setFirstName( fields[ BankUI.FIRSTNAME ] );
587         record.setLastName( fields[ BankUI.LASTNAME ] );
588         record.setBalance( Double.parseDouble(
589             fields[ BankUI.BALANCE ] ) );
590
591         // avança até a posição do registro
592         file.seek( ( record.getAccount() - 1 ) *
593             RandomAccessAccountRecord.size() );
594
595         // grava registro
596         record.write( file );
597     }
598
599     // processa formato de número da conta ou saldo inválido
600     catch ( NumberFormatException numberFormat ) {
601         JOptionPane.showMessageDialog( this,
602             "Invalid Balance", "Invalid Number Format",

```

Fig. 16.20 Programa de processamento de transações (parte 11 de 14).

```

603         JOptionPane.ERROR_MESSAGE );
604     }
605
606     // processa problemas no processamento de arquivos
607     catch ( IOException ioException ) {
608         JOptionPane.showMessageDialog( this,
609             "Error Writing To File",
610             "Error", JOptionPane.ERROR_MESSAGE );
611     }
612
613 } // fim do método addRecord
614
615 } // fim da classe NewDialog
616
617 // classe para apagar registros
618 class DeleteDialog extends JInternalFrame {
619     private RandomAccessFile file; // arquivo para saída
620     private BankUI userInterface;
621
622     // configura a GUI
623     public DeleteDialog( RandomAccessFile deleteFile )
624     {
625         super( "Delete Record" );
626
627         file = deleteFile;
628
629         // cria BankUI com apenas um campo de conta
630         userInterface = new BankUI( 1 );
631
632         getContentPane().add( userInterface,
633             BorderLayout.CENTER );
634
635         // configura o botão Delete Record e registra ouvinte
636         JButton deleteButton = userInterface.getDoTask1Button();
637         deleteButton.setText( "Delete Record" );
638
639         deleteButton.addActionListener(
640
641             new ActionListener() {
642
643                 // sobrescreve registro existente
644                 public void actionPerformed( ActionEvent event )
645                 {
646                     addRecord( getRecord() );
647                     setVisible( false );
648                     userInterface.clearFields();
649                 }
650
651             } // fim da classe interna anônima
652
653     }; // fim da chamada para addActionListener
654
655     // configura botão Cancel e registra ouvinte
656     JButton cancelButton = userInterface.getDoTask2Button();
657     cancelButton.setText( "Cancel" );
658
659     cancelButton.addActionListener(
660
661         new ActionListener() {
662

```

Fig. 16.20 Programa de processamento de transações (parte 12 de 14).

```

663         // cancela operação de apagar registro escondendo o diálogo
664         public void actionPerformed( ActionEvent event )
665         {
666             setVisible( false );
667         }
668
669     } // fim da classe interna anônima
670
671 ); // fim da chamada para addActionListener
672
673 // configura ouvinte para campo de texto da conta
674 JTextField accountField =
675     userInterface.getFields()[ BankUI.ACCOUNT ];
676
677 accountField.addActionListener(
678
679     new ActionListener() {
680
681         public void actionPerformed( ActionEvent event )
682         {
683             RandomAccessAccountRecord record = getRecord();
684         }
685
686     } // fim da classe interna anônima
687
688 ); // fim da chamada para addActionListener
689
690 setSize( 300, 100 );
691 setVisible( false );
692
693 } // fim do construtor
694
695 // obtém registro do arquivo
696 private RandomAccessAccountRecord getRecord()
697 {
698     RandomAccessAccountRecord record =
699         new RandomAccessAccountRecord();
700
701 // obtém registro do arquivo
702 try {
703     JTextField accountField =
704         userInterface.getFields()[ BankUI.ACCOUNT ];
705
706     int accountNumber =
707         Integer.parseInt( accountField.getText() );
708
709     if ( accountNumber < 1 || accountNumber > 100 ) {
710         JOptionPane.showMessageDialog( this,
711             "Account Does Not Exist",
712             "Error", JOptionPane.ERROR_MESSAGE );
713         return( record );
714     }
715
716     // avança até a posição do registro e lê o registro
717     file.seek( ( accountNumber - 1 ) *
718             RandomAccessAccountRecord.size() );
719     record.read( file );
720
721     if ( record.getAccount() == 0 )
722         JOptionPane.showMessageDialog( this,

```

Fig. 16.20 Programa de processamento de transações (parte 13 de 14).

```

723             "Account Does Not Exist",
724             "Error", JOptionPane.ERROR_MESSAGE );
725     }
726
727     // processa formato de número de conta inválido
728     catch ( NumberFormatException numberFormat ) {
729         JOptionPane.showMessageDialog( this,
730             "Account Does Not Exist",
731             "Invalid Number Format",
732             JOptionPane.ERROR_MESSAGE );
733     }
734
735     // processa problemas no processamento do arquivo
736     catch ( IOException ioException ) {
737         JOptionPane.showMessageDialog( this,
738             "Error Reading File",
739             "Error", JOptionPane.ERROR_MESSAGE );
740     }
741
742     return record;
743
744 } // fim do método getRecord
745
746 // adiciona registro ao arquivo
747 public void addRecord( RandomAccessAccountRecord record )
748 {
749     if ( record.getAccount() == 0 )
750         return;
751
752     // apaga registro configurando o número da conta como 0
753     try {
754         int accountNumber = record.getAccount();
755
756         // avança até a posição do registro
757         file.seek( ( accountNumber - 1 ) *
758                     RandomAccessAccountRecord.size() );
759
760         // configura conta como 0 e sobrescreve registro
761         record.setAccount( 0 );
762         record.write( file );
763     }
764
765     // processa problemas no processamento do arquivo
766     catch ( IOException ioException ) {
767         JOptionPane.showMessageDialog( this,
768             "Error Writing To File",
769             "Error", JOptionPane.ERROR_MESSAGE );
770     }
771
772 } // fim do método addRecord
773
774 } // fim da classe DeleteDialog

```

Fig. 16.20 Programa de processamento de transações (parte 14 de 14).

16.12 A classe `File`

Como declaramos no começo deste capítulo, o pacote `java.io` contém muitas classes para processamento de entrada e saída. Concentramo-nos nas classes para processamento de arquivos seqüenciais (`FileInputStream` e `FileOutputStream`), para processamento de fluxos de objetos (`ObjectInputStream` e `ObjectOut-`

`putStream`) e para processamento de arquivos de acesso aleatório (`RandomAccessFile`). Nesta seção, discutiremos a classe `File`, que é particularmente útil para recuperar do disco as informações sobre um arquivo ou diretório. Na verdade, os objetos da classe `File` não abrem um arquivo nem fornecem nenhuma capacidade de processamento de arquivo.

Uma aplicação da utilização de um objeto `File` é verificar se existe um arquivo antes de tentar abrir o arquivo. No *Erro comum de programação 16.1*, advertimos que abrir um arquivo existente para saída utilizando um objeto `FileOutputStream` descarta o conteúdo desse arquivo *sem aviso*. Se o programa que usa um `File` determina que um arquivo já existe, o programa pode avisar que o usuário está prestes a descartar o conteúdo original do arquivo.



Boa prática de programação 16.2

Utilize um objeto File para determinar se existe um arquivo antes de abri-lo com um objeto FileOutputStream.

A classe `File` fornece três construtores. O construtor

```
public File( String name )
```

armazena o argumento `String name` no objeto. O `name` pode conter *as informações de caminho* e um nome de arquivo ou diretório. O caminho de arquivo ou diretório leva-o ao arquivo ou diretório em disco. O caminho inclui alguns ou todos os diretórios que levam ao arquivo ou diretório. O *caminho absoluto* contém todos os diretórios, desde o *diretório-raiz*, que levam a um arquivo ou diretório específico. Cada arquivo ou diretório em uma unidade de disco particular tem o mesmo diretório-raiz em seu caminho. O *caminho relativo* contém um subconjunto dos diretórios iniciais para um arquivo ou diretório específicos. Os caminhos relativos começam no diretório em que o aplicativo foi iniciado.

O construtor

```
public File( String pathToName, String name )
```

utiliza o argumento `pathToName` (um caminho absoluto ou relativo) para localizar o arquivo ou diretório especificado por `name`.

O construtor

```
public File( File directory, String name )
```

utiliza o objeto `File directory` (um caminho absoluto ou relativo) previamente criado para localizar o arquivo ou diretório especificado por `name`.

Alguns métodos `public` da classe `File` comumente utilizados são mostrados na Fig. 16.21. Veja outros métodos `File` na Java API.

Método	Descrição
<code>boolean canRead()</code>	Devolve <code>true</code> se um arquivo pode ser lido; caso contrário, <code>false</code> .
<code>boolean canWrite()</code>	Devolve <code>true</code> se um arquivo pode ser gravado; caso contrário, <code>false</code> .
<code>boolean exists()</code>	Devolve <code>true</code> se o nome especificado como argumento para o construtor <code>File</code> é um arquivo ou diretório no caminho especificado; caso contrário, <code>false</code> .
<code>boolean isFile()</code>	Devolve <code>true</code> se o nome especificado como argumento para o construtor <code>File</code> é um arquivo; caso contrário, <code>false</code> .
<code>boolean isDirectory()</code>	Devolve <code>true</code> se o nome especificado como argumento para o construtor <code>File</code> é um diretório; caso contrário, <code>false</code> .
<code>boolean isAbsolute()</code>	Devolve <code>true</code> se os argumentos especificados para o construtor <code>File</code> indicam um caminho absoluto para um arquivo ou diretório; caso contrário, <code>false</code> .
<code>String getAbsolutePath()</code>	Devolve um <code>String</code> com o caminho absoluto do arquivo ou diretório.

Fig. 16.21 Alguns métodos de `File` comumente utilizados (parte 1 de 2).

Método	Descrição
<code>String getName()</code>	Devolve um <code>String</code> com o nome do arquivo ou diretório.
<code>String getPath()</code>	Devolve um <code>String</code> com o caminho do arquivo ou diretório.
<code>String getParent()</code>	Devolve um <code>String</code> com o diretório-pai do arquivo ou diretório – isto é, o diretório em que se pode localizar o arquivo ou diretório.
<code>long length()</code>	Devolve o comprimento do arquivo em <code>bytes</code> . Se o objeto <code>File</code> representa um diretório, 0 é devolvido.
<code>long lastModified()</code>	Devolve a representação do horário em que o ocorreu a última modificação do arquivo ou diretório. O valor devolvido é somente útil para comparação com outros valores devolvidos por esse método.
<code>String[] list()</code>	Devolve um <code>array</code> de <code>Strings</code> que representa o conteúdo de um diretório.

Fig. 16.21 Alguns métodos de `File` comumente utilizados (parte 2 de 2).



Boa prática de programação 16.3

Utilize o método `isFile` de `File` para determinar se um objeto `File` representa um arquivo (não um diretório), antes de tentar abrir um arquivo.



Boa prática de programação 16.4

Antes de tentar abrir um arquivo para leitura, utilize o método `canRead` de `File` para determinar se o arquivo pode ser lido.



Boa prática de programação 16.5

Antes de tentar abrir um arquivo para gravação, utilize o método `canWrite` de `File` para determinar se o arquivo pode ser gravado.

A Fig. 16.22 demonstra a classe `File`. O aplicativo `FileTest` cria uma GUI que contém um `JTextField` para digitar um nome de arquivo ou um nome de diretório e uma `JTextArea` para exibir informações sobre o nome de arquivo ou o nome de diretório digitado.

```

1 // Fig. 16.22: FileTest.java
2 // Demonstrando a classe File.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.io.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class FileTest extends JFrame
13     implements ActionListener {
14
15     private JTextField enterField;
16     private JTextArea outputArea;
17
18     // configura a GUI
19     public FileTest()
20     {

```

Fig. 16.22 Demonstrando a classe `File` (parte 1 de 4).

```

21     super( "Testing class File" );
22
23     enterField = new JTextField(
24         "Enter file or directory name here" );
25     enterField.addActionListener( this );
26     outputArea = new JTextArea();
27
28     ScrollPane scrollPane = new ScrollPane();
29     scrollPane.add( outputArea );
30
31     Container container = getContentPane();
32     container.add( enterField, BorderLayout.NORTH );
33     container.add( scrollPane, BorderLayout.CENTER );
34
35     setSize( 400, 400 );
36     show();
37 }
38
39 // exibe informações sobre o arquivo que o usuário especifica
40 public void actionPerformed( ActionEvent actionEvent )
41 {
42     File name = new File( actionEvent.getActionCommand() );
43
44     // se name existe, envia para a saída informações sobre ele
45     if ( name.exists() ) {
46         outputArea.setText(
47             name.getName() + " exists\n" +
48             ( name.isFile() ?
49                 "is a file\n" : "is not a file\n" ) +
50             ( name.isDirectory() ?
51                 "is a directory\n" : "is not a directory\n" ) +
52             ( name.isAbsolute() ? "is absolute path\n" :
53                 "is not absolute path\n" ) +
54             "Last modified: " + name.lastModified() +
55             "\nLength: " + name.length() +
56             "\nPath: " + name.getPath() +
57             "\nAbsolute path: " + name.getAbsolutePath() +
58             "\nParent: " + name.getParent() );
59
60     // envia informações para a saída se name é um arquivo
61     if ( name.isFile() ) {
62
63         // acrescenta conteúdo do arquivo a outputArea
64         try {
65             BufferedReader input = new BufferedReader(
66                 new FileReader( name ) );
67             StringBuffer buffer = new StringBuffer();
68             String text;
69             outputArea.append( "\n\n" );
70
71             while ( ( text = input.readLine() ) != null )
72                 buffer.append( text + "\n" );
73
74             outputArea.append( buffer.toString() );
75         }
76
77         // processa problemas no processamento do arquivo
78         catch( IOException ioException ) {
79             JOptionPane.showMessageDialog( this,

```

Fig. 16.22 Demonstrando a classe `File` (parte 2 de 4).

```

80             "FILE ERROR",
81             "FILE ERROR", JOptionPane.ERROR_MESSAGE );
82         }
83     }
84
85     // envia para a saída uma listagem do diretório
86     else if ( name.isDirectory() ) {
87         String directory[] = name.list();
88
89         outputArea.append( "\n\nDirectory contents:\n" );
90
91         for ( int i = 0; i < directory.length; i++ )
92             outputArea.append( directory[ i ] + "\n" );
93     }
94 }
95
96 // se não é arquivo nem diretório, envia mensagem de erro para a saída
97 else {
98     JOptionPane.showMessageDialog( this,
99         actionEvent.getActionCommand() + " Does Not Exist",
100        "ERROR", JOptionPane.ERROR_MESSAGE );
101 }
102
103 } // fim do método actionPerformed
104
105 // executa o aplicativo
106 public static void main( String args[] )
107 {
108     FileTest application = new FileTest();
109
110     application.setDefaultCloseOperation(
111         JFrame.EXIT_ON_CLOSE );
112 }
113
114 } // fim da classe FileTest

```

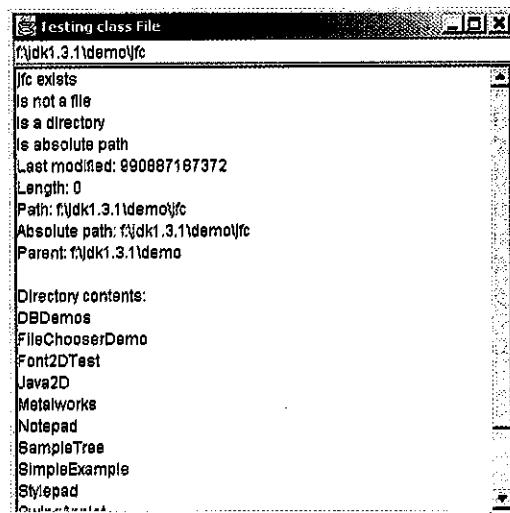


Fig. 16.22 Demonstrando a classe `File` (parte 3 de 4).

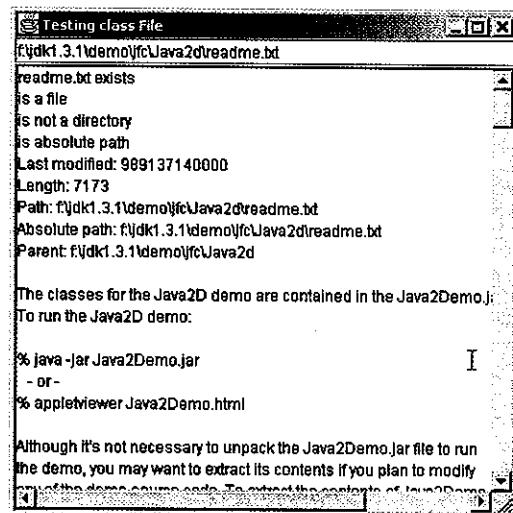


Fig. 16.22 Demonstrando a classe **File** (parte 4 de 4).

O usuário digita um nome de arquivo ou um nome de diretório no campo de texto e pressiona a tecla *Enter* para invocar o método **actionPerformed** (linhas 40 a 103), que cria um novo objeto **File** e o atribui a **name**. A linha 45 invoca o método **exists** de **File** para determinar se o nome digitado pelo usuário existe (seja como arquivo ou diretório), no disco. Se o nome digitado pelo usuário não existe, o método **actionPerformed** prossegue para as linhas 97 a 101 e exibe um diálogo de mensagem contendo o nome que o usuário digitou seguido por “**Does Not Exist**”. Caso contrário, o corpo da estrutura **if** é executado (linhas 45 a 94). O programa envia para saída o nome do arquivo ou diretório, depois envia para saída os resultados de testar o objeto **File** com **isFile** (linha 48), **isDirectory** (linha 50) e **isAbsolute** (linha 52). Em seguida, o programa exibe os valores devolvidos por **lastModified** (linha 54), **length** (linha 55), **getPath** (linha 56), **getAbsolutePath** (linha 57) e **getParent** (linha 58).

Se o objeto **File** representa um arquivo (linha 61), o programa lê o conteúdo do arquivo e exibe o conteúdo na **JTextArea**. O programa utiliza um objeto **BufferedReader** encadeado com um objeto **FileReader** (linhas 65 e 66) para abrir o arquivo para leitura e ler o arquivo uma linha por vez com o método **readLine** (linha 71). Observe que o objeto **FileReader** foi inicializado com o objeto **File name** (linha 66). Se o objeto **File** representa um diretório (linha 86), o programa lê o conteúdo do diretório utilizando o método **list** de **File**, depois o conteúdo do diretório é exibido na **JTextArea**.

A primeira saída demonstra um objeto **File** associado ao diretório **jfc** do Java 2 Software Development Kit. A segunda saída desse programa demonstra um objeto **File** associado ao arquivo **readme.txt** do Java 2 Software Development Kit. Em ambos os casos, especificamos um caminho absoluto em nosso computador pessoal.

Observe que o caractere separador **** (barra invertida) é utilizado para separar diretórios e arquivos no caminho. Em uma estação de trabalho UNIX, o caractere separador seria uma barra normal, **/**. Java considera os dois caracteres como idênticos em um nome de caminho. Assim, se especificamos o caminho

```
c:\java/readme.txt
```

que utiliza os dois tipos de caractere separador, Java ainda processaria o arquivo adequadamente.



Erro comum de programação 16.2

Utilizar **** como separador de diretório em vez de **\ ** em um literal de string é um erro de lógica. Uma única **** indica que a **** e o próximo caractere representam uma sequência de escape. Para inserir uma **** em um literal de string, você deve utilizar **\ **.



Boa prática de programação 16.6

Ao construir strings que representam informações sobre caminho, utilize `File.separatorChar` para obter o caractere separador próprio do computador local, em vez de usar explicitamente / ou \.

Resumo

- Todos os itens de dados processados por um computador são reduzidos a combinações de zeros e uns.
- O menor item de dados em um computador (um *bit*) pode assumir o valor 0 ou o valor 1.
- Os dígitos, as letras e os símbolos especiais são denominados caracteres. O conjunto de todos os caracteres utilizados para gravar programas e representar itens de dados em um computador particular se chama conjunto de caracteres desse computador. Cada caractere em um conjunto de caracteres do computador é representado como um padrão de 1s e 0s (os caracteres em Java são caracteres Unicode compostos de 2 *bytes*).
- O campo é um grupo de caracteres (ou *bytes*) que possui um significado.
- O registro é um grupo de campos relacionados.
- Pelo menos um campo em um registro é escolhido como chave de registro para identificar que um registro pertence a uma pessoa ou entidade particular que é único entre todos os outros registros no arquivo.
- Java não impõe uma estrutura a um arquivo. Noções como “registro” não existem em Java. O programador deve estruturar o arquivo adequadamente para atender aos requisitos de um aplicativo.
- A coleção de programas projetados para criar e gerenciar bancos de dados é denominada sistema de gerenciamento de bancos de dados (DBMS – *database management system*).
- Java vê cada arquivo como um fluxo seqüencial de *bytes*.
- Cada arquivo acaba com alguma forma de marcador de fim de arquivo dependente de máquina.
- Os fluxos fornecem canais de comunicação entre arquivos e programas, memória ou outros programas através de uma rede.
- Os programas usam as classes do pacote `java.io` para realizar E/S de arquivos em Java. Esse pacote inclui as definições para as classes de fluxo como `FileInputStream`, `FileOutputStream`, `DataInputStream` e `DataOutputStream`.
- Os arquivos são abertos instanciando-se objetos das classes de fluxo `FileInputStream`, `FileOutputStream` e `RandomAccessFile`.
- `InputStream` (subclasse de `Object`) e `OutputStream` (subclasse de `Object`) são classes `abstract` que definem métodos para realizar entrada e saída respectivamente.
- A entrada/saída de arquivos é feita com `FileInputStream` (subclasse de `InputStream`) e `FileOutputStream` (subclasse de `OutputStream`).
- Os *pipes* são canais de comunicação sincronizada entre *threads*. O *pipe* é estabelecido entre duas *threads*. A *thread* envia dados para outra gravando em um `PipedOutputStream` (subclasse de `OutputStream`). A *thread* de destino lê as informações do *pipe* através de um `PipedInputStream` (subclasse de `InputStream`).
- O `PrintStream` (subclasse de `FilterOutputStream`) é utilizado para enviar a saída para a tela (ou a “saída-padrão” como definido por seu sistema operacional local). O `System.out` é um `PrintStream` (assim como `System.err`).
- O `FilterInputStream` filtra um `InputStream` e o `FilterOutputStream` filtra um `OutputStream`; filtrar significa simplesmente que o fluxo fornece funcionalidade adicional como armazenamento em *buffer* (*buffering*), monitoração dos números de linha ou agregação de *bytes* de dados em unidades significativas do tipo primitivo de dados.
- Ler dados diretamente como *bytes* é rápido, mas grosseiro. Normalmente os programas lêem os dados como agrupados de *bytes* que formam um `int`, um `float`, um `double` e assim por diante. Para realizar isso, utilizamos um `DataInputStream` (subclasse de classe `FilterInputStream`).
- A interface `DataInput` é implementada pelas classes `DataInputStream` e `RandomAccessFile`, as quais precisam ler tipos primitivos de dados de um fluxo.
- `DataInputStreams` permitem a um programa ler dados binários de um `InputStream`.
- A interface `DataInput` inclui os métodos `read` (para arrays de byte), `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readFully` (para arrays de byte), `readInt`, `readLine`, `readLong`, `readShort`, `readUnsignedByte`, `readUnsignedShort`, `readUTF` (para Unicode) e `skipBytes`.
- A interface `DataOutput` é implementada pelas classes `DataOutputStream` (subclasse da classe `FilterOutputStream`) e `RandomAccessFile`, as quais precisam gravar tipos primitivos de dados em um `OutputStream`.

- **DataOutputStreams** permitem que um programa grave dados binários em um **OutputStream**. A interface **DataOutput** inclui os métodos **flush**, **size**, **write** (para um byte), **write** (para um array de byte), **writeBoolean**, **writeByte**, **writeBytes**, **writeChar**, **writeChars** (para Strings Unicode), **writeDouble**, **writeFloat**, **writeInt**, **writeLong**, **writeShort** e **writeUTF**.
- O armazenamento em *buffer (buffering)* é uma técnica de aprimoramento do desempenho de E/S.
- Com um **BufferedOutputStream** (subclasse da classe **FilterOutputStream**), cada instrução de saída não resulta necessariamente em uma transferência física real de dados para o dispositivo de saída. Em vez disso, cada operação de saída é dirigida para uma região na memória, chamada *buffer*, que é suficientemente grande para armazenar os dados de muitas operações de saída. Assim, a saída real para o dispositivo de saída é realizada em uma operação física grande de saída, toda vez que o *buffer* se enche. As operações de saída dirigidas para o *buffer* de saída na memória são freqüentemente chamadas de operações lógicas de saída.
- Com um **BufferedInputStream**, muitos pedaços ou trechos “lógicos” de dados de um arquivo são lidos como uma grande operação física de entrada em um *buffer* de memória. À medida que o programa solicita cada trecho novo dos dados, a busca é feita no *buffer* (isso é às vezes conhecido como operação lógica de entrada). Quando o *buffer* está vazio, a próxima operação física de entrada do dispositivo de entrada é realizada lendo-se o próximo grupo de trechos “lógicos” de dados. Portanto, o número de operações físicas reais de entrada é pequeno, comparado com o número de solicitações de leitura emitido pelo programa.
- Com um **BufferedOutputStream**, um *buffer* parcialmente cheio pode ser forçado a enviar a saída para o dispositivo a qualquer momento com um **flush** explícito.
- A interface **ObjectInput** é semelhante à interface **DataInput**, mas inclui métodos adicionais para ler **Objects** a partir de **InputStreams**.
- A interface **ObjectOutput** é semelhante à interface **DataOutput**, mas inclui métodos adicionais para gravar **Objects** em **OutputStreams**.
- As classes **ObjectInputStream** e **ObjectOutputStream** implementam as interfaces **ObjectInput** e **ObjectOutput**, respectivamente.
- O **PushBackInputStream** é uma subclasse da classe **FilterInputStream**. O aplicativo que lê um **PushBackInputStream** lê bytes do fluxo e forma agregados que consistem em vários bytes. Às vezes, para determinar se o agregado está completo, o aplicativo deve ler o primeiro caractere além do fim do primeiro agregado. Uma vez que o programa determinou que o agregado atual está completo, o caractere extra é reinserido de volta no fluxo.
- **PushBackInputStreams** são utilizados por programas como compiladores que *analisam sintaticamente* suas entradas, isto é, dividem-nas em unidades significativas (como palavras-chave, identificadores e operadores que o compilador Java deve reconhecer).
- O **RandomAccessFile** (subclasse de **Object**) é útil para aplicativos de acesso direto e para aplicativos de processamento de transações, como sistemas de reserva de passagens aéreas e sistemas de ponto de venda.
- Com um arquivo de acesso seqüencial, cada solicitação sucessiva de entrada/saída lê ou grava o próximo conjunto consecutivo de dados no arquivo.
- Com um arquivo de acesso aleatório, cada solicitação sucessiva de entrada/saída pode ser dirigida para qualquer parte do arquivo, talvez muito distante da parte do arquivo à que se faz referência na solicitação anterior.
- Os aplicativos de acesso direto fornecem acesso rápido a itens de dados específicos em arquivos grandes; tais aplicativos são freqüentemente utilizados quando as pessoas estão esperando respostas – essas respostas devem ser disponibilizadas rapidamente ou as pessoas podem ficar impacientes e “levar seu negócio para outro lugar”.
- O **ByteArrayInputStream** (subclasse da classe **abstract InputStream**) obtém os dados de entrada a partir de um array de bytes na memória.
- O **ByteArrayOutputStream** (subclasse da classe **abstract OutputStream**) envia a saída para um array de bytes na memória.
- A aplicação de entrada/saída de *array de byte* é a validação de dados. O programa pode inserir uma linha inteira por vez do fluxo de entrada em um *array de byte*. Assim, a rotina de validação pode escrutinar o conteúdo do *array de byte* e corrigir os dados, se necessário. O programa agora pode passar para a leitura do *array de byte*, sabendo que os dados de entrada estão no formato adequado.
- O **StringBufferInputStream** (subclasse da classe **abstract InputStream**) recebe entrada de um objeto **StringBuffer**.
- O **SequenceInputStream** (subclasse da classe **abstract InputStream**) permite que vários **InputStreams** sejam concatenados de modo que o programa veja o grupo como um **InputStream** contínuo. Quando o fim de cada fluxo de entrada é alcançado, o fluxo é fechado e o próximo fluxo na seqüência é aberto.

- As classes **BufferedReader** e **BufferedWriter** permitem armazenamento eficiente em *buffers* de fluxos baseados em caracteres.
- As classes **CharArrayReader** e **CharArrayWriter** lêem e gravam um fluxo de caracteres em um *array* de caracteres.
- O **PushbackReader** (subclasse da classe **abstract FilterReader**) permite devolver caracteres para um fluxo de caracteres. O **LineNumberReader** (subclasse de **BufferedReader**) é um fluxo de caracteres com *buffer* que monitora números de linha (isto é, uma nova linha, um retorno de carro ou uma combinação de quebra de linha e retorno de carro).
- As classes **FileReader** (uma subclasse de **InputStreamReader**) e **FileWriter** (uma subclasse de **OutputStreamWriter**) lêem e gravam caracteres em um arquivo. As classes **PipedReader** e **PipedWriter** são fluxos de caractere em *pipes*. As classes **StringReader** e **StringWriter** lêem e gravam caracteres em **Strings**. O **PrintWriter** grava caracteres em um fluxo.
- A classe **File** permite que os programas obtenham as informações sobre um arquivo ou diretório.
- Os arquivos são abertos para saída criando um objeto da classe **FileOutputStream**. Um argumento é passado para o construtor – o nome do arquivo. Os arquivos existentes são truncados e todos os dados no arquivo são perdidos. Os arquivos que não existem são criados.
- Os programas podem processar nenhum arquivo, um arquivo ou vários arquivos. Cada arquivo tem um nome único e é associado com um objeto adequado de fluxo de arquivo. Todos os métodos de processamento de arquivo devem fazer referência a um arquivo com o objeto adequado.
- O ponteiro de posição indica a posição no arquivo a partir da qual a próxima entrada deve ocorrer ou em que a próxima saída será colocada.
- Uma maneira conveniente de implementar os arquivos de acesso aleatório é utilizar somente registros de tamanho fixo. Utilizando essa técnica, um programa pode rapidamente calcular a localização exata de um registro em relação ao início do arquivo.
- Podem-se inserir dados em um arquivo de acesso aleatório sem destruir outros dados no arquivo. Podem-se atualizar ou excluir dados sem regravar o arquivo inteiro.
- A classe **RandomAccessFile** tem os mesmos recursos para entrada e saída que as classes **DataInputStream** e **DataOutputStream**, e também pode buscar uma posição específica de *byte* no arquivo com o método **seek**.

Terminologia

<i>abrir um arquivo</i>	<i>classe ByteArrayOutputStream</i>
<i>aplicativo de acesso instantâneo</i>	<i>classe CharArrayReader</i>
<i>aplicativos de acesso direto</i>	<i>classe CharArrayWriter</i>
<i>armazenamento em buffer</i>	<i>classe File</i>
<i>arquivo</i>	<i>classe FileInputStream</i>
<i>arquivo de acesso aleatório</i>	<i>classe FileOutputStream</i>
<i>arquivo de acesso seqüencial</i>	<i>classe FileReader</i>
<i>banco de dados</i>	<i>classe FileWriter</i>
<i>bit</i>	<i>classe FilterInputStream</i>
<i>buffer</i>	<i>classe FilterOutputStream</i>
<i>buffer de memória</i>	<i>classe FilterReader</i>
<i>buffer parcialmente cheio</i>	<i>classe InputStream</i>
<i>byte</i>	<i>classe InputStreamReader</i>
<i>caminho absoluto</i>	<i>classe JFileChooser</i>
<i>caminho relativo</i>	<i>classe LineNumberReader</i>
<i>campo</i>	<i>classe ObjectInputStream</i>
<i>campo alfabético</i>	<i>classe ObjectOutputStream</i>
<i>campo alfanumérico</i>	<i>classe OutputStream</i>
<i>campo de caractere</i>	<i>classe OutputStreamWriter</i>
<i>campo numérico</i>	<i>classe PipedInputStream</i>
<i>chave de registro</i>	<i>classe PipedOutputStream</i>
<i>classe BufferedInputStream</i>	<i>classe PipedReader</i>
<i>classe BufferedOutputStream</i>	<i>classe PipedWriter</i>
<i>classe BufferedReader</i>	<i>classe PrintStream</i>
<i>classe BufferedWriter</i>	<i>classe PrintWriter</i>
<i>classe ByteArrayInputStream</i>	<i>classe PushBackInputStream</i>

classe PushbackReader	método <i>list</i> da classe File
classe RandomAccessFile	método <i>read</i>
classe Reader	método <i>readBoolean</i>
classe SequenceInputStream	método <i>readByte</i>
classe StringReader	método <i>readChar</i>
classe StringWriter	método <i>readDouble</i>
classe Writer	método <i>readFloat</i>
conjunto de caracteres	método <i>readFully</i>
conjunto de caracteres Unicode	método <i>readInt</i>
constante CANCEL_OPTION	método <i>readLong</i>
constante DIRECTORIES_ONLY	método <i>readObject</i>
constante FILES_AND_DIRECTORIES	método <i>readShort</i>
constante FILES_ONLY	método <i>readUnsignedByte</i>
dados persistentes	método <i>readUnsignedShort</i>
diálogo modal	método <i>seek</i>
dígito binário	método <i>setFileSelectionMode</i>
dígito decimal	método <i>showOpenDialog</i>
diretório	método <i>showSaveDialog</i>
diretório-raiz	método <i>write</i>
encadeando objetos de fluxo	método <i>writeBoolean</i>
EndOfFileException	método <i>writeByte</i>
fechar um arquivo	método <i>writeBytes</i>
final do arquivo	método <i>writeChar</i>
flush	método <i>writeChars</i>
fluxo de entrada	método <i>writeDouble</i>
fluxo de saída	método <i>writeFloat</i>
hierarquia de dados	método <i>writeInt</i>
interface DataInput	método <i>writeLong</i>
interface DataOutput	método <i>writeObject</i>
interface ObjectInput	método <i>writeShort</i>
interface ObjectOutput	modo de abertura de arquivo r
interface Serializable	modo de abertura de arquivo rw
IOException	nome de arquivo
marcador de final do arquivo	operação física de entrada
método <i>canRead</i> da classe File	operação física de saída
método <i>canWrite</i> da classe File	operação lógica de entrada
método close	operação lógica de saída
método <i>exists</i> da classe File	pipe
método <i>getAbsolutePath</i> da classe File	ponteiro de posição no arquivo
método <i>getName</i> da classe File	registro
método <i>getParent</i> da classe File	saída padrão
método <i>getPath</i> da classe File	sistema de gerenciamento de bancos de dados (DBMS)
método <i>getSelectedFile</i>	System.err (fluxo de erro padrão)
método <i>isAbsolute</i> da classe File	System.in (fluxo de entrada padrão)
método <i>isDirectory</i> da classe File	System.out (fluxo de saída padrão)
método <i>isFile</i> da classe File	truncar um arquivo existente
método <i>lastModified</i> da classe File	validação de dados
método <i>length</i> da classe File	

Exercícios de auto-revisão

16.1 Preencha as lacunas em cada uma das frases seguintes:

- Em última instância, todos os itens de dados processados por um computador são reduzidos a combinações de _____ e _____.
- O menor item de dados que um computador pode processar se chama _____.
- O _____ é um grupo de registros relacionados.
- Dígitos, letras e símbolos especiais são conhecidos como _____.

- e) Um grupo de arquivos relacionados é chamado de _____.
- f) O método _____ das classes de fluxo de arquivo `FileOutputStream`, `FileInputStream` e `RandomAccessFile` fecha um arquivo.
- g) O método _____ de `RandomAccessFile` lê um inteiro do fluxo especificado.
- h) O método _____ de `RandomAccessFile` lê uma linha de texto do fluxo especificado.
- i) O método _____ de `RandomAccessFile` configura o ponteiro de posição no arquivo para uma posição específica em um arquivo para entrada ou saída.
- 16.2** Determine quais das seguintes afirmações são *verdadeiras* e quais são *falsas*. Se for *falsa*, explique por quê.
- O programador deve criar explicitamente os objetos `System.in`, `System.out` e `System.err`.
 - Se o ponteiro de posição no arquivo aponta para uma posição em um arquivo sequencial que não é seu início, o arquivo deve ser fechado e reaberto para ler a partir do início do arquivo.
 - Não é necessário pesquisar todos os registros em um arquivo de acesso aleatório para localizar um registro específico.
 - Os registros em arquivos de acesso aleatório devem ter tamanho uniforme.
 - O método `seek` deve realizar a busca em relação ao início de um arquivo.
- 16.3** Suponha que cada uma das seguintes instruções se aplique ao mesmo programa.
- Escreva uma instrução que abre o arquivo "`oldmast.dat`" para entrada; utilize o objeto `inOldMaster` do tipo `ObjectInputStream` encadeado com um objeto `FileInputStream`.
 - Escreva uma instrução que abre o arquivo "`trans.dat`" para entrada; utilize o objeto `inTransaction` do tipo `ObjectInputStream` encadeado com um objeto `FileInputStream`.
 - Escreva uma instrução que abre o arquivo "`newmast.dat`" para saída (e criação); utilize o objeto `outNewMaster` do tipo `ObjectOutputStream` encadeado com um `FileOutputStream`.
 - Escreva um conjunto de instruções que lê um registro do arquivo "`oldmast.dat`". O registro consiste no inteiro `accountNumber`, no `string name` e no ponto flutuante `currentBalance`; utilize o objeto `inOldMaster` do tipo `ObjectInputStream`.
 - Escreva um conjunto de instruções que lê um registro do arquivo "`trans.dat`". O registro consiste em um inteiro `accountNumber` e no ponto flutuante `dollarAmount`; utilize o objeto `inTransaction` do tipo `ObjectInputStream`.
 - Escreva um conjunto de instruções que faz a saída de um registro para o arquivo "`newmast.dat`". O registro consiste no inteiro `accountNumber`, no `string name` e no ponto flutuante `currentBalance`; utilize o objeto `outNewMaster` do tipo `DataOutputStream`.
- 16.4** Localize o erro e mostre como corrigi-lo em cada uma das instruções seguintes.
- Suponha que `account`, `company` e `amount` estejam declarados.
- ```
ObjectOutputStream outputStream;
outputStream.writeInt(account);
outputStream.writeChars(company);
outputStream.writeDouble(amount);
```
- A instrução seguinte deve ler um registro do arquivo "`payables.dat`". O objeto `inPayable` do tipo `ObjectInputStream` refere-se a esse arquivo e o objeto `inReceivable` do tipo `FileInputStream` refere-se ao arquivo "`receivables.dat`".
- ```
account = inReceivable.readInt();
companyID = inReceivable.readLong();
amount = inReceivable.readDouble();
```

Respostas aos exercícios de auto-revisão

- 16.1** a) 1s, 0s. b) *bit*. c) arquivo. d) caracteres. e) banco de dados. f) `close`. g) `readInt`. h) `readLine`. i) `seek`.
- 16.2** a) Falsa. Esses três fluxos são criados automaticamente para o programador.
 b) Verdadeira.
 c) Verdadeira.
 d) Falsa. Os registros em um arquivo de acesso aleatório são normalmente de comprimento uniforme.
 e) Verdadeira.

- 16.3**
- a)

```
ObjectInputStream inOldMaster;
inOldMaster = new ObjectInputStream(
    new FileInputStream( "oldmast.dat" ) );
```
 - b)

```
ObjectInputStream inTransaction;
inTransaction = new ObjectInputStream(
    new FileInputStream( "trans.dat" ) );
```
 - c)

```
ObjectOutputStream outNewMaster;
outNewMaster = new ObjectOutputStream(
    new FileOutputStream( "newmast.dat" ) );
```
 - d)

```
accountNumber = inOldMaster.readInt();
name = inOldMaster.readUTF();
currentBalance = inOldMaster.readDouble();
```
 - e)

```
accountNumber = inTransaction.readInt();
dollarAmount = inTransaction.readDouble();
```
 - f)

```
outNewMaster.writeInt( accountNumber );
outNewMaster.writeUTF( name );
outNewMaster.writeDouble( currentBalance );
```
- 16.4**
- a) Erro: o arquivo não foi aberto antes de se fazer uma tentativa de realizar a saída dos dados para o fluxo.
Correção: crie um novo objeto `ObjectOutputStream` encadeado com um objeto `FileOutputStream` para abrir o arquivo para saída.
 - b) Erro: o objeto `FileInputStream` incorreto está sendo utilizado para ler um registro do arquivo "`payables.dat`".
Correção: utilize o objeto `inPayable` para fazer referência a "`payables.dat`".

Exercícios

- 16.5** Preencha as lacunas em cada uma das frases seguintes:
- a) Os computadores armazenam grandes quantidades de dados em dispositivos de armazenamento secundários como _____.
 - b) O _____ é composto por vários campos.
 - c) O campo que pode conter somente dígitos, letras e espaços em branco é chamado de campo _____.
 - d) Para facilitar a recuperação de registros específicos de um arquivo, um campo em cada registro é escolhido como uma _____.
 - e) A vasta maioria das informações armazenadas em sistemas de computadores é armazenada em arquivos _____.
 - f) Os objetos de fluxo padrão são _____, _____ e _____.
- 16.6** Declare quais das afirmações seguintes são *verdadeiras* e quais são *falsas*. Se for *falsa*, explique por quê.
- a) As impressionantes funções realizadas pelos computadores envolvem essencialmente a manipulação de zeros e uns.
 - b) As pessoas especificam programas e itens de dados como caracteres; os computadores, então, manipulam e processam esses caracteres como grupos de zeros e uns.
 - c) O CEP de 8 algarismos de uma pessoa é um exemplo de um campo numérico.
 - d) O endereço residencial de uma pessoa é geralmente considerado um campo alfabético.
 - e) Os itens de dados representados nos computadores formam uma hierarquia de dados na qual os itens de dados tornam-se cada vez maiores e mais complexos à medida que progredimos de campos para caracteres, para *bits*, etc.
 - f) A chave de registro identifica que um registro pertence a um campo em particular.
 - g) As empresas armazenam todas as suas informações em um único arquivo para facilitar o processamento por computador.
 - h) Quando um programa cria um arquivo, o arquivo é automaticamente retido pelo computador para referência futura.

16.7 O Exercício 16.3 pediu que o leitor escrevesse uma série de instruções isoladas. De fato, essas instruções formam o núcleo de um importante tipo de programa processador de arquivos, a saber, um programa de correspondência de arquivos (*file-matching program*). No processamento de dados comerciais, é comum ter vários arquivos em cada sistema aplicativo. Em um sistema de contas a receber, por exemplo, geralmente há um arquivo-mestre que contém informações detalhadas sobre cada cliente, como seu nome, endereço, número de telefone, saldo, limite de crédito, termos de desconto, cláusulas de contrato e possivelmente um histórico condensado de compras recentes e pagamentos de contas.

- À medida que ocorrem transações (isto é, são feitas vendas e pagamentos em dinheiro chegam pelo correio), elas são inseridas em um arquivo. No fim de cada período de negócios (isto é, um mês para algumas empresas, uma semana para outras e um dia em alguns casos), o arquivo de transações (chamado de "**trans.dat**" no Exercício 16.3) é aplicado ao arquivo-mestre (chamado de "**oldmast.dat**" no Exercício 16.3), atualizando, assim, o registro de compras e pagamentos de cada conta. Durante uma operação de atualização, o arquivo-mestre é regravado como um novo arquivo ("**newmast.dat**"), que é então utilizado no fim do próximo período de negócios para iniciar o processo de atualização novamente.
- Os programas de correspondência de arquivos devem lidar com certos problemas que não existem em programas de um único arquivo. Por exemplo, nem sempre ocorre uma correspondência. Um cliente no arquivo-mestre pode não ter feito nenhuma compra nem pagamentos em dinheiro no período atual de negócios; portanto, nenhum registro aparecerá para esse cliente no arquivo de transações. De maneira semelhante, o cliente que fez algumas compras ou pagamentos em dinheiro pode ter mudado para essa comunidade recentemente e a empresa pode não ter tido oportunidade de criar um registro-mestre para esse cliente.
- Utilize as instruções do Exercício 16.3 como base para escrever um programa completo de correspondência de arquivos de contas a receber. Utilize o número de conta em cada arquivo como chave de registro para fins de correspondência. Suponha que cada arquivo é um arquivo sequencial com registros armazenados em ordem crescente de número de conta.
- Quando ocorre uma correspondência (isto é, quando os registros com o mesmo número de conta aparecem tanto no arquivo-mestre como no arquivo de transação), adicione a quantia em dólares no arquivo de transação ao saldo atual no arquivo-mestre e grave o registro "**newmast.dat**" (suponha que as compras são indicadas por quantias positivas no arquivo de transação e que os pagamentos são indicados por quantias negativas.) Quando houver um registro-mestre para uma conta particular, mas nenhum registro de transação correspondente, apenas grave o registro-mestre em "**newmast.dat**". Quando houver um registro de transação mas não o registro mestre correspondente, imprima a mensagem "**Unmatched transaction record for account number ...**" (preencha com o número de conta do registro de transação).

16.8 Depois de escrever o programa do Exercício 16.7, escreva um programa simples para criar alguns dados de teste para verificar o programa. Utilize os exemplos de dados de conta das Figs. 16.23 e 16.24. Execute o programa do Exercício 16.7 utilizando os arquivos de dados de teste criados neste exercício. Imprima o novo arquivo-mestre. Verifique se as contas foram atualizadas corretamente.

16.9 É possível (na verdade, é comum) ter vários registros de transação com a mesma chave de registro. Isso ocorre porque um cliente em particular talvez faça várias compras e pagamentos em dinheiro durante um período de negócios. Rescreva seu programa de correspondência de arquivo de contas a receber do Exercício 16.7 para oferecer a possibilidade de tratamento de vários registros de transação com a mesma chave de registro. Modifique os dados de teste do Exercício 16.8 para incluir os registros de transação adicionais da Fig. 16.25.

Número da conta do arquivo-mestre	Nome	Saldo
100	Alan Jones	348,17
300	Mary Smith	27,19
500	Sam Sharp	0,00
700	Suzy Green	-14,22

Fig. 16.23 Exemplo de dados para o arquivo-mestre.

Número da conta do arquivo de transação	Valor de transação
100	27,14
300	62,11
400	100,56
900	82,17

Fig. 16.24 Exemplo de dados para o arquivo de transações.

Número da conta	Quantia em dólar
300	83,89
700	80,78
700	1,53

Fig. 16.25 Registros de transações adicionais.

16.10 Você é o proprietário de uma loja de equipamentos de construção e precisa manter um inventário que informe as diferentes ferramentas que você tem, quantas de cada você tem à disposição e o preço de cada uma. Escreva um programa que inicializa o arquivo de acesso aleatório "hardware.dat" para 100 registros vazios, permite que você digite os dados relativos a cada ferramenta, permite listar todas as suas ferramentas, permite que você exclua um registro de uma ferramenta que você não tem mais e permite que você atualize *qualquer* informação no arquivo. O número de identificação da ferramenta deve ser o número do registro. Utilize as informações da Fig. 16.26 para iniciar seu arquivo.

Nº do registro	Nome da ferramenta	Quantidade	Preço
3	Lixadeira	18	35,99
19	Martelo	128	10,00
26	Serra tico-tico	16	14,25
39	Cortador de grama	10	79,50
56	Serra elétrica	8	89,99
76	Chave de fenda	236	4,99
81	Marreta	32	19,75
88	Chave inglesa	65	6,48

Fig. 16.26 Dados para o Exercício 16.10.

16.11 (*Gerador de palavra de número de telefone*) Os teclados de telefone padrão contêm os dígitos de 0 a 9. Os números 2 a 9 têm três letras associadas a cada número (veja Fig. 16.27).

Muitas pessoas acham difícil memorizar números de telefone, assim utilizam a correspondência entre dígitos e letras para criar palavras de sete letras que correspondam aos seus números de telefone. Por exemplo, uma pessoa cujo número de telefone é 686-2377 talvez utilize a correspondência indicada na Fig. 16.27 para desenvolver a palavra de sete letras "NUMBERS". Cada palavra de sete letras corresponde a exatamente um número de telefone de sete dígitos. O restaurante que deseja aumentar seu negócio de entregas a domicílio ("takeout", em inglês) seguramente poderia fazer isso com o número 825-3688 (isto é, "TAKEOUT").

Dígitos	Letras
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P R S
8	T U V
9	W X Y

Fig. 16.17 Dígitos e letras do teclado do telefone.

Cada número de telefone de sete letras corresponde a muitas palavras separadas de sete letras. Infelizmente, a maioria delas representam justaposições irreconhecíveis de letras. É possível, entretanto, que o proprietário de um salão de cabeleireiro ficasse satisfeito em saber que o número de telefone de seu salão, 424-7288, corresponde a “HAIRCUT” (corte de cabelo, em inglês). O proprietário de uma loja de bebidas sem dúvida ficaria encantado em descobrir que o número de telefone da loja, 233-7226, corresponde a “BEERCAN” (lata de cerveja, em inglês). O veterinário que tem o número de telefone 738-2273 gostaria de saber que seu número corresponde à palavra de sete letras “PETCARE” (cuidado de animais de estimação). O vendedor de automóveis ficaria satisfeito em saber que o número de telefone de sua loja, 639-2277, corresponde a “NEWCARS” (carros novos).

Escreva um programa que, dado um número de sete dígitos, grava em um arquivo todas as combinações possíveis de palavra de sete letras correspondentes àquele número. Há 2187 (3^7) dessas palavras. Evite números de telefone com os dígitos 0 e 1.

17

Redes

Objetivos

- Entender os elementos de redes de Java com URIs, soquetes e datagramas.
- Implementar aplicativos de redes em Java utilizando soquetes e datagramas.
- Entender como implementar clientes e servidores Java que se comuniquem entre si.
- Entender como implementar aplicativos colaboradores baseados em redes.
- Construir um servidor com múltiplas *threads*.

Se a presença da eletricidade pode se tornar visível em qualquer parte de um circuito, não vejo razão por que a inteligência não possa ser transmitida instantaneamente pela eletricidade.

Samuel F. B. Morse

Sr. Watson, venha aqui, preciso de você.

Alexander Graham Bell

O que as redes de ferrovias, rodovias e canais foram em outra era, as redes de telecomunicações, informações e computação... são hoje em dia.

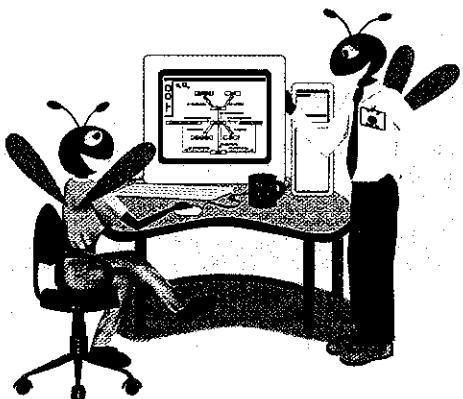
Bruno Kreisky, chanceler austriaco

Talvez a ciência nunca descubra um sistema de comunicação em escritórios melhor que o intervalo para o café.

Earl Wilson

... atualmente a questão é como acessar gigabits de informações através de linhas de velocidade ridícula.

J. C. R. Licklider



Sumário do capítulo

- 17.1 Introdução
- 17.2 Manipulando URLs
- 17.3 Lendo um arquivo em um servidor da Web
- 17.4 Estabelecendo um servidor simples com soquetes de fluxo
- 17.5 Estabelecendo um cliente simples com soquetes de fluxo
- 17.6 Interacão cliente/servidor com conexões de soquete de fluxo
- 17.7 Interacão cliente/servidor sem conexão com datagramas
- 17.8 Tic-Tac-Toe cliente/servidor utilizando um servidor com múltiplas threads
- 17.9 Segurança e a rede
- 17.10 Servidor e cliente de bate-papo DeitelMessenger
 - 17.10.1 DeitelMessengerServer e classes de suporte
 - 17.10.2 Cliente DeitelMessenger e classes de suporte
- 17.11 (Opcional) Descobrindo padrões de projeto: padrões de projeto utilizados nos pacotes `java.io` e `java.net`
 - 17.11.1 Padrões de criação de projeto
 - 17.11.2 Padrões estruturais de projeto
 - 17.11.3 Padrões de arquitetura de projeto
 - 17.11.4 Conclusão

Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão • Exercícios

17.1 Introdução

Existe muito entusiasmo com a Internet e a World Wide Web. A Internet une o “mundo das informações”. A World Wide Web torna a Internet fácil de usar e lhe dá o estilo e a animação da multimídia. As empresas vêem a Internet e a Web como cruciais para suas estratégias de sistemas de informações. Java oferece diversos recursos primitivos para uso de redes que tornam fácil desenvolver aplicativos baseados na Internet e na Web. Java não só pode especificar paralelismo através de *multithreading*, como pode habilitar os programas a procurar informações pelo mundo e colaborar com os programas que estão sendo executados em outros computadores internacionalmente, nacionalmente ou apenas dentro de uma organização. Java pode permitir que os *applets* e aplicativos se comuniquem uns com os outros (sujeito a restrições de segurança).

As redes são um tópico enorme e complexo. Os estudantes de ciência da computação e de engenharia da computação normalmente terão um curso de um semestre inteiro, de nível superior, sobre redes de computadores, e continuarão com estudos adicionais em nível de pós-graduação. Java oferece um rico complemento de recursos para redes e provavelmente será usada como veículo de implementação em cursos sobre redes de computadores. Em *Java Como Programar, Quarta Edição*, apresentamos uma parte dos conceitos e recursos das redes em Java. Para conhecer os recursos de redes mais avançados, consulte nosso livro *Advanced Java 2 Platform How to Program*.

Os recursos para redes em Java estão agrupados em diversos pacotes. Os recursos fundamentais são definidos por classes e interfaces do pacote `java.net`, através dos quais Java oferece *comunicações baseadas em soquetes*, que permitem que os aplicativos vejam as redes como fluxos de dados. As classes e interfaces do pacote `java.net` também oferecem *comunicações baseadas em pacotes* que permitem que os pacotes de informações sejam transmitidos – este é o processo geralmente usado para transmitir áudio e vídeo através da Internet. Neste capítulo, mostramos como criar e manipular soquetes e como se comunicar com pacotes de dados.

Nossa discussão sobre redes focaliza os dois lados de um *relacionamento cliente/servidor*. O *cliente* pede que alguma ação seja executada e o *servidor* executa a ação e responde para o cliente. Uma implementação comum do modelo pedido-resposta é entre navegadores e servidores da World Wide Web. Quando um usuário seleciona um si-

te da Web para navegar através de um navegador (o aplicativo cliente), é enviado um pedido para o servidor da Web apropriado (o aplicativo servidor). O servidor normalmente responde para o cliente enviando uma página da Web em HTML.

Demonstramos o componente GUI do Swing **JEditorPane** e sua capacidade de exibir um documento HTML baixado da World Wide Web. Também apresentamos as *comunicações baseadas em soquetes* de Java, que permitem aos aplicativos visualizar a rede como se ela fosse E/S com arquivos – o programa pode ler de um *soquete* ou gravar em um soquete tão simplesmente quanto lê de um arquivo ou grava em um arquivo. Mostraremos aqui como criar e manipular soquetes.

Java fornece *soquetes de fluxo* e *soquetes de datagrama*. Com *soquetes de fluxo*, o processo estabelece uma *conexão* com outro processo. Enquanto a conexão estiver estabelecida, os dados fluem entre os processos em *fluxos* contínuos. Dizemos que os soquetes de fluxo fornecem um *serviço orientado para conexão*. O protocolo utilizado para transmissão é o popular *TCP (Transmission Control Protocol)*.

Com *soquetes de datagrama*, são transmitidos *pacotes* individuais de informações. Esse não é o protocolo certo para os usuários rotineiros porque, ao contrário do TCP, o protocolo utilizado, *UDP (User Datagram Protocol)* é um *serviço sem conexão* e não garante que os pacotes cheguem em qualquer ordem particular. Na verdade, os pacotes podem ser perdidos, duplicados e até chegar fora da seqüência. Assim, com o UDP, é necessário que o usuário possua uma programação extra significativa para lidar com esses problemas (se o usuário optar por fazer isso). O UDP é mais apropriada para os aplicativos de redes que não exigem a verificação de erros e a confiabilidade do TCP. Os soquetes de fluxo e o protocolo TCP serão os mais desejáveis para a ampla maioria dos programadores de Java.

Dica de desempenho 17.1



Os serviços sem conexão geralmente oferecem desempenho maior mas menos confiabilidade que os serviços orientados para conexão.

Dica de portabilidade 17.1



O protocolo TCP e seu conjunto relacionado de protocolos permitem a intercomunicação de uma grande variedade de sistemas heterogêneos de computadores (isto é, sistemas de computadores com processadores diferentes e sistemas operacionais diferentes).

O capítulo termina com um estudo de caso no qual implementamos um aplicativo cliente/servidor de bate-papo semelhante aos serviços de mensagens instantâneas populares na Web hoje em dia. O programa incorpora muitas técnicas para redes apresentadas neste capítulo. O programa também apresenta o *multicasting*, com o qual um servidor pode publicar informações e os clientes podem subscrevê-las. Cada vez que o servidor publica mais informações, todos os subscritores as recebem. Através dos exemplos deste capítulo, veremos que muitos dos detalhes das redes são tratados pelas classes de Java que usamos.

17.2 Manipulando URIs

A Internet oferece muitos protocolos. O protocolo *Hypertext Transfer Protocol (HTTP)* que forma a base da World Wide Web utiliza URIs (*Uniform Resource Identifiers*) para localizar dados na Internet. Os URIs são freqüentemente chamadas de *URLs (Uniform Resource Locators)*. Na verdade, o URL é um tipo de URI. Os URIs comuns fazem referência a arquivos ou diretórios e podem fazer referência a objetos que executam tarefas complexas, como pesquisas em banco de dados e pesquisas na Internet. Se você conhece o URI de arquivos HTML publicamente disponíveis em qualquer lugar na World Wide Web, você pode acessar esses dados através do HTTP. Java facilita a manipulação de URIs. Utilizar um URI que faz referência ao local exato de um recurso (como uma página da Web) como argumento para o método **showDocument** da interface **AppletContext** faz com que o navegador em que o applet está sendo executado exiba o recurso que está no URI especificado. O *applet* das Figs. 17.1 e 17.2 permite que o usuário selecione uma página da Web de uma **JList** e faz com que o navegador exiba a página correspondente.

Esse *applet* tira proveito dos *parâmetros de applet* especificados no documento HTML que invoca o *applet*. Ao navegar pela World Wide Web, você freqüentemente encontrará *applets* que são de domínio público – você pode usá-los gratuitamente em suas próprias páginas da Web (normalmente em troca de mencionar o criador do *applet*). Uma característica comum de tais *applets* é a capacidade de personalizar o *applet* através de parâmetros que são fornecidos a partir do arquivo HTML que invoca o *applet*. Por exemplo, a Fig. 17.1 contém o código HTML que invoca o *applet SiteSelector* na Fig. 17.2. O documento HTML contém oito parâmetros especificados com o *ele-*

mento **param** – estas linhas devem aparecer entre as marcas **applet** de abertura e fechamento. O **applet** pode ler estes valores e usá-los para personalizar a si mesmo. Um número qualquer de marcas **param** pode aparecer entre as marcas **applet** de abertura e fechamento. Cada parâmetro tem um **name** e um **value**. O método **getParameter** de **Applet** recupera o **value** associado a um parâmetro específico e devolve o **value** como **String**. O argumento passado para **getParameter** é um **String** que contém o nome do parâmetro na marca **param**. Neste exemplo, os parâmetros representam o título de cada *site* da Web que o usuário pode selecionar e a localização de cada *site*. Um número qualquer de parâmetros pode ser especificado. Entretanto, estes parâmetros devem ser chamados de **title#**, onde o valor de # começa em 0 e é incrementado de um para cada novo título. Cada título deve ter um parâmetro de localização correspondente, no formato **location#**, onde o valor de # inicia em 0 e é incrementado de um para cada nova localização. A instrução

```
String title = getParameter( "title0" );
```

obtém o valor associado com o parâmetro "title0" e o atribui à referência para **String title**. Se não existe uma marca **param** que contém o parâmetro especificado, **getParameter** devolve **null**.

O **applet** (Fig. 17.2) obtém do documento HTML (Fig. 17.1) as opções que serão exibidas na **JList** do **applet**. A classe **SiteSelector** utiliza uma **Hashtable** (pacote **java.util**) para armazenar os nomes de *sites* da World Wide Web e URIs. A **Hashtable** armazena pares *chave/valor*. O programa usa a *chave* para armazenar e recuperar o *valor* associado na **Hashtable**. Nesse exemplo, a *chave* é o **String** na **JList** que representa o nome do *site* da Web e o valor é um objeto **URL** que armazena o URI do *site* para exibir no navegador. A classe **Hashtable** fornece dois métodos de importância nesse exemplo – **put** e **get**. O método **put** recebe dois argumentos – uma chave e seu valor associado – e coloca o valor na **Hashtable** em uma localização determinada pela chave. O método **get** recebe um argumento – uma chave – e recupera o valor (como referência **Object**) associado à chave. A classe **SiteSelector** também contém um **Vector** (pacote **java.util**) em que os nomes de *sites* são colocados de modo que possam ser utilizados para inicializar a **JList** (uma versão do construtor **JList** recebe um objeto **Vector**). O **Vector** é um *array* de **Objects** dinamicamente redimensionável. A classe **Vector** fornece o método **add** para adicionar um novo elemento ao fim do **Vector**. As classes **Hashtable** e **Vector** são discutidas em detalhes no Capítulo 20.

```

1 <html>
2 <title>Site Selector</title>
3 <body>
4   <applet code = "SiteSelector.class" width = "300" height = "75">
5     <param name = "title0" value = "Java Home Page">
6     <param name = "location0" value = "http://java.sun.com/">
7     <param name = "title1" value = "Deitel">
8     <param name = "location1" value = "http://www.deitel.com/">
9     <param name = "title2" value = "JGuru">
10    <param name = "location2" value = "http://www.jGuru.com/">
11    <param name = "title3" value = "JavaWorld">
12    <param name = "location3" value = "http://www.javaworld.com/">
13  </applet>
14 </body>
15 </html>
```

Fig. 17.1 Documento HTML para carregar o *applet* **SiteSelector**.

As linhas 23 e 24 no método **init** (linhas 20 a 63) criam os objetos **Hashtable** e **Vector**. A linha 27 chama nosso método utilitário **getSitesFromHTMLParameters** (linhas 66 a 108) para obter os parâmetros de HTML do documento HTML que invocou o *applet*.

No método **getSitesFromHTMLParameters**, a linha 75 utiliza o método **getParameter** de **Applet** para obter um título de *site* da Web. Se o **title** não for **null**, o laço nas linhas 78 a 106 começa a ser executado. A linha 81 utiliza o método **getParameter** de **Applet** para obter a localização correspondente. A linha 87 utiliza a **location** como valor inicial de um novo objeto **URL**. O construtor **URL** determina se o **String** passado como argumento representa um URI válido. Se não for, o construtor **URL** dispara uma **MalformedURLException**. Observe que o construtor **URL** deve ser chamado em um bloco **try**. Se o construtor **URL** gerar uma **MalformedURLException**, a chamada para **printStackTrace** (linha 98) faz com que o programa exiba um monitoramento de pilha.

Então, o programa tenta obter o próximo título de *site* da Web. O programa não adiciona o *site* correspondente ao URI inválido à **Hashtable**, de modo que o título não será exibido na **JList**.



Erro comum de programação 17.1

Uma MalformedURLException é disparada quando um String que não está no formato adequado de URI é passado para um construtor URL.

Para um **URL** apropriado, a linha 90 coloca o **title** e o **URL** na **Hashtable** e a linha 93 adiciona o **title** ao **Vector**. A linha 104 obtém o próximo título do documento HTML. Quando a chamada a **getParameter** na linha 104 devolve **null**, o laço termina.

Quando o método **getSitesFromHTMLParameters** devolve para **init**, as linhas 30 a 61 constroem a GUI do *applet*. As linhas 31 e 32 adicionam o **JLabel** “Choose a site to browse” ao **NORTH** do **BorderLayout** do painel de conteúdo. As linhas 36 a 58 registram uma instância de uma classe interna anônima que implementa **ListSelectionListener** para tratar os eventos do **siteChooser**. As linhas 60 e 61 adicionam **siteChooser** ao **CENTER** do **BorderLayout** do painel de conteúdo.

```

1 // Fig. 17.2: SiteSelector.java
2 // Este programa usa um botão para carregar o documento de um URL.
3
4 // Pacotes do núcleo de Java
5 import java.net.*;
6 import java.util.*;
7 import java.awt.*;
8 import java.applet.AppletContext;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12 import javax.swing.event.*;
13
14 public class SiteSelector extends JApplet {
15     private Hashtable sites;    // nomes e URLs dos sites
16     private Vector siteNames;  // nomes dos sites
17     private JList siteChooser; // lista de sites para escolher
18
19     // lê parâmetros de HTML e configura a GUI
20     public void init()
21     {
22         // cria a Hashtable e o Vector
23         sites = new Hashtable();
24         siteNames = new Vector();
25
26         // obtém parâmetros do documento HTML
27         getSitesFromHTMLParameters();
28
29         // cria componentes da GUI e define layout da interface
30         Container container = getContentPane();
31         container.add( new JLabel( "Choose a site to browse" ),
32                         BorderLayout.NORTH );
33
34         siteChooser = new JList( siteNames );
35
36         siteChooser.addListSelectionListener(
37             new ListSelectionListener() {
38
39                 // vai para o site que o usuário selecionou
40                 public void valueChanged( ListSelectionEvent event )
```

Fig. 17.2 Carregando um documento a partir de um URL para um navegador (parte 1 de 3).

```

42         {
43             // obtém o nome do site selecionado
44             Object object = siteChooser.getSelectedValue();
45
46             // usa o nome do site para localizar o URL correspondente
47             URL newDocument = ( URL ) sites.get( object );
48
49             // obtém referência para o contêiner de applets
50             AppletContext browser = getAppletContext();
51
52             // diz ao contêiner de applets para mudar de página
53             browser.showDocument( newDocument );
54         } // fim do método valueChanged
55
56     } // fim da classe interna anônima
57
58 ); // fim da chamada para addListSelectionListener
59
60 container.add( new JScrollPane( siteChooser ),
61     BorderLayout.CENTER );
62
63 } // fim do método init
64
65 // obtém parâmetros do documento HTML
66 private void getSitesFromHTMLParameters()
67 {
68     // procura os parâmetros do applet no documento HTML
69     // e adiciona os sites à Hashtable
70     String title, location;
71     URL url;
72     int counter = 0;
73
74     // obtém o título do primeiro site
75     title = getParameter( "title" + counter );
76
77     // repete o laço até que não haja mais parâmetros no documento HTML
78     while ( title != null ) {
79
80         // obtém a posição do site
81         location = getParameter( "location" + counter );
82
83         // coloca título/URL na Hashtable e o título no Vector
84         try {
85
86             // converte a posição em um URL
87             url = new URL( location );
88
89             // coloca título/URL na Hashtable
90             sites.put( title, url );
91
92             // coloca título no Vector
93             siteNames.add( title );
94         }
95
96         // processa formato de URL inválido
97         catch ( MalformedURLException urlException ) {
98             urlException.printStackTrace();
99         }
100
101         ++counter;

```

Fig. 17.2 Carregando um documento a partir de um URL para um navegador (parte 2 de 3).

```

102
103     // obtém o título do próximo site
104     title = getParameter( "title" + counter );
105
106 } // fim do while
107
108 } // fim do método getSitesFromHTMLParameters
109
110 } // fim da classe SiteSelector

```



Fig. 17.2 Carregando um documento a partir de um URL para um navegador (parte 3 de 3).

Quando o usuário seleciona um dos *sites* da Web em `siteChooser`, o programa chama o método `valueChanged` (linhas 41 a 54). A linha 44 obtém da `JList` o nome do *site* selecionado. A linha 47 passa o nome do site selecionado (a *chave*) para o método `get` de `Hashtable`, que localiza e devolve uma referência `Object` para o objeto `URL` correspondente (o *valor*). O operador de coerção `URL` converte a referência em um `URL` que pode ser atribuído para se referir ao `newDocument`.

A linha 50 utiliza o método `getAppletContext` de `Applet` para obter uma referência a um objeto `AppletContext` que representa o contêiner de *applets*. A linha 53 utiliza a referência para `AppletContext` `browser` para invocar o método `showDocument` de `AppletContext`, que recebe um objeto `URL` como argumento e o passa para o `AppletContext` (isto é, o navegador). O navegador exibe o recurso da World Wide Web associado àquele `URL`. Nesse exemplo, todos os recursos são documentos HTML.

Para os programadores familiarizados com *frames de HTML*, há uma segunda versão do método `showDocument` de `AppletContext` que permite a um *applet* especificar a chamada `frame-alvo` (*target frame*), na qual o recurso da World Wide Web deve ser exibido. A segunda versão de `showDocument` recebe dois argumentos – um objeto `URL` que especifica o recurso a exibir e um `String` que representa a `frame-alvo`. Há algumas *frames-alvo* especiais que podem ser utilizadas como segundo argumento. A `frame-alvo _blank` resulta em uma nova janela de navegador da Web para exibir o URI. A `frame-alvo _self` especifica que o conteúdo do URI especificado deve ser exibido na mesma `frame` que o *applet* (nesse caso, a página HTML do *applet* é substituída). A `frame-alvo _top` especifica que o navegador deve remover as *frames* atuais da janela do navegador e depois exibir o conteúdo do URI

especificado na janela atual. Para obter mais informações sobre HTML e *frames*, veja o site do *World Wide Web Consortium (W3C)* na Web

<http://www.w3.org>

[Nota: esse *applet* deve ser executado a partir de um navegador da World Wide Web, como o Netscape *Navigator* ou o Microsoft *Internet Explorer*, para ver os resultados da exibição de outra página da Web. O **appletviewer** é capaz apenas de executar *applets* – ele ignora todas as outras marcas de HTML. Se os *sites* da Web no programa contivessem *applets* Java, apenas eles apareceriam no **appletviewer** quando o usuário selecionasse um *site* da Web. Cada *applet* seria executado em uma janela **appletviewer** separada. Dos navegadores mencionados aqui, apenas o Netscape Navigator 6 suporta atualmente os recursos de Java 2. Você precisará utilizar o *Plug-in Java* (discutido no Capítulo 3) para executar esse *applet* no Microsoft *Internet Explorer* ou em versões mais antigas do Netscape Navigator.]

17.3 Lendo um arquivo em um servidor da Web

O aplicativo da Fig. 17.3 utiliza o componente GUI do Swing **JEditorPane** (do pacote `javax.swing`) para exibir o conteúdo de um arquivo em um servidor da Web. O usuário digita o URI no **JTextField** na parte superior da janela e o programa exibe o documento correspondente (se existir) no **JEditorPane**. A classe **JEditorPane** é capaz de exibir tanto texto simples como texto formatado em HTML; portanto, esse aplicativo funciona como um navegador simples da Web. O aplicativo também demonstra como processar **HyperlinkEvents** quando o usuário clica em um *hyperlink* no documento HTML. As capturas de tela na Fig. 17.3 ilustram que o **JEditorPane** pode exibir texto simples (a primeira tela) texto de HTML (a segunda tela). As técnicas mostradas nesse exemplo também podem ser utilizadas em *applets*. Entretanto, os *applets* têm permissão para ler arquivos apenas no servidor do qual o *applet* foi baixado.

```

1 // Fig. 17.3: ReadServerFile.java
2 // Este programa usa um JEditorPane para exibir
3 // o conteúdo de um arquivo em um servidor da Web.
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8 import java.net.*;
9 import java.io.*;
10
11 // Pacotes de extensão de Java
12 import javax.swing.*;
13 import javax.swing.event.*;
14
15 public class ReadServerFile extends JFrame {
16     private JTextField enterField;
17     private JEditorPane contentsArea;
18
19     // configura a GUI
20     public ReadServerFile()
21     {
22         super( "Simple Web Browser" );
23
24         Container container = getContentPane();
25
26         // cria enterField e registra seu ouvinte
27         enterField = new JTextField( "Enter file URL here" );
28
29         enterField.addActionListener(
30             new ActionListener() {

```

Fig. 17.3 Lendo um arquivo ao se abrir uma conexão através de um URL (parte 1 de 3).

```

32
33      // obtém o documento especificado pelo usuário
34      public void actionPerformed( ActionEvent event )
35      {
36          getThePage( event.getActionCommand() );
37      }
38
39  } // fim da classe interna anônima
40
41 ); // fim da chamada para addActionListener
42
43 container.add( enterField, BorderLayout.NORTH );
44
45 // cria contentsArea e registra o ouvinte de HyperlinkEvent
46 contentsArea = new JEditorPane();
47 contentsArea.setEditable( false );
48
49 contentsArea.addHyperlinkListener(
50
51     new HyperlinkListener() {
52
53         // se o usuário clicou no hyperlink, vai para a página especificada
54         public void hyperlinkUpdate( HyperlinkEvent event )
55         {
56             if ( event.getEventType() ==
57                 HyperlinkEvent.EventType.ACTIVATED )
58                 getThePage( event.getURL().toString() );
59         }
60
61     } // fim da classe interna anônima
62
63 ); // fim da chamada para addHyperlinkListener
64
65 container.add( new JScrollPane( contentsArea ),
66     BorderLayout.CENTER );
67
68 setSize( 400, 300 );
69 setVisible( true );
70 }
71
72 // carrega o documento; troca o cursor do mouse para indicar o estado
73 private void getThePage( String location )
74 {
75     // troca o cursor do mouse para WAIT_CURSOR
76     setCursor( Cursor.getPredefinedCursor(
77         Cursor.WAIT_CURSOR ) );
78
79     // carrega o documento na contentsArea
80     // e exibe a posição em enterField
81     try {
82         contentsArea.setPage( location );
83         enterField.setText( location );
84     }
85
86     // processa problemas durante a carga do documento
87     catch ( IOException ioException ) {
88         JOptionPane.showMessageDialog( this,
89             "Error retrieving specified URL",
90             "Bad URL", JOptionPane.ERROR_MESSAGE );

```

Fig. 17.3 Lendo um arquivo ao se abrir uma conexão através de um URL (parte 2 de 3).

```

91      }
92
93     setCursor( Cursor.getPredefinedCursor(
94         Cursor.DEFAULT_CURSOR ) );
95   }
96
97 // começa a execução do aplicativo
98 public static void main( String args[] )
99 {
100    ReadServerFile application = new ReadServerFile();
101
102    application.setDefaultCloseOperation(
103        JFrame.EXIT_ON_CLOSE );
104  }
105
106 } // fim da classe ReadServerFile

```

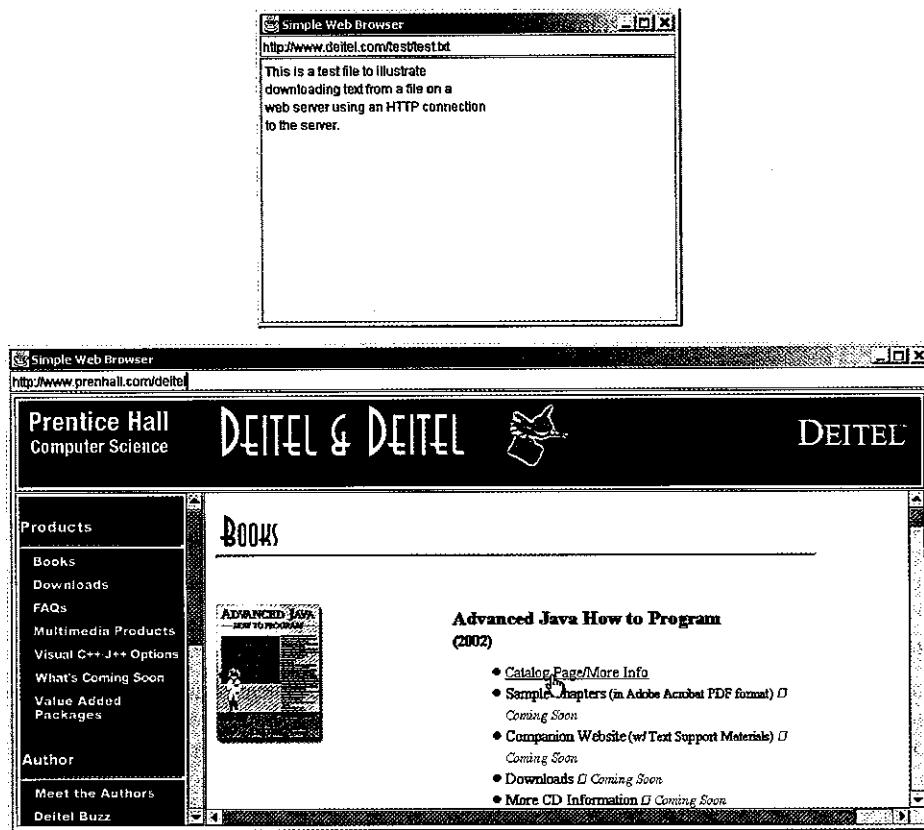


Fig. 17.3 Lendo um arquivo ao se abrir uma conexão através de um URL (parte 3 de 3).

A classe de aplicativo `ReadServerFile` contém o `JTextField enterField` em que o usuário digita o URI do arquivo a ser lido e o `JEditorPane contentsArea` para exibir o conteúdo do arquivo. Quando o usuário pressiona a tecla `Enter` no `JTextField`, o programa chama o método `actionPerformed` (linhas 34 a 37). A linha 36 utiliza o método `getActionCommand` de `ActionEvent` para obter o `String` que o usuário digitou no `JTextField` e passa esse *string* para o método utilitário `getThePage` (linhas 73 a 95).

As linhas 76 e 77 no método `getThePage` utilizam o método `setCursor` (herdado pela classe `JFrame` da classe `Component`) para alterar o cursor do mouse para o *cursor de espera* (normalmente, uma ampulheta ou um relógio). Se o arquivo que está sendo baixado for grande, o cursor de espera indica para o usuário que o programa

está realizando uma tarefa e que ele deve esperar a tarefa ser concluída. O método `static getPredefinedCursor` de `Cursor` recebe um inteiro que indica o tipo de cursor (`Cursor.WAIT_CURSOR` nesse caso). Veja a documentação da API para a classe `Cursor` para obter uma lista completa de cursores.

A linha 82 utiliza o método `setPage` de `JEditorPane` para baixar o documento especificado por `location` e o exibe no `JEditorPane`'s `contentsArea`. Se houver um erro ao baixar o documento, o método `setPage` dispara uma `IOException`. Ademais, se um URI inválido for especificado, ocorre uma `MalformedURLException` (uma subclasse de `IOException`). Se o documento for carregado com sucesso, a linha 83 exibe a localização atualmente indicada em `enterField`.

As linhas 93 e 94 configuram o `Cursor` de volta para `Cursor.DEFAULT_CURSOR` (o `Cursor default`) para indicar que o `download` do documento está completo.

Em geral, um documento HTML contém *hyperlinks* – texto, imagens ou componentes GUI que, ao serem clicados, fornecem acesso rápido para outro documento na Web. Se um `JEditorPane` contém um documento HTML e o usuário clica em um *hyperlink*, o `JEditorPane` gera um `HyperlinkEvent` (pacote `javax.swing.event`) e notifica todos os `HyperlinkListeners` registrados (pacote `javax.swing.event`) daquele evento. As linhas 49 a 63 registram um `HyperlinkListener` para tratar `HyperlinkEvents`. Quando ocorre um `HyperlinkEvent`, o programa chama o método `hyperlinkUpdate` (linhas 54 a 59). As linhas 56 e 57 utilizam o método `getEventType` de `HyperlinkEvent` para determinar o tipo do `HyperlinkEvent`. A classe `HyperlinkEvent` contém a classe interna `public EventType` que define três tipos de evento de `hyperlink`: `ACTIVATED` (o usuário clicou em um *hyperlink* para mudar de página na Web), `ENTERED` (o usuário moveu o mouse sobre um *hyperlink*) e `EXITED` (o usuário removeu o mouse de cima de um *hyperlink*). Se um *hyperlink* for `ACTIVATED`, a linha 58 utiliza o método `getURL` de `HyperlinkEvent` para obter o URL representado pelo *hyperlink*. O método `toString` converte o URL devolvido para um formato de `String` que pode ser passado para o método utilitário `getPage`.



Observação de engenharia de software 17.1

O `JEditorPane` gera `HyperlinkEvents` somente se ele for não-editável.

17.4 Estabelecendo um servidor simples com soquetes de fluxo

Os dois exemplos discutidos até agora usam recursos de alto nível de Java para redes para fazer a comunicação entre aplicativos. Nesses exemplos, não era responsabilidade do programador de Java estabelecer a conexão entre um cliente e um servidor. O primeiro programa confiava no navegador da Web para se comunicar com um servidor da Web. O segundo programa confiava em um `JEditorPane` para fazer a conexão. Esta seção começa nossa discussão sobre como criar seus próprios aplicativos que podem se comunicar uns com os outros.

Estabelecer um servidor simples em Java exige cinco passos. O passo 1 é criar um objeto `ServerSocket`. Uma chamada para o construtor `ServerSocket` como

```
ServerSocket server = new ServerSocket( porta, tamanhoDaFila );
```

registra um número de `porta` disponível e especifica um número máximo de clientes que podem esperar para se conectar ao servidor (isto é, o `tamanhoDaFila`). O número da porta é usado pelos clientes para localizar o aplicativo servidor no computador servidor. Chama-se ponto de *handshake*. Se a fila estiver cheia, o servidor recusa conexões de clientes. A instrução precedente estabelece a porta em que o servidor espera conexões de clientes (um processo conhecido como *vincular o servidor à porta*). Cada cliente solicitará conexão ao servidor nessa `porta`.

Os programas gerenciam cada conexão de cliente com um objeto `Socket`. Depois de vincular o servidor a uma porta com um `ServerSocket` (passo 2), o servidor espera indefinidamente (ou *bloqueia*) uma tentativa de se conectar por parte de um cliente. Para esperar um cliente, o programa chama o método `accept` de `Socket`, como em

```
Socket connection = server.accept();
```

Esta instrução devolve um objeto `Socket` quando uma conexão com um cliente é estabelecida.

O passo 3 é obter os objetos `OutputStream` e `InputStream` que permitem que o servidor se comunique com o cliente enviando e recebendo *bytes*. O servidor envia as informações para o cliente através de um objeto `OutputStream`. O servidor recebe as informações do cliente através de um objeto `InputStream`. Para obter os fluxos, o servidor invoca o método `getOutputStream` sobre o `Socket` para obter uma referência ao `OutputS-`

`tream` associado ao `Socket` e invoca o método `getInputStream` sobre o `Socket` para obter uma referência ao `InputStream` associado ao `Socket`.

Os objetos `OutputStream` e `InputStream` podem ser utilizados para enviar ou receber `bytes` isolados ou conjuntos de `bytes` com o método `write` de `OutputStream` e o método `read` de `InputStream`, respectivamente. Em geral, é útil enviar ou receber valores de tipos primitivos de dados (como `int` e `double`) ou tipos de dados da classe `Serializable` (como `String`) em vez de enviar `bytes`. Nesse caso, podemos utilizar as técnicas do Capítulo 16 para *encadear* outros tipos de fluxos (como `ObjectOutputStream` e `ObjectInputStream`) ao `OutputStream` e ao `InputStream` associados ao `Socket`. Por exemplo,

```
ObjectInputStream input =
    new ObjectInputStream( connection.getInputStream() );

ObjectOutputStream output =
    new ObjectOutputStream( connection.getOutputStream() );
```

A beleza de estabelecer esses relacionamentos é que qualquer coisa que o servidor grave no `ObjectOutputStream` é enviada através do `OutputStream` e está disponível no `InputStream` do cliente, e qualquer coisa que o cliente grave em seu `OutputStream` (com um `ObjectOutputStream` correspondente) está disponível através do `InputStream` do servidor.

O passo 4 é a fase de *processamento* em que o servidor e o cliente se comunicam através dos objetos `InputStream` e `OutputStream`. No passo 5, quando a transmissão está completa, o servidor fecha a conexão invocando o método `close` sobre o `Socket` e os fluxos correspondentes.



Observação de engenharia de software 17.2

Com soquetes, a E/S de rede aparece para os programas Java como idêntica à E/S de arquivos seqüenciais. Os soquetes ocultam do programador muita da complexidade da programação de redes.



Observação de engenharia de software 17.3

Com o multithreading de Java, podemos facilmente criar servidores com múltiplas threads que podem gerenciar muitas conexões simultâneas com muitos clientes; essa arquitetura de servidor com múltiplas threads é precisamente a que é utilizada nos servidores de redes UNIX, Windows NT e OS/2.



Observação de engenharia de software 17.4

Um servidor com múltiplas threads pode receber o `Socket` devolvido por cada chamada a `accept` e criar uma nova thread que gerencia a E/S de rede através daquele `Socket`, ou um servidor com múltiplas threads pode manter um pool de threads (um conjunto de threads já existentes) prontas para gerenciar E/S de rede através dos novos `Sockets` à medida que são criados.



Dica de desempenho 17.2

Nos sistemas de alto desempenho em que a memória é abundante, pode ser implementado um servidor com múltiplas threads para criar um pool de threads que pode ser rapidamente designado para tratar E/S de rede através de cada novo `Socket` quando ele é criado. Assim, quando o servidor recebe uma conexão, ele não precisa incorrer na sobrecarga da criação de thread.

17.5 Estabelecendo um cliente simples com soquetes de fluxo

Estabelecer um cliente simples em Java exige quatro passos. No passo 1, criamos um `Socket` para conectar ao servidor. O construtor de `Socket` estabelece a conexão ao servidor. Por exemplo, a instrução

```
Socket connection = new Socket( endereçoDoServidor, porta );
```

utiliza o construtor `Socket` com dois argumentos – o endereço do servidor na Internet (`endereçoDoServidor`) e o número da `porta`. Se a tentativa de conexão for bem-sucedida, essa instrução devolve um `Socket`. A tentativa de conexão que não for bem-sucedida dispara uma instância de uma subclasse de `IOException`, de modo que muitos programas simplesmente capturam `IOException`. Ocorre uma `UnknownHostException` quando um endereço de servidor indicado por um cliente não pode ser resolvido. A `ConnectException` é disparada quando ocorre um erro durante uma tentativa de conexão a um servidor.

No passo 2, o cliente utiliza os métodos `getInputStream` e `getOutputStream` de `Socket` para obter referências ao `InputStream` e ao `OutputStream` do `Socket`. Como mencionamos na seção precedente, em

geral é útil enviar ou receber valores de tipos primitivos de dados (como `int` e `double`) ou tipos de dados de classe (como `String` e `Employee`) em vez de enviar *bytes*. Se o servidor está enviando informações na forma de tipos de dados reais, o cliente deve receber as informações no mesmo formato. Portanto, se o servidor envia valores com um `ObjectOutputStream`, o cliente deve ler esses valores com um `ObjectInputStream`.

O passo 3 é a fase de processamento em que o cliente e o servidor se comunicam através dos objetos `InputStream` e `OutputStream`. No passo 4, o cliente fecha a conexão quando a transmissão estiver completa, invocando o método `close` sobre o `Socket` e sobre os fluxos correspondentes. Ao processar as informações enviadas por um servidor, o cliente deve determinar quando o servidor terminou de enviar as informações, de modo que o cliente possa chamar `close` para fechar a conexão de `Socket`. Por exemplo, o método `read` de `InputStream` devolve -1 quando ele detecta o fim do fluxo (também chamado EOF – fim do arquivo [*end-of-life*]). Se um `ObjectInputStream` é utilizado para ler as informações do servidor, ocorre uma `EOFException` quando o cliente tenta ler um valor de um fluxo no qual o fim de fluxo é detectado.

17.6 Interação cliente/servidor com conexões de soquete de fluxo

Os aplicativos das Figs. 17.4 e 17.5 utilizam *soquetes de fluxo* para demonstrar um *aplicativo de bate-papo cliente/servidor* simples. O servidor espera uma tentativa de conexão do cliente. Quando um aplicativo cliente se conecta ao servidor, o aplicativo servidor envia para o cliente um objeto `String` (lembre-se de que `Strings` são `Serializable`) indicando que a conexão foi bem-sucedida. Então, o cliente exibe a mensagem. Ambos os aplicativos, cliente e servidor, contêm `JTextFields`, que permitem ao usuário digitar uma mensagem e enviá-la para o outro aplicativo. Quando o cliente ou o servidor envia o `String "TERMINATE"`, a conexão entre o cliente e o servidor termina. Então, o servidor espera o próximo cliente se conectar. A definição da classe `Server` aparece na Fig. 17.4. A definição da classe `Client` aparece na Fig. 17.5. As capturas de tela que mostram a execução entre o cliente e o servidor são mostradas como parte da Fig. 17.5.

O construtor da classe `Server` (linhas 25 a 58) cria a GUI do aplicativo (um `JTextField` e uma `JTextArea`). O objeto `Server` exibe sua saída em uma `JTextArea`. Quando o método `main` (linhas 186 a 194) é executado, ele cria uma instância da classe `Server`, especifica a operação *default* de fechamento da janela e chama o método `runServer` (definido nas linhas 61 a 97).

O método `runServer` faz o trabalho de configurar o servidor para receber uma conexão e processar a conexão quando ela ocorre. O método cria um `ServerSocket` denominado `server` (linha 68) para esperar conexões. O `ServerSocket` é configurado para esperar uma conexão de um cliente na porta 5000. O segundo argumento para o construtor é o número de conexões que podem esperar em uma fila para se conectar ao servidor (100, nesse exemplo). Se a fila estiver cheia quando um cliente tenta se conectar, o servidor recusa a conexão.

```

1 // Fig. 17.4: Server.java
2 // Configura um servidor que irá receber uma conexão
3 // de um cliente, enviar um string para o cliente,
4 // e fechar a conexão.
5
6 // Pacotes do núcleo de Java
7 import java.io.*;
8 import java.net.*;
9 import java.awt.*;
10 import java.awt.event.*;
11
12 // Pacotes de extensão de Java
13 import javax.swing.*;
14
15 public class Server extends JFrame {
16     private JTextField enterField;
17     private JTextArea displayArea;
18     private ObjectOutputStream output;
19     private ObjectInputStream input;
20     private ServerSocket server;

```

Fig. 17.4 A parte do servidor de uma conexão cliente/servidor com soquete de fluxo (parte 1 de 4).

```

21     private Socket connection;
22     private int counter = 1;
23
24     // configura a GUI
25     public Server()
26     {
27         super( "Server" );
28
29         Container container = getContentPane();
30
31         // cria enterField e registra listener
32         enterField = new JTextField();
33         enterField.setEnabled( false );
34
35         enterField.addActionListener(
36
37             new ActionListener() {
38
39                 // envia mensagem para o cliente
40                 public void actionPerformed( ActionEvent event )
41                 {
42                     sendData( event.getActionCommand() );
43                 }
44
45             } // fim da classe interna anônima
46
47         ); // fim da chamada para addActionListener
48
49         container.add( enterField, BorderLayout.NORTH );
50
51         // cria displayArea
52         displayArea = new JTextArea();
53         container.add( new JScrollPane( displayArea ),
54                         BorderLayout.CENTER );
55
56         setSize( 300, 150 );
57         setVisible( true );
58     }
59
60     // configura e executa o servidor
61     public void runServer()
62     {
63         // configura o servidor para receber conexões;
64         // processa conexões
65         try {
66
67             // Etapa 1: cria um ServerSocket.
68             server = new ServerSocket( 5000, 100 );
69
70             while ( true ) {
71
72                 // Etapa 2: espera uma conexão.
73                 waitForConnection();
74
75                 // Etapa 3: obtém fluxos de entrada e de saída.
76                 getStreams();
77
78                 // Etapa 4: Processa a conexão.
79                 processConnection();
80

```

Fig. 17.4 A parte do servidor de uma conexão cliente/servidor com soquete de fluxo (parte 2 de 4).

```

81         // Etapa 5: fecha a conexão.
82         closeConnection();
83
84         ++counter;
85     }
86 }
87
88 // processa EOFException quando o cliente fecha a conexão
89 catch ( EOFException eofException ) {
90     System.out.println( "Client terminated connection" );
91 }
92
93 // processa problemas com E/S
94 catch ( IOException ioException ) {
95     ioException.printStackTrace();
96 }
97 }
98
99 // espera que a conexão chegue, depois exibe informações sobre a conexão
100 private void waitForConnection() throws IOException
101 {
102     displayArea.setText( "Waiting for connection\n" );
103
104     // permite que o servidor aceite uma conexão
105     connection = server.accept();
106
107     displayArea.append( "Connection " + counter +
108         " received from: " +
109         connection.getInetAddress().getHostName() );
110 }
111
112 // obtém fluxos para enviar e receber dados
113 private void getStreams() throws IOException
114 {
115     // configura o fluxo de saída para objetos
116     output = new ObjectOutputStream(
117         connection.getOutputStream() );
118
119     // descarrega o buffer de saída para enviar as informações do cabeçalho
120     output.flush();
121
122     // configura o fluxo de entrada para objetos
123     input = new ObjectInputStream(
124         connection.getInputStream() );
125
126     displayArea.append( "\nGot I/O streams\n" );
127 }
128
129 // processa a conexão com o cliente
130 private void processConnection() throws IOException
131 {
132     // envia mensagem de conexão bem-sucedida para o cliente
133     String message = "SERVER>> Connection successful";
134     output.writeObject( message );
135     output.flush();
136
137     // habilita enterField para que o usuário do servidor possa enviar mensagens
138     enterField.setEnabled( true );
139

```

Fig. 17.4 A parte do servidor de uma conexão cliente/servidor com soquete de fluxo (parte 3 de 4).

```

140     // processa mensagens enviadas a partir do cliente
141     do {
142
143         // lê a mensagem e a exibe
144         try {
145             message = ( String ) input.readObject();
146             displayArea.append( "\n" + message );
147             displayArea.setCaretPosition(
148                 displayArea.getText().length() );
149         }
150
151         // captura problemas ocorridos durante a leitura do cliente
152         catch ( ClassNotFoundException classNotFoundException ) {
153             displayArea.append( "\nUnknown object type received" );
154         }
155
156     } while ( !message.equals( "CLIENT>>> TERMINATE" ) );
157 }
158
159 // fecha fluxos e soquete
160 private void closeConnection() throws IOException
161 {
162     displayArea.append( "\nUser terminated connection" );
163     enterField.setEnabled( false );
164     output.close();
165     input.close();
166     connection.close();
167 }
168
169 // envia mensagem para o cliente
170 private void sendData( String message )
171 {
172     // envia objeto para o cliente
173     try {
174         output.writeObject( "SERVER>>> " + message );
175         output.flush();
176         displayArea.append( "\nSERVER>>>" + message );
177     }
178
179     // processa problemas ocorridos durante o envio do objeto
180     catch ( IOException ioException ) {
181         displayArea.append( "\nError writing object" );
182     }
183 }
184
185 // executa o aplicativo
186 public static void main( String args[] )
187 {
188     Server application = new Server();
189
190     application.setDefaultCloseOperation(
191         JFrame.EXIT_ON_CLOSE );
192
193     application.runServer();
194 }
195
196 } // fim da classe Server

```

Fig. 17.4 A parte do servidor de uma conexão cliente/servidor com soquete de fluxo (parte 4 de 4).



Observação de engenharia de software 17.5

Os números de porta podem estar entre 0 e 65535. Muitos sistemas operacionais reservam números de porta abaixo de 1024 para os serviços de sistema (como correio eletrônico e servidores de World Wide Web). Geralmente, essas portas não devem ser especificadas como portas de conexão em programas de usuários. Na verdade, alguns sistemas operacionais exigem privilégios de acesso especiais para utilizar números de porta abaixo de 1024.

A linha 73 chama o método `waitForConnection` (linhas 100 a 110) para esperar uma conexão de um cliente. Depois que a conexão é estabelecida, a linha 76 chama o método `getStreams` (linhas 113 a 127) para obter referências para o `InputStream` e o `OutputStream` para a conexão. A linha 79 chama o método `processConnection` para enviar a mensagem inicial de conexão para o cliente e processar todas as mensagens recebidas do cliente. A linha 82 chama o método `closeConnection` para terminar a conexão com o cliente.

No método `waitForConnection` (linhas 100 a 110), a linha 105 usa o método `accept` de `ServerSocket` para esperar uma conexão de um cliente e atribui o `Socket` resultante a `connection`. Este método bloqueia até que uma conexão seja recebida (isto é, a `thread` na qual `accept` é chamado pôrás de ser executada até que um cliente se conecte). As linhas 107 a 109 enviam para a saída o nome de `host` do computador que fez a conexão. O método `getInetAddress` devolve um objeto `InetAddress` (pacote `java.net`) contendo as informações sobre o computador cliente. Por exemplo, se o endereço na Internet do computador fosse `127.0.0.1`, o nome de `host` correspondente seria `localhost`.

O método `getStreams` (linhas 113 a 127) obtém as referências para o `InputStream` e para o `OutputStream` do `Socket` e as utiliza para inicializar um `ObjectInputStream` e um `ObjectOutputStream`, respectivamente. Observe a chamada para o método `flush` de `ObjectOutputStream` na linha 120. Esta instrução faz com que o `ObjectOutputStream` no servidor envie um *cabeçalho de fluxo* para o `ObjectInputStream` correspondente no cliente. O cabeçalho de fluxo contém informações como a versão da serialização de objetos que está sendo utilizada para enviar objetos. Este informação é exigida pelo `ObjectInputStream` para que ele se prepare para receber aqueles objetos corretamente.



Observação de engenharia de software 17.6

Ao utilizar um `ObjectOutputStream` e um `ObjectInputStream` para enviar e receber objetos através de uma conexão de rede, sempre crie primeiro o `ObjectOutputStream` e descarregue (com `flush`) o fluxo de modo que o `ObjectInputStream` do cliente possa se preparar para receber os dados. Isto é necessário somente para os aplicativos que se comunicam usando `ObjectInputStream` e `ObjectOutputStream`.

A linha 134 do método `processConnection` (linhas 130 a 157) utiliza o método `writeObject` de `ObjectOutputStream` para enviar o string “ **SERVER>>> Connection successful**” para o cliente. A linha 135 esvazia o fluxo de saída para assegurar que o objeto seja enviado imediatamente; caso contrário, o objeto pode ser armazenado em um `buffer` de saída até mais informações estarem disponíveis para envio.



Dica de desempenho 17.3

Os buffers de saída são geralmente utilizados para aumentar a eficiência de um aplicativo mediante o envio de maiores quantidades de dados em menos vezes. Os componentes de entrada e saída de um computador são, em geral, muito mais lentos que a memória do computador.

A estrutura `do/while` nas linhas 141 a 156 repete o laço até que o servidor receba a mensagem “ **CLIENT>>> TERMINATE**”. A linha 145 utiliza o método `readObject` de `ObjectInputStream` para ler um `String` do cliente. A linha 146 exibe a mensagem na `JTextArea`. As linhas 147 e 148 utilizam o método `setCaretPosition` de `JTextComponent` para posicionar o cursor de entrada na `JTextArea` depois do último caractere na `JTextArea`. Isso faz a `JTextArea` rolar à medida que se acrescenta texto a ela.

Quando a transmissão está completa, o método `processConnection` retorna e o programa chama o método `closeConnection` (linhas 160 a 167) para fechar os fluxos associados com o `Socket` e o `Socket`. Em seguida, o servidor espera a próxima tentativa de conexão de um cliente continuando com a linha 73 no começo do laço `while`.

Quando o usuário do aplicativo servidor insere um `String` no `JTextField` e pressiona a tecla `Enter`, o programa chama o método `actionPerformed` (linhas 40 a 43), lê o `String` do `JTextField` e chama o método utilitário `sendData` (linhas 170 a 183). O método `sendData` envia o objeto `String` para o cliente, esvazia o `buffer` de saída e acrescenta o mesmo `String` à `JTextArea` na janela do servidor.

Observe que o **Server** recebe uma conexão, processa-a, fecha-a e espera a próxima. Um cenário mais provável seria um **Server** que recebe uma conexão, configura essa conexão para ser processada como uma *thread* de execução separada, depois espera novas conexões. As *threads* separadas que processam conexões existentes podem continuar a ser executadas enquanto o **Server** se concentra em novas solicitações de conexão.

Como na classe **Server**, o construtor da classe **Client** (Fig. 17.5) cria a GUI do aplicativo (um **JTextField** e uma **JTextArea**). O objeto **Client** exibe sua saída em uma **JTextArea**. Quando o método **main** (linhas 175 a 188) é executado, ele cria uma instância da classe **Client**, especifica a operação *default* de fechamento da janela e chama o método **runClient** (definido nas linhas 63 a 90). Neste exemplo, você pode executar o cliente a partir de qualquer computador na Internet e especificar o endereço na Internet ou o nome de *host* do computador servidor como argumento de linha de comando para o programa. Por exemplo,

```
java Client 192.168.1.15
conecta-se com o Server no computador com o endereço de Internet 192.168.1.15,
```

```

1 // Fig. 17.5: Client.java
2 // Configura um Client que lerá as informações enviadas
3 // a partir de um Server e exibir as informações.
4
5 // Pacotes do núcleo de Java
6 import java.io.*;
7 import java.net.*;
8 import java.awt.*;
9 import java.awt.event.*;
10
11 // Pacotes de extensão de Java
12 import javax.swing.*;
13
14 public class Client extends JFrame {
15     private JTextField enterField;
16     private JTextArea displayArea;
17     private ObjectOutputStream output;
18     private ObjectInputStream input;
19     private String message = "";
20     private String chatServer;
21     private Socket client;
22
23     // inicializa chatServer e configura a GUI
24     public Client( String host )
25     {
26         super( "Client" );
27
28         // configura o servidor ao qual o cliente se conecta
29         chatServer = host;
30
31         Container container = getContentPane();
32
33         // cria enterField e registra ouvinte
34         enterField = new JTextField();
35         enterField.setEnabled( false );
36
37         enterField.addActionListener(
38             new ActionListener() {
39
40                 // envia mensagem para o servidor
41                 public void actionPerformed( ActionEvent event )
42                 {
43

```

Fig. 17.5 Demonstrando a parte do cliente de uma conexão com soquete de fluxo entre um cliente e um servidor (parte 1 de 5).

```

44         sendData( event.getActionCommand() );
45     }
46
47     } // fim da classe interna anônima
48
49 ); // fim da chamada para addActionListener
50
51 container.add( enterField, BorderLayout.NORTH );
52
53 // cria displayArea
54 displayArea = new JTextArea();
55 container.add( new JScrollPane( displayArea ),
56     BorderLayout.CENTER );
57
58 setSize( 300, 150 );
59 setVisible( true );
60 }
61
62 // conecta-se ao servidor e processa as mensagens do servidor
63 public void runClient()
64 {
65     // conecta ao servidor, obtém fluxos, processa a conexão
66     try {
67
68         // Etapa 1: cria um Socket para fazer a conexão
69         connectToServer();
70
71         // Etapa 2: obtém os fluxos de entrada e de saída
72         getStreams();
73
74         // Etapa 3: processa a conexão
75         processConnection();
76
77         // Etapa 4: fecha a conexão
78         closeConnection();
79     }
80
81     // servidor fechou a conexão
82     catch ( EOFException eofException ) {
83         System.out.println( "Server terminated connection" );
84     }
85
86     // processa problemas na comunicação com o servidor
87     catch ( IOException ioException ) {
88         ioException.printStackTrace();
89     }
90 }
91
92 // obtém fluxos para enviar e receber dados
93 private void getStreams() throws IOException
94 {
95     // configura fluxo de saída para objetos
96     output = new ObjectOutputStream(
97         client.getOutputStream() );
98
99     // descarrega o buffer de saída para enviar informações de cabeçalho
100    output.flush();
101
102    // configura o fluxo de entrada para objetos

```

Fig. 17.5 Demonstrando a parte do cliente de uma conexão com soquete de fluxo entre um cliente e um servidor (parte 2 de 5).

```

103     input = new ObjectInputStream(
104         client.getInputStream() );
105
106     displayArea.append( "\nGot I/O streams\n" );
107 }
108
109 // conecta-se ao servidor
110 private void connectToServer() throws IOException
111 {
112     displayArea.setText( "Attempting connection\n" );
113
114     // cria Socket para fazer a conexão ao servidor
115     client = new Socket(
116         InetAddress.getByName( chatServer ), 5000 );
117
118     // exibe informações sobre a conexão
119     displayArea.append( "Connected to: " +
120         client.getInetAddress().getHostName() );
121 }
122
123 // processa a conexão com o servidor
124 private void processConnection() throws IOException
125 {
126     // habilita enterField para que o usuário possa enviar mensagens
127     enterField.setEnabled( true );
128
129     // processa mensagens enviadas do servidor
130     do {
131
132         // lê mensagem e a exibe
133         try {
134             message = ( String ) input.readObject();
135             displayArea.append( "\n" + message );
136             displayArea.setCaretPosition(
137                 displayArea.getText().length() );
138         }
139
140         // captura problemas na leitura do servidor
141         catch ( ClassNotFoundException classNotFoundException ) {
142             displayArea.append( "\nUnknown object type received" );
143         }
144
145     } while ( !message.equals( "SERVER>>> TERMINATE" ) );
146
147 } // fim do método processConnection
148
149 // fecha fluxos e soquete
150 private void closeConnection() throws IOException
151 {
152     displayArea.append( "\nClosing connection" );
153     output.close();
154     input.close();
155     client.close();
156 }
157
158 // envia mensagem para o servidor
159 private void sendData( String message )
160 {

```

Fig. 17.5 Demonstrando a parte do cliente de uma conexão com soquete de fluxo entre um cliente e um servidor (parte 3 de 5).

```

161     // envia objeto para o servidor
162     try {
163         output.writeObject( "CLIENT>>> " + message );
164         output.flush();
165         displayArea.append( "\nCLIENT>>>" + message );
166     }
167
168     // processa problemas no envio do objeto
169     catch ( IOException ioException ) {
170         displayArea.append( "\nError writing object" );
171     }
172 }
173
174 // executa o aplicativo
175 public static void main( String args[] )
176 {
177     Client application;
178
179     if ( args.length == 0 )
180         application = new Client( "127.0.0.1" );
181     else
182         application = new Client( args[ 0 ] );
183
184     application.setDefaultCloseOperation(
185         JFrame.EXIT_ON_CLOSE );
186
187     application.runClient();
188 }
189
190 } // fim da classe Client

```

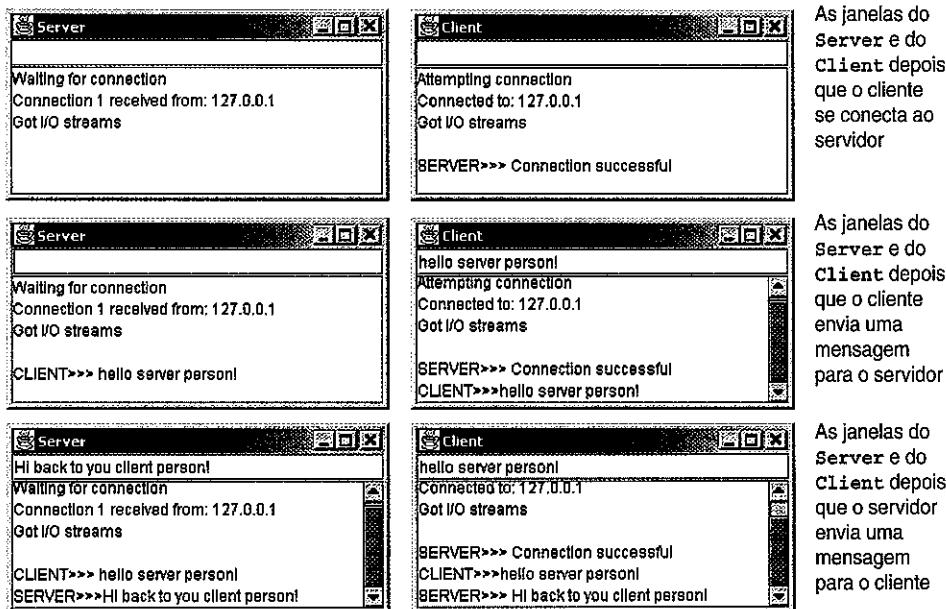


Fig. 17.5 Demonstrando a parte do cliente de uma conexão com soquete de fluxo entre um cliente e um servidor (parte 4 de 5).

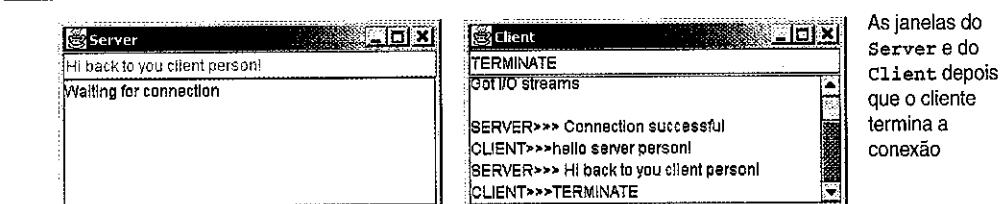


Fig. 17.5 Demonstrando a parte do cliente de uma conexão com soquete de fluxo entre um cliente e um servidor (parte 5 de 5).

O método `runClient` de `Client` (linhas 63 a 90) realiza o trabalho necessário para se conectar ao `Server`, receber-lhe os dados e enviar os dados para lá. A linha 69 chama o método `connectToServer` (linhas 110 a 121) para fazer a conexão. Depois de conectar, a linha 72 chama o método `getStreams` (linhas 93 a 107) para obter referências para os objetos `InputStream` e `OutputStream` do `Socket`. Então, a linha 75 chama o método `processConnection` (124 a 147) para tratar as mensagens enviadas a partir do servidor. Quando a conexão termina, a linha 78 chama `closeConnection` para fechar os fluxos e o `Socket`.

O método `connectToServer` (linhas 110 a 121) cria um `Socket` chamado `client` (linhas 115 e 116) para estabelecer uma conexão. O método passa dois argumentos para o construtor `Socket` – o endereço na Internet do computador servidor e o número da porta (5000) em que esse computador está esperando conexões de clientes. A chamada para o método `static getAddress` de `InetAddress` no primeiro argumento devolve um objeto `InetAddress` que contém o endereço da Internet especificado como argumento de linha de comando para o aplicativo (ou 127.0.0.1 se nenhum argumento de linha de comando for especificado). O método `getByName` pode receber um `String` que contém tanto o endereço de Internet real como o nome de *host* do servidor. O primeiro argumento também poderia ter sido escrito de outras maneiras. Para o endereço 127.0.0.1 do `localhost`, o primeiro argumento poderia ser

```
InetAddress.getByName( "localhost" )
```

ou

```
InetAddress.getLocalHost()
```

Além disso, há versões do construtor `Socket` que recebem um `String` para o endereço da Internet ou o nome de *host*. O primeiro argumento podia ter sido especificado como "127.0.0.1" ou "localhost". [Nota: optamos por demonstrar o relacionamento cliente/servidor fazendo conexão entre os programas que estão sendo executados no mesmo computador (`localhost`). Normalmente, esse primeiro argumento seria o endereço na Internet de outro computador. O objeto `InetAddress` para outro computador pode ser obtido especificando-se o endereço na Internet ou o nome de *host* do outro computador como argumento `String` para `InetAddress.getByName`.]

O segundo argumento do construtor `Socket` é o número da porta do servidor. Esse número deve ser igual ao número da porta em que o servidor está esperando conexões (denominado *ponto de handshake*). Uma vez que a conexão é feita, é exibida na `JTextArea` uma mensagem (linhas 119 e 120) indicando o nome do computador servidor ao qual o cliente se conectou.

O `Client` usa um `ObjectOutputStream` para enviar dados para o servidor e um `ObjectInputStream` para receber dados do servidor. O método `getStreams` (linhas 93 a 107) cria os objetos `ObjectOutputStream` e `ObjectInputStream` que usam os objetos `OutputStream` e `InputStream` associados com `client`.

O método `processConnection` (linhas 124 a 147) contém uma estrutura `do/while` que repete o laço até que o cliente receba a mensagem "SERVER>>> TERMINATE". A linha 134 utiliza o método `readObject` de `ObjectInputStream` para ler um `String` do servidor. A linha 135 exibe a mensagem na `JTextArea`. As linhas 136 e 137 utilizam o método `setCaretPosition` de `JTextComponent` para posicionar o cursor de entrada na `JTextArea` depois do último caractere na `JTextArea`.

Quando a transmissão está completa, o método `closeConnection` (linhas 150 a 156) fecha os fluxos e o `Socket`.

Quando o usuário do aplicativo cliente digita um `String` no `JTextField` e pressiona a tecla *Enter*, o programa chama o método `actionPerformed` (linhas 42 a 45) para ler o `String` do `JTextField` e invoca o método utilitário `sendData` (159 a 172). O método `sendData` envia o objeto `String` para o cliente do servidor, esvazia o *buffer* de saída e acrescenta o mesmo `String` à `JTextArea` na janela do cliente.

17.7 Interação cliente/servidor sem conexão com datagramas

Já discutimos transmissão orientada para conexão e baseada em fluxos. Agora analisaremos a transmissão sem conexão com datagramas.

A transmissão orientada para conexão é como o sistema de telefonia em que você disca e recebe uma conexão ao telefone da pessoa com que você deseja se comunicar. A conexão é mantida durante toda a chamada telefônica, mesmo quando você não estiver conversando.

A transmissão sem conexão com *datagramas* é mais parecida com a maneira como a correspondência é transportada através do serviço postal. Se uma mensagem grande não couber em um envelope, você a divide em partes de mensagem separadas que você coloca em envelopes separados, numerados sequencialmente. Cada uma das cartas é, então, remetida ao mesmo tempo. As cartas podem chegar em ordem, fora da ordem ou simplesmente não chegar (embora o último caso seja raro, acontece). A pessoa na extremidade receptora monta novamente os pedaços da mensagem na ordem sequencial antes de tentar dar sentido à mensagem. Se sua mensagem é suficientemente pequena para caber em um envelope, você não precisa se preocupar com o problema “fora de seqüência”, mas ainda é possível que sua mensagem não chegue. Uma diferença entre os datagramas e o correio postal é que duplicatas de datagramas podem chegar no computador receptor.

Os programas das Figs. 17.6 e 17.7 utilizam datagramas para enviar pacotes de informações entre um aplicativo cliente e um aplicativo servidor. No aplicativo **Client** (Fig. 17.7), o usuário digita uma mensagem em um **JTextField** e pressiona *Enter*. O programa converte a mensagem em um **array byte** e a coloca em um pacote de datagrama que é enviado para o servidor. O **Server** (Fig. 17.6) recebe o pacote e exibe as informações do pacote, então *echoa* o pacote de volta para o cliente. Quando o cliente recebe o pacote, ele exibe as informações do pacote. Nesse exemplo, as classes **Client** e **Server** são implementadas de maneira semelhante.

A classe **Server** (Fig. 17.6) define dois **DatagramPackets** que o servidor utiliza para enviar e receber informações e um **DatagramSocket** que envia e recebe esses pacotes. O construtor para a classe **Server** (linhas 20 a 41) cria a interface gráfica com o usuário na qual os pacotes de informações serão exibidos. Em seguida, o construtor cria o **DatagramSocket** em um bloco **try**. A linha 32 utiliza o construtor **DatagramSocket** que recebe um argumento de número de porta inteiro (5000) para vincular o servidor a uma porta em que o servidor pode receber pacotes de clientes. Os **Clients** que enviam pacotes para esse **Server** especificam a porta 5000 nos pacotes que eles enviam. O construtor **DatagramSocket** dispara uma **SocketException** se não conseguir vincular o **DatagramSocket** a uma porta.



Erro comum de programação 17.2

Especificando uma porta que já está em utilização ou especificando um número de porta inválido ao criar um **DatagramSocket** resulta em uma **BindException**.

```

1 // Fig. 17.6: Server.java
2 // Configura um Server que irá receber pacotes de um
3 // cliente e enviar-lhe pacotes.
4
5 // Pacotes do núcleo de Java
6 import java.io.*;
7 import java.net.*;
8 import java.awt.*;
9 import java.awt.event.*;
10
11 // Pacotes de extensão de Java
12 import javax.swing.*;
13
14 public class Server extends JFrame {
15     private JTextArea displayArea;
16     private DatagramPacket sendPacket, receivePacket;
17     private DatagramSocket socket;
18
19     // configura a GUI e o DatagramSocket

```

Fig. 17.6 Demonstrando o lado do servidor da computação cliente/servidor sem conexão com datagramas (parte 1 de 3).

```

20    public Server()
21    {
22        super( "Server" );
23
24        displayArea = new JTextArea();
25        getContentPane().add( new JScrollPane( displayArea ),
26            BorderLayout.CENTER );
27        setSize( 400, 300 );
28        setVisible( true );
29
30        // cria DatagramSocket para enviar e receber pacotes
31        try {
32            socket = new DatagramSocket( 5000 );
33        }
34
35        // processa problemas ocorridos na criação de DatagramSocket
36        catch( SocketException socketException ) {
37            socketException.printStackTrace();
38            System.exit( 1 );
39        }
40
41    } // fim do construtor Server
42
43    // espera que os pacotes chequem e, depois,
44    // exibe os dados e ecoa o pacote para o cliente
45    public void waitForPackets()
46    {
47        // repete o laço para sempre
48        while ( true ) {
49
50            // recebe o pacote, exibe o conteúdo e ecoa para o cliente
51            try {
52
53                // configura o pacote
54                byte data[] = new byte[ 100 ];
55                receivePacket =
56                    new DatagramPacket( data, data.length );
57
58                // espera o pacote
59                socket.receive( receivePacket );
60
61                // processa o pacote
62                displayPacket();
63
64                // ecoa as informações do pacote de volta para o cliente
65                sendPacketToClient();
66            }
67
68            // processa problemas ocorridos na manipulação do pacote
69            catch( IOException ioException ) {
70                displayArea.append( ioException.toString() + "\n" );
71                ioException.printStackTrace();
72            }
73
74        } // fim do while
75
76    } // fim do método waitForPackets
77

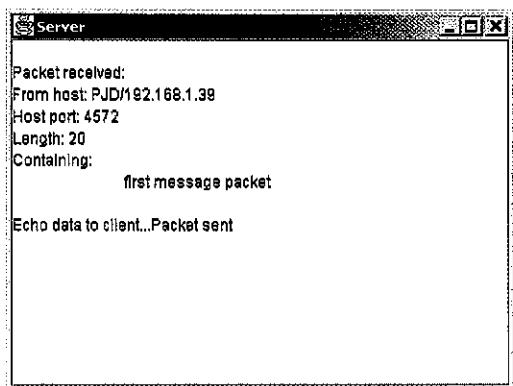
```

Fig. 17.6 Demonstrando o lado do servidor da computação cliente/servidor sem conexão com datagramas (parte 2 de 3).

```

78     // exibe conteúdo do pacote
79     private void displayPacket()
80     {
81         displayArea.append( "\nPacket received:" +
82             "\nFrom host: " + receivePacket.getAddress() +
83             "\nHost port: " + receivePacket.getPort() +
84             "\nLength: " + receivePacket.getLength() +
85             "\nContaining:\n\t" +
86             new String( receivePacket.getData(), 0,
87                 receivePacket.getLength() ) );
88     }
89
90     // ecoa pacote para o cliente
91     private void sendPacketToClient() throws IOException
92     {
93         displayArea.append( "\n\nEcho data to client..." );
94
95         // cria pacote para enviar
96         sendPacket = new DatagramPacket( receivePacket.getData(),
97             receivePacket.getLength(), receivePacket.getAddress(),
98             receivePacket.getPort() );
99
100        // envia pacote
101        socket.send( sendPacket );
102
103        displayArea.append( "Packet sent\n" );
104        displayArea.setCaretPosition(
105            displayArea.getText().length() );
106    }
107
108    // executa o aplicativo
109    public static void main( String args[] )
110    {
111        Server application = new Server();
112
113        application.setDefaultCloseOperation(
114            JFrame.EXIT_ON_CLOSE );
115
116        application.waitForPackets();
117    }
118
119 } // fim da classe Server

```



A janela Server depois que o cliente envia um pacote de dados

Fig. 17.6 Demonstrando o lado do servidor da computação cliente/servidor sem conexão com datagramas (parte 3 de 3).

O método `waitForPackets` de `Server` (linhas 45 a 76) utiliza um laço infinito para esperar que os pacotes cheguem no `Server`. As linhas 54 a 56 criam um `DatagramPacket` em que um pacote de informações recebido pode ser armazenado. O construtor `DatagramPacket` para essa finalidade recebe dois argumentos – um `array` de `byte` contendo os dados e o comprimento do `array` de `byte`. A linha 59 espera um pacote chegar no `Server`. O método `receive` fica bloqueado até um pacote chegar e depois armazena o pacote em seu argumento `DatagramPacket`. O método `receive` dispara uma `IOException` se ocorrer um erro ao se receber um pacote.

Quando chega um pacote, o programa chama o método `displayPacket` (linhas 79 a 88) para acrescentar o conteúdo do pacote a `displayArea`. O método `getAddress` de `DatagramPacket` (linha 82) devolve um objeto `InetAddress` contendo o nome de *host* do computador do qual o pacote foi enviado. O método `getPort` (linha 83) devolve um inteiro que especifica o número da porta através da qual o computador *host* enviou o pacote. O método `getLength` (linha 84) devolve um inteiro que representa o número de *bytes* de dados que foram enviados. O método `getData` (linha 86) devolve um `array` de `byte` contendo os dados que foram enviados. O programa usa o `array` de `byte` para inicializar um objeto `String` de modo que os dados possam ser enviados para saída na `JTextArea`.

Após exibir um pacote, o programa chama o método `sendPacketToClient` (linha 65) para criar um novo pacote e enviá-lo para o cliente. As linhas 96 a 98 criam `sendPacket` e passam quatro argumentos para o construtor `DatagramPacket`. O primeiro argumento especifica o `array` de `byte` a ser enviado. O segundo argumento especifica o número de *bytes* a ser enviado. O terceiro argumento especifica o endereço na Internet do computador cliente para o qual o pacote será enviado. O quarto argumento especifica a porta através da qual o cliente está esperando para receber os pacotes. O método `send` dispara uma `IOException` se ocorrer um erro ao se enviar um pacote.

A classe `Client` (Fig. 17.7) funciona de maneira semelhante à classe `Server`, com a diferença que o `Client` envia pacotes apenas quando o usuário digita uma mensagem em um `JTextField` e pressiona a tecla *Enter*. Quando isso ocorrer, o programa chama o método `actionPerformed` (linhas 34 a 67), que converte o `String` que o usuário digitou no `JTextField` em um `array` de `byte` (linha 45). As linhas 48 a 50 criam um `DatagramPacket` e o inicializam com o `array` de `byte`, o comprimento do `String` que foi digitado pelo usuário, o endereço na Internet para o qual o pacote deve ser enviado (`InetAddress.getLocalHost()`, nesse exemplo) e o número da porta em que o `Server` está esperando os pacotes. A linha 53 envia o pacote. Observe que nesse exemplo o cliente deve saber que o servidor está recebendo pacotes na porta 5000; caso contrário, o servidor não irá receber os pacotes.

```

1 // Fig. 17.7: Client.java
2 // Configura um Client que irá enviar pacotes para
3 // um servidor e receber pacotes de um servidor.
4
5 // Pacotes do núcleo de Java
6 import java.io.*;
7 import java.net.*;
8 import java.awt.*;
9 import java.awt.event.*;
10
11 // Pacotes de extensão de Java
12 import javax.swing.*;
13
14 public class Client extends JFrame {
15     private JTextField enterField;
16     private JTextArea displayArea;
17     private DatagramPacket sendPacket, receivePacket;
18     private DatagramSocket socket;
19
20     // configura a GUI e o DatagramSocket
21     public Client()
22     {
23         super( "Client" );
24
25         Container container = getContentPane();

```

Fig. 17.7 Demonstrando o lado do cliente da computação cliente/servidor sem conexão com datagramas (parte 1 de 4).

```

26
27     enterField = new JTextField( "Type message here" );
28
29     enterField.addActionListener(
30
31         new ActionListener() {
32
33             // cria e envia um pacote
34             public void actionPerformed( ActionEvent event )
35             {
36                 // cria e envia pacote
37                 try {
38                     displayArea.append(
39                         "\nSending packet containing: " +
40                         event.getActionCommand() + "\n" );
41
42                     // obtém mensagem do campo de texto e a converte
43                     // para um array de bytes
44                     String message = event.getActionCommand();
45                     byte data[] = message.getBytes();
46
47                     // cria sendPacket
48                     sendPacket = new DatagramPacket(
49                         data, data.length,
50                         InetAddress.getLocalHost(), 5000 );
51
52                     // envia pacote
53                     socket.send( sendPacket );
54
55                     displayArea.append( "Packet sent\n" );
56                     displayArea.setCaretPosition(
57                         displayArea.getText().length() );
58                 }
59
60                     // processa problemas ocorridos na criação ou no envio do pacote
61                     catch ( IOException ioException ) {
62                         displayArea.append(
63                             ioException.toString() + "\n" );
64                         ioException.printStackTrace();
65                     }
66
67             } // fim do método actionPerformed
68
69         } // fim da classe interna anônima
70
71     ); // fim da chamada para addActionListener
72
73     container.add( enterField, BorderLayout.NORTH );
74
75     displayArea = new JTextArea();
76     container.add( new JScrollPane( displayArea ),
77                     BorderLayout.CENTER );
78
79     setSize( 400, 300 );
80     setVisible( true );
81
82     // cria DatagramSocket para enviar e receber pacotes
83     try {
84         socket = new DatagramSocket();

```

Fig. 17.7 Demonstrando o lado do cliente da computação cliente/servidor sem conexão com datagramas (parte 2 de 4).

```

85      }
86
87      // captura problemas ocorridos na criação de DatagramSocket
88      catch( SocketException socketException ) {
89          socketException.printStackTrace();
90          System.exit( 1 );
91      }
92
93  } // fim do construtor Client
94
95  // espera que os pacotes cheguem do Server,
96  // depois exibe o conteúdo dos pacotes
97  public void waitForPackets()
98  {
99      // repete o laço para sempre
100     while ( true ) {
101
102         // recebe o pacote e exibe o conteúdo
103         try {
104
105             // configura o pacote
106             byte data[] = new byte[ 100 ];
107             receivePacket =
108                 new DatagramPacket( data, data.length );
109
110             // espera o pacote
111             socket.receive( receivePacket );
112
113             // exibe o conteúdo do pacote
114             displayPacket();
115         }
116
117         // processa problemas ocorridos na recepção ou exibição do pacote
118         catch( IOException exception ) {
119             displayArea.append( exception.toString() + "\n" );
120             exception.printStackTrace();
121         }
122
123     } // fim do while
124
125 } // fim do método waitForPackets
126
127 // exibe conteúdo do receivePacket
128 private void displayPacket()
129 {
130     displayArea.append( "\nPacket received:" +
131         "\nFrom host: " + receivePacket.getAddress() +
132         "\nHost port: " + receivePacket.getPort() +
133         "\nLength: " + receivePacket.getLength() +
134         "\nContaining:\n\t" +
135         new String( receivePacket.getData(), 0,
136             receivePacket.getLength() ) );
137
138     displayArea.setCaretPosition(
139         displayArea.getText().length() );
140 }
141
142 // executa o aplicativo
143 public static void main( String args[] )

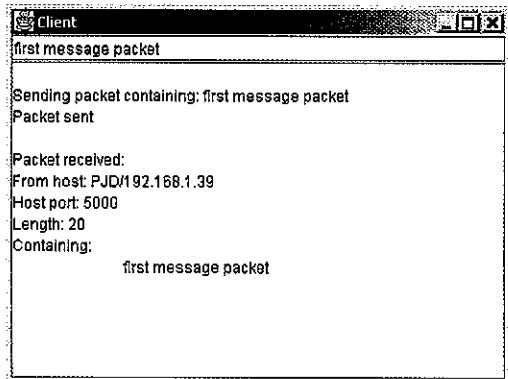
```

Fig. 17.7 Demonstrando o lado do cliente da computação cliente/servidor sem conexão com datagramas (parte 3 de 4).

```

144    {
145        Client application = new Client();
146
147        application.setDefaultCloseOperation(
148            JFrame.EXIT_ON_CLOSE );
149
150        application.waitForPackets();
151    }
152
153 } // fim da classe Client

```



A janela Client após enviar um pacote para o servidor e receber o pacote de volta do servidor

Fig. 17.7 Demonstrando o lado do cliente da computação cliente/servidor sem conexão, com datagramas (parte 4 de 4).

Repare que a chamada para o construtor de `DatagramSocket` (linha 84) nesse aplicativo não especifica quaisquer argumentos. Este construtor permite selecionar o próximo número de porta disponível para o `DatagramSocket`. O cliente não precisa de um número específico de porta porque o servidor recebe o número de porta do cliente como parte de cada `DatagramPacket` enviado pelo cliente. Portanto, o servidor pode enviar pacotes de volta para o mesmo computador e número de porta do qual o servidor recebe um pacote de informações.

O método `waitForPackets` de `Client` (linhas 97 a 125) utiliza um laço infinito para esperar pacotes do servidor. A linha 111 bloqueia até que chegue um pacote. Observe que isso não impede o usuário de enviar um pacote porque os eventos GUI são tratados na *thread* de despacho de eventos. Só impede que o laço `while` continue até que um pacote chegue no `Client`. Quando chega um pacote, a linha 111 armazena o pacote no `receivePacket`, e a linha 114 chama o método `displayPacket` (128 a 140) para exibir o conteúdo do pacote na `JTextArea`.

17.8 Tic-Tac-Toe cliente/servidor utilizando um servidor com múltiplas threads

Nesta seção, apresentamos o popular jogo Tic-Tac-Toe (jogo-da-velha) utilizando técnicas cliente/servidor com sockets de fluxo. O programa consiste em um aplicativo `TicTacToeServer` (Fig. 17.8) que permite que dois *applets* `TicTacToeClient` (Fig. 17.9) se conectem ao servidor e joguem jogo-da-velha (saídas mostradas na Fig. 17.10). À medida que cada conexão de cliente é recebida pelo servidor, ele cria uma instância da classe interna `Player` (linhas 158 a 279 da Fig. 17.8) para processar o cliente em uma *thread* separada. Estas *threads* permitem que os clientes joguem o jogo independentemente. O servidor associa os Xs ao primeiro cliente que se conecta (o X faz o primeiro movimento) e associa os Os ao segundo cliente a se conectar. O servidor mantém as informações sobre o tabuleiro de modo que possa determinar se uma jogada de um dos jogadores é uma jogada válida ou inválida. Cada *applet* `TicTacToeClient` (Fig. 17.9) mantém sua própria versão GUI do tabuleiro de jogo-da-velha no qual ele exibe o estado do jogo. Os clientes podem colocar uma marca apenas em um quadrado vazio no tabuleiro. A classe `Square` (linhas 212 a 270 da Fig. 17.9) implementa cada um dos nove quadrados no tabuleiro. A classe `TicTacToeServer` e a classe `Player` são implementadas no arquivo `TicTacToeServer.java` (Fig. 17.8). A classe `TicTacToeClient` e a classe `Square` são implementadas no arquivo `TicTacToeClient.java` (Fig. 17.9).

```

1 // Fig. 17.8: TicTacToeServer.java
2 // Esta classe mantém um jogo-da-velha
3 // para dois applets clientes.
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8 import java.net.*;
9 import java.io.*;
10
11 // Pacotes de extensão de Java
12 import javax.swing.*;
13
14 public class TicTacToeServer extends JFrame {
15     private byte board[];
16     private JTextArea outputArea;
17     private Player players[];
18     private ServerSocket server;
19     private int currentPlayer;
20
21     // configura servidor de tic-tac-toe e a GUI que exibe mensagens
22     public TicTacToeServer()
23     {
24         super( "Tic-Tac-Toe Server" );
25
26         board = new byte[ 9 ];
27         players = new Player[ 2 ];
28         currentPlayer = 0;
29
30         // configura ServerSocket
31         try {
32             server = new ServerSocket( 5000, 2 );
33         }
34
35         // processa problemas ocorridos na criação do ServerSocket
36         catch( IOException ioException ) {
37             ioException.printStackTrace();
38             System.exit( 1 );
39         }
40
41         // configura a JTextArea para exibir mensagens durante a execução
42         outputArea = new JTextArea();
43         getContentPane().add( outputArea, BorderLayout.CENTER );
44         outputArea.setText( "Server awaiting connections\n" );
45
46         setSize( 300, 300 );
47         setVisible( true );
48     }
49
50     // espera duas conexões para que o jogo possa ser jogado
51     public void execute()
52     {
53         // espera que cada cliente se conecte
54         for ( int i = 0; i < players.length; i++ ) {
55
56             // espera a conexão, cria jogador, dispara thread
57             try {
58                 players[ i ] = new Player( server.accept(), i );
59                 players[ i ].start();
60             }

```

Fig. 17.8 O lado do servidor do programa cliente/servidor do jogo-da-velha (parte 1 de 5).

```

61      // processa problemas ocorridos no recebimento de conexão do cliente
62      catch( IOException ioException ) {
63          ioException.printStackTrace();
64          System.exit( 1 );
65      }
66  }
67 }
68
69 // Jogador X fica suspenso até que o Jogador O se conecte.
70 // Resume o jogador X agora.
71 synchronized ( players[ 0 ] ) {
72     players[ 0 ].setSuspended( false );
73     players[ 0 ].notify();
74 }
75
76 } // fim do método execute
77
78 // exibe uma mensagem na outputArea
79 public void display( String message )
80 {
81     outputArea.append( message + "\n" );
82 }
83
84 // Determina se uma jogada é válida.
85 // Este método é synchronized porque somente
86 // uma jogada pode ser feita de cada vez.
87 public synchronized boolean validMove(
88     int location, int player )
89 {
90     boolean moveDone = false;
91
92     // enquanto não for o jogador atual, deve esperar sua vez
93     while ( player != currentPlayer ) {
94
95         // espera a vez
96         try {
97             wait();
98         }
99
100        // captura interrupções de wait
101        catch( InterruptedException interruptedException ) {
102            interruptedException.printStackTrace();
103        }
104    }
105
106    // se uma posição não estiver ocupada, faz a jogada
107    if ( !isOccupied( location ) ) {
108
109        // configura a jogada no array do tabuleiro
110        board[ location ] =
111            ( byte ) ( currentPlayer == 0 ? 'X' : 'O' );
112
113        // troca o jogador atual
114        currentPlayer = ( currentPlayer + 1 ) % 2;
115
116        // permite que o novo jogador saiba que a jogada ocorreu
117        players[ currentPlayer ].otherPlayerMoved( location );
118
119        // diz ao jogador que está esperando para continuar
120        notify();

```

Fig. 17.8 O lado do servidor do programa cliente/servidor do jogo-da-velha (parte 2 de 5).

```

121         // diz ao jogador que fez a jogada que a jogada era válida
122         return true;
123     }
124
125     // diz ao jogador que fez a jogada que a jogada não era válida
126     else
127         return false;
128     }
129
130     // determina se a posição está ocupada
131     public boolean isOccupied( int location )
132     {
133         if ( board[ location ] == 'X' || board [ location ] == 'O' )
134             return true;
135         else
136             return false;
137     }
138
139     // coloque código neste método para determinar se o jogo terminou
140     public boolean gameOver()
141     {
142         return false;
143     }
144
145
146     // executa o aplicativo
147     public static void main( String args[] )
148     {
149         TicTacToeServer application = new TicTacToeServer();
150
151         application.setDefaultCloseOperation(
152             JFrame.EXIT_ON_CLOSE );
153
154         application.execute();
155     }
156
157     // classe interna privada Player administra cada jogador como uma thread
158     private class Player extends Thread {
159         private Socket connection;
160         private DataInputStream input;
161         private DataOutputStream output;
162         private int playerNumber;
163         private char mark;
164         protected boolean suspended = true;
165
166         // configura a thread Player
167         public Player( Socket socket, int number )
168         {
169             playerNumber = number;
170
171             // especifica a marca do jogador
172             mark = ( playerNumber == 0 ? 'X' : 'O' );
173
174             connection = socket;
175
176             // obtém fluxos de Socket
177             try {
178                 input = new DataInputStream(
179                     connection.getInputStream() );
180                 output = new DataOutputStream(

```

Fig. 17.8 O lado do servidor do programa cliente/servidor do jogo-da-velha (parte 3 de 5).

```

181         connection.getOutputStream() );
182     }
183
184     // processa problemas ocorridos na obtenção de fluxos
185     catch( IOException ioException ) {
186         ioException.printStackTrace();
187         System.exit( 1 );
188     }
189 }
190
191 // envia mensagem dizendo que o outro jogador jogou;
192 // a mensagem contém um String seguido por um int
193 public void otherPlayerMoved( int location )
194 {
195     // envia mensagem indicando a jogada
196     try {
197         output.writeUTF( "Opponent moved" );
198         output.writeInt( location );
199     }
200
201     // processa problemas ocorridos no envio da mensagem
202     catch ( IOException ioException ) {
203         ioException.printStackTrace();
204     }
205 }
206
207 // controla a execução da thread
208 public void run()
209 {
210     // envia mensagem para cliente indicando qual é sua marca (X ou O),
211     // processa mensagens do cliente
212     try {
213         display( "Player " + ( playerNumber == 0 ?
214             'X' : 'O' ) + " connected" );
215
216         // envia a marca do jogador
217         output.writeChar( mark );
218
219         // envia mensagem indicando conexão
220         output.writeUTF( "Player " +
221             ( playerNumber == 0 ? "X connected\n" :
222                 "O connected, please wait\n" ) );
223
224         // se jogador X, espera que outro jogador chegue
225         if ( mark == 'X' ) {
226             output.writeUTF( "Waiting for another player" );
227
228             // espera o jogador O
229             try {
230                 synchronized( this ) {
231                     while ( suspended )
232                         wait();
233                 }
234             }
235
236             // processa interrupções enquanto espera
237             catch ( InterruptedException exception ) {
238                 exception.printStackTrace();
239             }
240

```

Fig. 17.8 O lado do servidor do programa cliente/servidor do jogo-da-velha (parte 4 de 5).

```

241         // envia mensagem dizendo que outro jogador se conectou e
242         // que o jogador X pode fazer uma jogada
243         output.writeUTF(
244             "Other player connected. Your move." );
245     }
246
247     // enquanto o jogo não terminar
248     while ( ! gameOver() ) {
249
250         // obtém a posição da jogada do cliente
251         int location = input.readInt();
252
253         // verifica se a jogada é válida
254         if ( validMove( location, playerNumber ) ) {
255             display( "loc: " + location );
256             output.writeUTF( "Valid move." );
257         }
258         else
259             output.writeUTF( "Invalid move, try again" );
260     }
261
262     // fecha conexão com o cliente
263     connection.close();
264 }
265
266     // processa problemas ocorridos na comunicação com o cliente
267     catch( IOException ioException ) {
268         ioException.printStackTrace();
269         System.exit( 1 );
270     }
271 }
272
273     // configura se a thread está suspensa ou não
274     public void setSuspended( boolean status )
275     {
276         suspended = status;
277     }
278
279 } // fim da classe Player
280
281 } // fim da classe TicTacToeServer

```

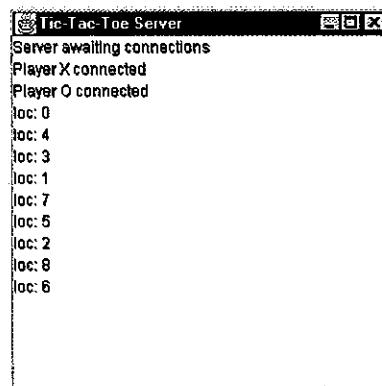


Fig. 17.8 O lado do servidor do programa cliente/servidor do jogo-da-velha (parte 5 de 5).

Começamos com uma discussão do lado do servidor do jogo-da-velha. Quando o aplicativo **TicTacToeServer** é executado, o método **main** (linhas 147 a 155) cria um objeto **TicTacToeServer** chamado de **application**. O construtor (linhas 22 a 48) tenta configurar um **ServerSocket**. Se for bem-sucedido, o programa exibe a janela de servidor e o método **main** invoca o método **execute** de **TicTacToeServer** (linhas 51 a 76). O método **execute** executa o laço duas vezes, ficando bloqueado na linha 58 a cada vez, esperando uma conexão de um cliente. Quando um cliente se conecta, a linha 58 cria um novo objeto **Player** para gerenciar a conexão como uma *thread* separada e a linha 59 chama o método **start** daquele objeto para começar a executar a *thread*.

Quando o **TicTacToeServer** cria um **Player**, o construtor de **Player** (linhas 167 a 189) recebe o objeto **Socket** que representa a conexão com o cliente e obtém os fluxos de entrada e saída associados. O método **run** do **Player** (linhas 208 a 271) controla as informações que são enviadas para o cliente e as informações que são recebidas do cliente. Primeiro, ele diz ao cliente que a conexão do cliente foi estabelecida (linhas 213 e 214), depois ele passa para o cliente o caractere que o cliente colocará no tabuleiro quando uma jogada é feita (linha 217). As linhas 230 a 233 suspendem cada *thread* de **Player** quando ela começa a ser executada, porque nenhum jogador tem permissão para fazer uma jogada logo que ele se conecta. O jogador X só pode jogar quando o jogador 0 se conectar e o jogador 0 só pode jogar depois do jogador X.

Nesse ponto, o jogo pode ser jogado e o método **run** começa executando sua estrutura **while** (linhas 248 a 260). Cada iteração dessa estrutura **while** lê um inteiro (linha 253) que representa a posição em que o cliente quer colocar uma marca e a linha 254 invoca o método **validMove** (linhas 87 a 129) de **TicTacToeServer** para verificar a jogada. As linhas 254 a 259 enviam uma mensagem para o cliente indicando se a jogada foi válida. O programa mantém as posições do tabuleiro como números de 0 a 8 (0 a 2 para a primeira linha, 3 a 5 para a segunda linha e 6 a 8 para a terceira linha).

O método **validMove** (linhas 87 a 129 na classe **TicTacToeServer**) é um método **synchronized** que permite que apenas um jogador de cada vez faça uma jogada. Sincronizar **validMove** impede que os dois jogadores modifiquem as informações do estado do jogo simultaneamente. Se o **Player** que tenta validar uma jogada não é o jogador atual (isto é, aquele que tem permissão para fazer uma jogada), o **Player** é colocado em um estado de *espera* até que chegue a sua vez de jogar. Se a posição para uma jogada que está sendo validada já estiver ocupada no tabuleiro, **validMove** devolve **false**. Caso contrário, o servidor coloca uma marca para o jogador em sua representação local do tabuleiro (linhas 110 e 111), notifica o outro objeto **Player** de que uma jogada foi feita (para que uma mensagem possa ser enviada ao cliente), invoca o método **notify** (linha 120) de modo que o **Player** que está esperando (se houver algum) possa validar uma jogada e devolve **true** (linha 123) para indicar que a jogada é válida.

Quando um *applet* **TicTacToeClient** (Fig. 17.9) começa a ser executado, ele cria uma **JTextArea** em que são exibidas as mensagens do servidor e uma representação do tabuleiro utilizando nove objetos **Square**. O método **start** do *applet* (linhas 80 a 104) abre uma conexão com o servidor e obtém do objeto **Socket** os fluxos de entrada e saída associados. A classe **TicTacToeClient** implementa a interface **Runnable**, de modo que uma *thread* separada possa ler mensagens do servidor. Esta abordagem permite ao usuário interagir com o tabuleiro (na *thread* de despacho de eventos) enquanto está esperando mensagens do servidor. Depois que a conexão com o servidor é estabelecida, a linha 102 cria o objeto **Thread outputThread** e o inicializa com o *applet* **Runnable**, e depois a linha 103 chama o método **start** da *thread*. O método **run** do *applet* (linhas 108 a 137) controla a *thread* de execução separada. O método primeiro lê o caractere de marcação (X ou O) do servidor (linha 112), depois repete um laço continuamente (linhas 123 a 135) e lê mensagens do servidor (linha 127). Cada mensagem é passada para o método **processMessage** do *applet* (linhas 140 a 211) para processamento.

```

1 // Fig. 17.9: TicTacToeClient.java
2 // Cliente para o programa do jogo-da-velha
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.net.*;
8 import java.io.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
```

Fig. 17.9 O lado do cliente do programa cliente/servidor do jogo-da-velha (parte 1 de 6).

```

12
13 // Classe cliente para deixar um usuário jogar o jogo-da-velha
14 // com um outro usuário através de uma rede.
15 public class TicTacToeClient extends JApplet
16     implements Runnable {
17
18     private JTextField idField;
19     private JTextArea displayArea;
20     private JPanel boardPanel, panel2;
21     private Square board[][][], currentSquare;
22     private Socket connection;
23     private DataInputStream input;
24     private DataOutputStream output;
25     private Thread outputThread;
26     private char myMark;
27     private boolean myTurn;
28
29     // configura a interface com o usuário e o tabuleiro
30     public void init()
31     {
32         Container container = getContentPane();
33
34         // configura a JTextArea para exibir mensagens para o usuário
35         displayArea = new JTextArea( 4, 30 );
36         displayArea.setEditable( false );
37         container.add( new JScrollPane( displayArea ),
38             BorderLayout.SOUTH );
39
40         // configura um painel para os quadrados do tabuleiro
41         boardPanel = new JPanel();
42         boardPanel.setLayout( new GridLayout( 3, 3, 0, 0 ) );
43
44         // cria o tabuleiro
45         board = new Square[ 3 ][ 3 ];
46
47         // Ao criar um quadrado, o argumento de posição para
48         // o construtor é um valor de 0 a 8 indicando a posição do
49         // quadrado no tabuleiro. Os valores 0, 1 e 2 são a
50         // primeira linha, os valores 3, 4 e 5 são a segunda
51         // linha. Os valores 6, 7 e 8 são a terceira linha.
52         for ( int row = 0; row < board.length; row++ ) {
53
54             for ( int column = 0;
55                 column < board[ row ].length; column++ ) {
56
57                 // cria Square
58                 board[ row ][ column ] =
59                     new Square( ' ', row * 3 + column );
60
61                 boardPanel.add( board[ row ][ column ] );
62             }
63
64         }
65
66         // campo de texto para exibir a marca do jogador
67         idField = new JTextField();
68         idField.setEditable( false );
69         container.add( idField, BorderLayout.NORTH );
70

```

Fig. 17.9 O lado do cliente do programa cliente/servidor do jogo-da-velha (parte 2 de 6).

```

71     // configura um painel que conterá boardPanel (para fins de leiaute)
72     panel2 = new JPanel();
73     panel2.add( boardPanel, BorderLayout.CENTER );
74     container.add( panel2, BorderLayout.CENTER );
75 }
76
77 // Faz a conexão com o servidor e obtém os fluxos associados.
78 // Inicia thread separada para permitir que este applet
79 // atualize continuamente sua saída na área de texto displayArea.
80 public void start()
81 {
82     // conecta ao servidor, obtém fluxos e inicia a outputThread
83     try {
84
85         // faz a conexão
86         connection = new Socket(
87             InetAddress.getByName( "127.0.0.1" ), 5000 );
88
89         // obtém fluxos
90         input = new DataInputStream(
91             connection.getInputStream() );
92         output = new DataOutputStream(
93             connection.getOutputStream() );
94     }
95
96     // captura problemas ocorridos na configuração da conexão e dos fluxos
97     catch ( IOException ioException ) {
98         ioException.printStackTrace();
99     }
100
101    // cria e inicia a thread de saída
102    outputThread = new Thread( this );
103    outputThread.start();
104 }
105
106 // thread de controle que permite a atualização contínua
107 // da área de texto displayArea
108 public void run()
109 {
110     // obtém a marca do jogador (X ou O)
111     try {
112         myMark = input.readChar();
113         idField.setText( "You are player \""
114             + myMark + "\"" );
115         myTurn = ( myMark == 'X' ? true : false );
116     }
117
118     // processa problemas na comunicação com o servidor
119     catch ( IOException ioException ) {
120         ioException.printStackTrace();
121     }
122
123     // recebe mensagens enviadas para o cliente e as exibe
124     while ( true ) {
125
126         // lê a mensagem do servidor e processa a mensagem
127         try {
128             String message = input.readUTF();
129             processMessage( message );
130         }

```

Fig. 17.9 O lado do cliente do programa cliente/servidor do jogo-da-velha (parte 3 de 6).

```

131      // processa problemas na comunicação com o servidor
132      catch ( IOException ioException ) {
133          ioException.printStackTrace();
134      }
135  }
136
137 } // fim do método run
138
139 // processa mensagens recebidas pelo cliente
140 public void processMessage( String message )
141 {
142     // ocorreu jogada válida
143     if ( message.equals( "Valid move." ) ) {
144         displayArea.append( "Valid move, please wait.\n" );
145
146     // configura marca no quadrado a partir da thread de despacho de eventos
147     SwingUtilities.invokeLater(
148
149         new Runnable() {
150
151             public void run()
152             {
153                 currentSquare.setMark( myMark );
154             }
155
156         }
157
158     ); // fim da chamada para invokeLater
159 }
160
161 // ocorreu uma jogada inválida
162 else if ( message.equals( "Invalid move, try again" ) ) {
163     displayArea.append( message + "\n" );
164     myTurn = true;
165 }
166
167 // o oponente jogou
168 else if ( message.equals( "Opponent moved" ) ) {
169
170     // obtém a posição da jogada e atualiza o tabuleiro
171     try {
172         final int location = input.readInt();
173
174         // configura a marca no quadrado a partir da thread de despacho
175         // de eventos
176         SwingUtilities.invokeLater(
177
178             new Runnable() {
179
180                 public void run()
181                 {
182                     int row = location / 3;
183                     int column = location % 3;
184
185                     board[ row ][ column ].setMark(
186                         ( myMark == 'X' ? 'O' : 'X' ) );
187                     displayArea.append(
188                         "Opponent moved. Your turn.\n" );
189                 }
190             }
191         );
192     }
193
194     // configura a marca no quadrado a partir da thread de despacho
195     // de eventos
196     SwingUtilities.invokeLater(
197
198         new Runnable() {
199
200             public void run()
201             {
202                 int row = location / 3;
203                 int column = location % 3;
204
205                 board[ row ][ column ].setMark(
206                     ( myMark == 'O' ? 'X' : 'O' ) );
207                 displayArea.append(
208                     "Opponent moved. Your turn.\n" );
209             }
210         }
211     );
212
213     myTurn = false;
214
215     displayArea.append( "Opponent moved. Your turn.\n" );
216
217     SwingUtilities.invokeLater(
218
219         new Runnable() {
220
221             public void run()
222             {
223                 int row = location / 3;
224                 int column = location % 3;
225
226                 board[ row ][ column ].setMark(
227                     ( myMark == 'X' ? 'O' : 'X' ) );
228                 displayArea.append(
229                     "Opponent moved. Your turn.\n" );
230             }
231         }
232     );
233
234     myTurn = true;
235
236     displayArea.append( "Opponent moved. Your turn.\n" );
237
238     SwingUtilities.invokeLater(
239
240         new Runnable() {
241
242             public void run()
243             {
244                 int row = location / 3;
245                 int column = location % 3;
246
247                 board[ row ][ column ].setMark(
248                     ( myMark == 'O' ? 'X' : 'O' ) );
249                 displayArea.append(
250                     "Opponent moved. Your turn.\n" );
251             }
252         }
253     );
254
255     myTurn = false;
256
257     displayArea.append( "Opponent moved. Your turn.\n" );
258
259     SwingUtilities.invokeLater(
260
261         new Runnable() {
262
263             public void run()
264             {
265                 int row = location / 3;
266                 int column = location % 3;
267
268                 board[ row ][ column ].setMark(
269                     ( myMark == 'X' ? 'O' : 'X' ) );
270                 displayArea.append(
271                     "Opponent moved. Your turn.\n" );
272             }
273         }
274     );
275
276     myTurn = true;
277
278     displayArea.append( "Opponent moved. Your turn.\n" );
279
280     SwingUtilities.invokeLater(
281
282         new Runnable() {
283
284             public void run()
285             {
286                 int row = location / 3;
287                 int column = location % 3;
288
289                 board[ row ][ column ].setMark(
290                     ( myMark == 'O' ? 'X' : 'O' ) );
291                 displayArea.append(
292                     "Opponent moved. Your turn.\n" );
293             }
294         }
295     );
296
297     myTurn = false;
298
299     displayArea.append( "Opponent moved. Your turn.\n" );
300
301     SwingUtilities.invokeLater(
302
303         new Runnable() {
304
305             public void run()
306             {
307                 int row = location / 3;
308                 int column = location % 3;
309
310                 board[ row ][ column ].setMark(
311                     ( myMark == 'X' ? 'O' : 'X' ) );
312                 displayArea.append(
313                     "Opponent moved. Your turn.\n" );
314             }
315         }
316     );
317
318     myTurn = true;
319
320     displayArea.append( "Opponent moved. Your turn.\n" );
321
322     SwingUtilities.invokeLater(
323
324         new Runnable() {
325
326             public void run()
327             {
328                 int row = location / 3;
329                 int column = location % 3;
330
331                 board[ row ][ column ].setMark(
332                     ( myMark == 'O' ? 'X' : 'O' ) );
333                 displayArea.append(
334                     "Opponent moved. Your turn.\n" );
335             }
336         }
337     );
338
339     myTurn = false;
340
341     displayArea.append( "Opponent moved. Your turn.\n" );
342
343     SwingUtilities.invokeLater(
344
345         new Runnable() {
346
347             public void run()
348             {
349                 int row = location / 3;
350                 int column = location % 3;
351
352                 board[ row ][ column ].setMark(
353                     ( myMark == 'X' ? 'O' : 'X' ) );
354                 displayArea.append(
355                     "Opponent moved. Your turn.\n" );
356             }
357         }
358     );
359
360     myTurn = true;
361
362     displayArea.append( "Opponent moved. Your turn.\n" );
363
364     SwingUtilities.invokeLater(
365
366         new Runnable() {
367
368             public void run()
369             {
370                 int row = location / 3;
371                 int column = location % 3;
372
373                 board[ row ][ column ].setMark(
374                     ( myMark == 'O' ? 'X' : 'O' ) );
375                 displayArea.append(
376                     "Opponent moved. Your turn.\n" );
377             }
378         }
379     );
380
381     myTurn = false;
382
383     displayArea.append( "Opponent moved. Your turn.\n" );
384
385     SwingUtilities.invokeLater(
386
387         new Runnable() {
388
389             public void run()
390             {
391                 int row = location / 3;
392                 int column = location % 3;
393
394                 board[ row ][ column ].setMark(
395                     ( myMark == 'X' ? 'O' : 'X' ) );
396                 displayArea.append(
397                     "Opponent moved. Your turn.\n" );
398             }
399         }
400     );
401
402     myTurn = true;
403
404     displayArea.append( "Opponent moved. Your turn.\n" );
405
406     SwingUtilities.invokeLater(
407
408         new Runnable() {
409
410             public void run()
411             {
412                 int row = location / 3;
413                 int column = location % 3;
414
415                 board[ row ][ column ].setMark(
416                     ( myMark == 'O' ? 'X' : 'O' ) );
417                 displayArea.append(
418                     "Opponent moved. Your turn.\n" );
419             }
420         }
421     );
422
423     myTurn = false;
424
425     displayArea.append( "Opponent moved. Your turn.\n" );
426
427     SwingUtilities.invokeLater(
428
429         new Runnable() {
430
431             public void run()
432             {
433                 int row = location / 3;
434                 int column = location % 3;
435
436                 board[ row ][ column ].setMark(
437                     ( myMark == 'X' ? 'O' : 'X' ) );
438                 displayArea.append(
439                     "Opponent moved. Your turn.\n" );
440             }
441         }
442     );
443
444     myTurn = true;
445
446     displayArea.append( "Opponent moved. Your turn.\n" );
447
448     SwingUtilities.invokeLater(
449
450         new Runnable() {
451
452             public void run()
453             {
454                 int row = location / 3;
455                 int column = location % 3;
456
457                 board[ row ][ column ].setMark(
458                     ( myMark == 'O' ? 'X' : 'O' ) );
459                 displayArea.append(
460                     "Opponent moved. Your turn.\n" );
461             }
462         }
463     );
464
465     myTurn = false;
466
467     displayArea.append( "Opponent moved. Your turn.\n" );
468
469     SwingUtilities.invokeLater(
470
471         new Runnable() {
472
473             public void run()
474             {
475                 int row = location / 3;
476                 int column = location % 3;
477
478                 board[ row ][ column ].setMark(
479                     ( myMark == 'X' ? 'O' : 'X' ) );
480                 displayArea.append(
481                     "Opponent moved. Your turn.\n" );
482             }
483         }
484     );
485
486     myTurn = true;
487
488     displayArea.append( "Opponent moved. Your turn.\n" );
489
490     SwingUtilities.invokeLater(
491
492         new Runnable() {
493
494             public void run()
495             {
496                 int row = location / 3;
497                 int column = location % 3;
498
499                 board[ row ][ column ].setMark(
500                     ( myMark == 'O' ? 'X' : 'O' ) );
501                 displayArea.append(
502                     "Opponent moved. Your turn.\n" );
503             }
504         }
505     );
506
507     myTurn = false;
508
509     displayArea.append( "Opponent moved. Your turn.\n" );
510
511     SwingUtilities.invokeLater(
512
513         new Runnable() {
514
515             public void run()
516             {
517                 int row = location / 3;
518                 int column = location % 3;
519
520                 board[ row ][ column ].setMark(
521                     ( myMark == 'X' ? 'O' : 'X' ) );
522                 displayArea.append(
523                     "Opponent moved. Your turn.\n" );
524             }
525         }
526     );
527
528     myTurn = true;
529
530     displayArea.append( "Opponent moved. Your turn.\n" );
531
532     SwingUtilities.invokeLater(
533
534         new Runnable() {
535
536             public void run()
537             {
538                 int row = location / 3;
539                 int column = location % 3;
540
541                 board[ row ][ column ].setMark(
542                     ( myMark == 'O' ? 'X' : 'O' ) );
543                 displayArea.append(
544                     "Opponent moved. Your turn.\n" );
545             }
546         }
547     );
548
549     myTurn = false;
550
551     displayArea.append( "Opponent moved. Your turn.\n" );
552
553     SwingUtilities.invokeLater(
554
555         new Runnable() {
556
557             public void run()
558             {
559                 int row = location / 3;
560                 int column = location % 3;
561
562                 board[ row ][ column ].setMark(
563                     ( myMark == 'X' ? 'O' : 'X' ) );
564                 displayArea.append(
565                     "Opponent moved. Your turn.\n" );
566             }
567         }
568     );
569
570     myTurn = true;
571
572     displayArea.append( "Opponent moved. Your turn.\n" );
573
574     SwingUtilities.invokeLater(
575
576         new Runnable() {
577
578             public void run()
579             {
580                 int row = location / 3;
581                 int column = location % 3;
582
583                 board[ row ][ column ].setMark(
584                     ( myMark == 'O' ? 'X' : 'O' ) );
585                 displayArea.append(
586                     "Opponent moved. Your turn.\n" );
587             }
588         }
589     );
590
591     myTurn = false;
592
593     displayArea.append( "Opponent moved. Your turn.\n" );
594
595     SwingUtilities.invokeLater(
596
597         new Runnable() {
598
599             public void run()
600             {
601                 int row = location / 3;
602                 int column = location % 3;
603
604                 board[ row ][ column ].setMark(
605                     ( myMark == 'X' ? 'O' : 'X' ) );
606                 displayArea.append(
607                     "Opponent moved. Your turn.\n" );
608             }
609         }
610     );
611
612     myTurn = true;
613
614     displayArea.append( "Opponent moved. Your turn.\n" );
615
616     SwingUtilities.invokeLater(
617
618         new Runnable() {
619
620             public void run()
621             {
622                 int row = location / 3;
623                 int column = location % 3;
624
625                 board[ row ][ column ].setMark(
626                     ( myMark == 'O' ? 'X' : 'O' ) );
627                 displayArea.append(
628                     "Opponent moved. Your turn.\n" );
629             }
630         }
631     );
632
633     myTurn = false;
634
635     displayArea.append( "Opponent moved. Your turn.\n" );
636
637     SwingUtilities.invokeLater(
638
639         new Runnable() {
640
641             public void run()
642             {
643                 int row = location / 3;
644                 int column = location % 3;
645
646                 board[ row ][ column ].setMark(
647                     ( myMark == 'X' ? 'O' : 'X' ) );
648                 displayArea.append(
649                     "Opponent moved. Your turn.\n" );
650             }
651         }
652     );
653
654     myTurn = true;
655
656     displayArea.append( "Opponent moved. Your turn.\n" );
657
658     SwingUtilities.invokeLater(
659
660         new Runnable() {
661
662             public void run()
663             {
664                 int row = location / 3;
665                 int column = location % 3;
666
667                 board[ row ][ column ].setMark(
668                     ( myMark == 'O' ? 'X' : 'O' ) );
669                 displayArea.append(
670                     "Opponent moved. Your turn.\n" );
671             }
672         }
673     );
674
675     myTurn = false;
676
677     displayArea.append( "Opponent moved. Your turn.\n" );
678
679     SwingUtilities.invokeLater(
680
681         new Runnable() {
682
683             public void run()
684             {
685                 int row = location / 3;
686                 int column = location % 3;
687
688                 board[ row ][ column ].setMark(
689                     ( myMark == 'X' ? 'O' : 'X' ) );
690                 displayArea.append(
691                     "Opponent moved. Your turn.\n" );
692             }
693         }
694     );
695
696     myTurn = true;
697
698     displayArea.append( "Opponent moved. Your turn.\n" );
699
700     SwingUtilities.invokeLater(
701
702         new Runnable() {
703
704             public void run()
705             {
706                 int row = location / 3;
707                 int column = location % 3;
708
709                 board[ row ][ column ].setMark(
710                     ( myMark == 'O' ? 'X' : 'O' ) );
711                 displayArea.append(
712                     "Opponent moved. Your turn.\n" );
713             }
714         }
715     );
716
717     myTurn = false;
718
719     displayArea.append( "Opponent moved. Your turn.\n" );
720
721     SwingUtilities.invokeLater(
722
723         new Runnable() {
724
725             public void run()
726             {
727                 int row = location / 3;
728                 int column = location % 3;
729
730                 board[ row ][ column ].setMark(
731                     ( myMark == 'X' ? 'O' : 'X' ) );
732                 displayArea.append(
733                     "Opponent moved. Your turn.\n" );
734             }
735         }
736     );
737
738     myTurn = true;
739
740     displayArea.append( "Opponent moved. Your turn.\n" );
741
742     SwingUtilities.invokeLater(
743
744         new Runnable() {
745
746             public void run()
747             {
748                 int row = location / 3;
749                 int column = location % 3;
750
751                 board[ row ][ column ].setMark(
752                     ( myMark == 'O' ? 'X' : 'O' ) );
753                 displayArea.append(
754                     "Opponent moved. Your turn.\n" );
755             }
756         }
757     );
758
759     myTurn = false;
760
761     displayArea.append( "Opponent moved. Your turn.\n" );
762
763     SwingUtilities.invokeLater(
764
765         new Runnable() {
766
767             public void run()
768             {
769                 int row = location / 3;
770                 int column = location % 3;
771
772                 board[ row ][ column ].setMark(
773                     ( myMark == 'X' ? 'O' : 'X' ) );
774                 displayArea.append(
775                     "Opponent moved. Your turn.\n" );
776             }
777         }
778     );
779
780     myTurn = true;
781
782     displayArea.append( "Opponent moved. Your turn.\n" );
783
784     SwingUtilities.invokeLater(
785
786         new Runnable() {
787
788             public void run()
789             {
790                 int row = location / 3;
791                 int column = location % 3;
792
793                 board[ row ][ column ].setMark(
794                     ( myMark == 'O' ? 'X' : 'O' ) );
795                 displayArea.append(
796                     "Opponent moved. Your turn.\n" );
797             }
798         }
799     );
800
801     myTurn = false;
802
803     displayArea.append( "Opponent moved. Your turn.\n" );
804
805     SwingUtilities.invokeLater(
806
807         new Runnable() {
808
809             public void run()
810             {
811                 int row = location / 3;
812                 int column = location % 3;
813
814                 board[ row ][ column ].setMark(
815                     ( myMark == 'X' ? 'O' : 'X' ) );
816                 displayArea.append(
817                     "Opponent moved. Your turn.\n" );
818             }
819         }
820     );
821
822     myTurn = true;
823
824     displayArea.append( "Opponent moved. Your turn.\n" );
825
826     SwingUtilities.invokeLater(
827
828         new Runnable() {
829
830             public void run()
831             {
832                 int row = location / 3;
833                 int column = location % 3;
834
835                 board[ row ][ column ].setMark(
836                     ( myMark == 'O' ? 'X' : 'O' ) );
837                 displayArea.append(
838                     "Opponent moved. Your turn.\n" );
839             }
840         }
841     );
842
843     myTurn = false;
844
845     displayArea.append( "Opponent moved. Your turn.\n" );
846
847     SwingUtilities.invokeLater(
848
849         new Runnable() {
850
851             public void run()
852             {
853                 int row = location / 3;
854                 int column = location % 3;
855
856                 board[ row ][ column ].setMark(
857                     ( myMark == 'X' ? 'O' : 'X' ) );
858                 displayArea.append(
859                     "Opponent moved. Your turn.\n" );
860             }
861         }
862     );
863
864     myTurn = true;
865
866     displayArea.append( "Opponent moved. Your turn.\n" );
867
868     SwingUtilities.invokeLater(
869
870         new Runnable() {
871
872             public void run()
873             {
874                 int row = location / 3;
875                 int column = location % 3;
876
877                 board[ row ][ column ].setMark(
878                     ( myMark == 'O' ? 'X' : 'O' ) );
879                 displayArea.append(
880                     "Opponent moved. Your turn.\n" );
881             }
882         }
883     );
884
885     myTurn = false;
886
887     displayArea.append( "Opponent moved. Your turn.\n" );
888
889     SwingUtilities.invokeLater(
890
891         new Runnable() {
892
893             public void run()
894             {
895                 int row = location / 3;
896                 int column = location % 3;
897
898                 board[ row ][ column ].setMark(
899                     ( myMark == 'X' ? 'O' : 'X' ) );
900                 displayArea.append(
901                     "Opponent moved. Your turn.\n" );
902             }
903         }
904     );
905
906     myTurn = true;
907
908     displayArea.append( "Opponent moved. Your turn.\n" );
909
910     SwingUtilities.invokeLater(
911
912         new Runnable() {
913
914             public void run()
915             {
916                 int row = location / 3;
917                 int column = location % 3;
918
919                 board[ row ][ column ].setMark(
920                     ( myMark == 'O' ? 'X' : 'O' ) );
921                 displayArea.append(
922                     "Opponent moved. Your turn.\n" );
923             }
924         }
925     );
926
927     myTurn = false;
928
929     displayArea.append( "Opponent moved. Your turn.\n" );
930
931     SwingUtilities.invokeLater(
932
933         new Runnable() {
934
935             public void run()
936             {
937                 int row = location / 3;
938                 int column = location % 3;
939
940                 board[ row ][ column ].setMark(
941                     ( myMark == 'X' ? 'O' : 'X' ) );
942                 displayArea.append(
943                     "Opponent moved. Your turn.\n" );
944             }
945         }
946     );
947
948     myTurn = true;
949
950     displayArea.append( "Opponent moved. Your turn.\n" );
951
952     SwingUtilities.invokeLater(
953
954         new Runnable() {
955
956             public void run()
957             {
958                 int row = location / 3;
959                 int column = location % 3;
960
961                 board[ row ][ column ].setMark(
962                     ( myMark == 'O' ? 'X' : 'O' ) );
963                 displayArea.append(
964                     "Opponent moved. Your turn.\n" );
965             }
966         }
967     );
968
969     myTurn = false;
970
971     displayArea.append( "Opponent moved. Your turn.\n" );
972
973     SwingUtilities.invokeLater(
974
975         new Runnable() {
976
977             public void run()
978             {
979                 int row = location / 3;
980                 int column = location % 3;
981
982                 board[ row ][ column ].setMark(
983                     ( myMark == 'X' ? 'O' : 'X' ) );
984                 displayArea.append(
985                     "Opponent moved. Your turn.\n" );
986             }
987         }
988     );
989
990     myTurn = true;
991
992     displayArea.append( "Opponent moved. Your turn.\n" );
993
994     SwingUtilities.invokeLater(
995
996         new Runnable() {
997
998             public void run()
999             {
1000                int row = location / 3;
1001                int column = location % 3;
1002
1003                board[ row ][ column ].setMark(
1004                    ( myMark == 'O' ? 'X' : 'O' ) );
1005                displayArea.append(
1006                    "Opponent moved. Your turn.\n" );
1007            }
1008        }
1009    );
1010
1011    myTurn = false;
1012
1013    displayArea.append( "Opponent moved. Your turn.\n" );
1014
1015    SwingUtilities.invokeLater(
1016
1017        new Runnable() {
1018
1019            public void run()
1020            {
1021                int row = location / 3;
1022                int column = location % 3;
1023
1024                board[ row ][ column ].setMark(
1025                    ( myMark == 'X' ? 'O' : 'X' ) );
1026                displayArea.append(
1027                    "Opponent moved. Your turn.\n" );
1028            }
1029        }
1030    );
1031
1032    myTurn = true;
1033
1034    displayArea.append( "Opponent moved. Your turn.\n" );
1035
1036    SwingUtilities.invokeLater(
1037
1038        new Runnable() {
1039
1040            public void run()
1041            {
1042                int row = location / 3;
1043                int column = location % 3;
1044
1045                board[ row ][ column ].setMark(
1046                    ( myMark == 'O' ? 'X' : 'O' ) );
1047                displayArea.append(
1048                    "Opponent moved. Your turn.\n" );
1049            }
1050        }
1051    );
1052
1053    myTurn = false;
1054
1055    displayArea.append( "Opponent moved. Your turn.\n" );
1056
1057    SwingUtilities.invokeLater(
1058
1059        new Runnable() {
1060
1061            public void run()
1062            {
1063                int row = location / 3;
1064                int column = location % 3;
1065
1066                board[ row ][ column ].setMark(
1067                    ( myMark == 'X' ? 'O' : 'X' ) );
1068                displayArea.append(
1069                    "Opponent moved. Your turn.\n" );
1070            }
1071        }
1072    );
1073
1074    myTurn = true;
1075
1076    displayArea.append( "Opponent moved. Your turn.\n" );
1077
1078    SwingUtilities.invokeLater(
1079
1080        new Runnable() {
1081
1082            public void run()
1083            {
1084                int row = location / 3;
1085                int column = location % 3;
1086
1087                board[ row ][ column ].setMark(
1088                    ( myMark == 'O' ? 'X' : 'O' ) );
1089                displayArea.append(
1090                    "Opponent moved. Your turn.\n" );
1091            }
1092        }
1093    );
1094
1095    myTurn = false;
1096
1097    displayArea.append( "Opponent moved. Your turn.\n" );
1098
1099    SwingUtilities.invokeLater(
1100
1101        new Runnable() {
1102
1103            public void run()
1104            {
1105                int row = location / 3;
1106                int column = location % 3;
1107
1108                board[ row ][ column ].setMark(
1109                    ( myMark == 'X' ? 'O' : 'X' ) );
1110                displayArea.append(
1111                    "Opponent moved. Your turn.\n" );
1112            }
1113        }
1114    );
1115
1116    myTurn = true;
1117
1118    displayArea.append( "Opponent moved. Your turn.\n" );
1119
1120    SwingUtilities.invokeLater(
1121
1122        new Runnable() {
1123
1124            public void run()
1125            {
1126                int row = location / 3;
1127                int column = location % 3;
1128
1129                board[ row ][ column ].setMark(
1130                    ( myMark == 'O' ? 'X' : 'O' ) );
1131                displayArea.append(
1132                    "Opponent moved. Your turn.\n" );
1133            }
1134        }
1135    );
1136
1137    myTurn = false;
1138
1139    displayArea.append( "Opponent moved. Your turn.\n" );
1140
1141    SwingUtilities.invokeLater(
1142
1143        new Runnable() {
1144
1145            public void run()
1146            {
1147                int row = location / 3;
1148                int column = location % 3;
1149
1150                board[ row ][ column ].setMark(
1151                    ( myMark == 'X' ? 'O' : 'X' ) );
1152                displayArea.append(
1153                    "Opponent moved. Your turn.\n" );
1154            }
1155        }
1156    );
1157
1158    myTurn = true;
1159
1160    displayArea.append( "Opponent moved. Your turn.\n" );
1161
1162    SwingUtilities.invokeLater(
1163
1164        new Runnable() {
1165
1166            public void run()
1167            {
1168                int row = location / 3;
1169                int column = location % 3;
1170
1171                board[ row ][ column ].setMark(
1172                    ( myMark == 'O' ? 'X' : 'O' ) );
1173                displayArea.append(
1174                    "Opponent moved. Your turn.\n" );
1175            }
1176        }
1177    );
1178
1179    myTurn = false;
1180
1181    displayArea.append( "Opponent moved. Your turn.\n" );
1182
1183    SwingUtilities.invokeLater(
1184
1185        new Runnable() {
1186
1187            public void run()
1188            {
1189                int row = location / 3;
1190                int column = location % 3;
1191
1192                board[ row ][ column ].setMark(
1193                    ( myMark == 'X' ? 'O' : 'X' ) );
1194                displayArea.append(
1195                    "Opponent moved. Your turn.\n" );
1196            }
1197        }
1198    );
1199
1200    myTurn = true;
1201
1202    displayArea.append( "Opponent moved. Your turn.\n" );
1203
1204    SwingUtilities.invokeLater(
1205
1206        new Runnable() {
1207
1208            public void run()
1209            {
1210                int row = location / 3;
1211                int column = location % 3;
1212
1213                board[ row ][ column ].setMark(
1214                    ( myMark == 'O' ? 'X' : 'O' ) );
1215                displayArea.append(
1216                    "Opponent moved. Your turn.\n" );
1217            }
1218        }
1219    );
1220
1221    myTurn = false;
1222
1223    displayArea.append( "Opponent moved. Your turn.\n" );
1224
1225    SwingUtilities.invokeLater(
1226
1227        new Runnable() {
1228
1229            public void run()
1230            {
1231                int row = location / 3;
1232                int column = location % 3;
1233
1234                board[ row ][ column ].setMark(
1235                    ( myMark == 'X' ? 'O' : 'X' ) );
1236                displayArea.append(
1237                    "Opponent moved. Your turn.\n" );
1238            }
1239        }
1240    );
1241
1242    myTurn = true;
1243
1244    displayArea.append( "Opponent moved. Your turn.\n" );
1245
1246    SwingUtilities.invokeLater(
1247
1248        new Runnable() {
1249
1250            public void run()
1251            {
1252                int row = location / 3;
1253                int column = location % 3;
1254
1255                board[ row ][ column ].setMark(
1256                    ( myMark == 'O' ? 'X' : 'O' ) );
1257                displayArea.append(
1258                    "Opponent moved. Your turn.\n" );
1259            }
1260        }
1261    );
1262
1263    myTurn = false;
1264
1265    displayArea.append( "Opponent moved. Your turn.\n" );
1266
1267    SwingUtilities.invokeLater(
1268
1269        new Runnable() {
1270
1271            public void run()
1272            {
1273                int row = location / 3;
1274                int column = location % 3;
1275
1276                board[ row ][ column ].setMark(
1277                    ( myMark == 'X' ? 'O' : 'X' ) );
1278                displayArea.append(
1279                    "Opponent moved. Your turn.\n" );
1280            }
1281        }
1282    );
1283
1284    myTurn = true;
1285
1286    displayArea.append( "Opponent moved. Your turn.\n" );
1287
1288    SwingUtilities.invokeLater(
1289
1290        new Runnable() {
1291
1292            public void run()
1293            {
1294                int row = location / 3;
1295                int column = location % 3;
1296
1297                board[ row ][ column ].setMark(
1298                    ( myMark == 'O' ? 'X' : 'O' ) );
1299                displayArea.append(
1300                    "Opponent moved. Your turn.\n" );
1301            }
1302        }
1303    );
1304
1305    myTurn = false;
1306
1307    displayArea.append( "Opponent moved. Your turn.\n" );
1308
1309    SwingUtilities.invokeLater(
1310
1311        new Runnable() {
1312
1313            public void run()
1314            {
1315                int row = location / 3;
1316                int column = location % 3;
1317
1318                board[ row ][ column ].setMark(
1319                    ( myMark == 'X' ? 'O' : 'X' ) );
1320                displayArea.append(
1321                    "Opponent moved. Your turn.\n" );
1322            }
1323        }
1324    );
1325
1326    myTurn = true;
1327
1328    displayArea.append( "Opponent moved. Your turn.\n" );
1329
1330    SwingUtilities.invokeLater(
1331
1332        new Runnable() {
1333
1334            public void run()
1335            {
1336                int row = location / 3;
1337                int column = location % 3;
1338
1339                board[ row ][ column ].setMark(
1340                    ( myMark == 'O' ? 'X' : 'O' ) );
1341                displayArea.append(
1342                    "Opponent moved. Your turn.\n" );
1343            }
1344        }
1345    );
1346
1347    myTurn = false;
1348
1349    displayArea.append( "Opponent moved. Your turn.\n" );
1350
1351    SwingUtilities.invokeLater(
1352
1353        new Runnable() {
1354
1355            public void run()
1356            {
1357                int row = location / 3;
1358                int column = location % 3;
1359
1360                board[ row ][ column ].setMark(
1361                    ( myMark == 'X' ? 'O' : 'X' ) );
1362                displayArea.append(
1363                    "Opponent moved. Your turn.\n" );
1364            }
1365        }
1366    );
1367
1368    myTurn = true;
1369
1370    displayArea.append( "Opponent moved. Your turn.\n" );
1371
1372    SwingUtilities.invokeLater(
1373
1374        new Runnable() {
1375
1376            public void run()
1377            {
1378                int row = location / 3;
1379                int column = location % 3;
1380
1381                board[ row ][ column ].setMark(
1382                    ( myMark == 'O' ? 'X' : 'O' ) );
1383                displayArea.append(
1384                    "Opponent moved. Your turn.\n" );
1385            }
1386        }
1387    );
1388
1389    myTurn = false;
1390
1391    displayArea.append( "Opponent moved. Your turn.\n" );
1392
1393    SwingUtilities.invokeLater(
1394
1395        new Runnable() {
1396
1397            public void run()
1398            {
1399                int row = location / 3;
1400                int column = location % 3;
1401
1402                board[ row ][ column ].setMark(
1403                    ( myMark == 'X' ? 'O' : 'X' ) );
1404                displayArea.append(
1405                    "Opponent moved. Your turn.\n" );
1406            }
1407        }
1408    );
1409
1410    myTurn = true;
1411
1412    displayArea.append( "Opponent moved. Your turn.\n" );
1413
1414    SwingUtilities.invokeLater(
1415
1416        new Runnable() {
1417
1418            public void run()
1419            {
1420                int row = location / 3;
1421                int column = location % 3;
1422
1423                board[ row ][ column ].setMark(
1424                    ( myMark == 'O' ? 'X' : 'O' ) );
1425                displayArea.append(
1426                    "Opponent moved. Your turn.\n" );
1427            }
1428        }
1429    );
1430
1431    myTurn = false;
1432
1433    displayArea.append( "Opponent moved. Your turn.\n" );
1434
1435    SwingUtilities.invokeLater(
1436
1437        new Runnable() {
1438
1439            public void run()
1440            {
1441                int row = location / 3;
1442                int column = location % 3;
1443
1444                board[ row ][ column ].setMark(
1445                    ( myMark == 'X' ? 'O' : 'X' ) );
1446                displayArea.append(
1447                    "Opponent moved. Your turn.\n" );
1448            }
1449        }
1450    );
1451
1452    myTurn = true;
1453
1454    displayArea.append( "Opponent moved. Your turn.\n" );
1455
1456    SwingUtilities.invokeLater(
1457
1458        new Runnable() {
1459
1460            public void run()
1461            {
1462                int row = location / 3;
1463                int column = location % 3;
1464
1465                board[ row ][ column ].setMark(
1466                    ( myMark == 'O' ? 'X' : 'O' ) );
1467                displayArea.append(
1468                    "Opponent moved. Your turn
```

```

189
190      }
191
192      ); // fim da chamada para invokeLater
193
194      myTurn = true;
195  }
196
197      // processa problemas ocorridos na comunicação com o servidor
198      catch ( IOException ioException ) {
199          ioException.printStackTrace();
200      }
201
202  }
203
204      // simplesmente exibe a mensagem
205      else
206          displayArea.append( message + "\n" );
207
208      displayArea.setCaretPosition(
209          displayArea.getText().length() );
210
211  } // fim do método processMessage
212
213      // envia mensagem para o servidor indicando qual o quadrado em que se clicou
214      public void sendClickedSquare( int location )
215  {
216      if ( myTurn ) {
217
218          // envia a posição para o servidor
219          try {
220              output.writeInt( location );
221              myTurn = false;
222          }
223
224          // processa problemas ocorridos na comunicação com o servidor
225          catch ( IOException ioException ) {
226              ioException.printStackTrace();
227          }
228      }
229  }
230
231      // configura o quadrado atual
232      public void setCurrentSquare( Square square )
233  {
234          currentSquare = square;
235  }
236
237      // classe privada para os quadrados do tabuleiro
238      private class Square extends JPanel {
239          private char mark;
240          private int location;
241
242          public Square( char squareMark, int squareLocation )
243  {
244              mark = squareMark;
245              location = squareLocation;
246
247              addMouseListener(
248

```

Fig. 17.9 O lado do cliente do programa cliente/servidor do jogo-da-velha (parte 5 de 6).

```

249         new MouseAdapter() {
250
251             public void mouseReleased( MouseEvent e )
252             {
253                 setCurrentSquare( Square.this );
254                 sendClickedSquare( getSquareLocation() );
255             }
256
257         } // fim da classe interna anônima
258
259     ); // fim da chamada para addMouseListener
260
261 } // fim do construtor Square
262
263 // devolve tamanho preferido de Square
264 public Dimension getPreferredSize()
265 {
266     return new Dimension( 30, 30 );
267 }
268
269 // devolve o tamanho mínimo de Square
270 public Dimension getMinimumSize()
271 {
272     return getPreferredSize();
273 }
274
275 // configura a marca para o quadrado
276 public void setMark( char newMark )
277 {
278     mark = newMark;
279     repaint();
280 }
281
282 // devolve a posição do quadrado
283 public int getSquareLocation()
284 {
285     return location;
286 }
287
288 // desenha quadrado
289 public void paintComponent( Graphics g )
290 {
291     super.paintComponent( g );
292
293     g.drawRect( 0, 0, 29, 29 );
294     g.drawString( String.valueOf( mark ), 11, 20 );
295 }
296
297 } // fim da classe Square
298
299 } // fim da classe TicTacToeClient

```

Fig. 17.9 O lado do cliente do programa cliente/servidor do jogo-da-velha (parte 6 de 6).

Se a mensagem recebida for **Valid move..**, as linhas 143 a 159 exibem a mensagem **Valid move, please wait.** e chamam o método **setMark** da classe **Square** para colocar a marca do cliente no quadrado atual (aquele no qual o usuário clicou) usando o método **invokeLater** de **SwingUtilities** para assegurar que a GUI seja atualizada na *thread* de despacho de eventos. Se a mensagem recebida for **Invalid move, try again..**, as linhas 162 a 165 exibem a mensagem para o usuário clicar em um quadrado diferente. Se a mensagem recebida for **Opponent moved..**, as linhas 168 a 195 lêem do servidor um inteiro que indica on-

de o oponente jogou e colocam uma marca naquele quadrado do tabuleiro (novamente, usando o método `invokeLater` de `SwingUtilities` para assegurar que a GUI seja atualizada na *thread* de despacho de eventos). Se qualquer outra mensagem for recebida, a linha 206 simplesmente exibe a mensagem.

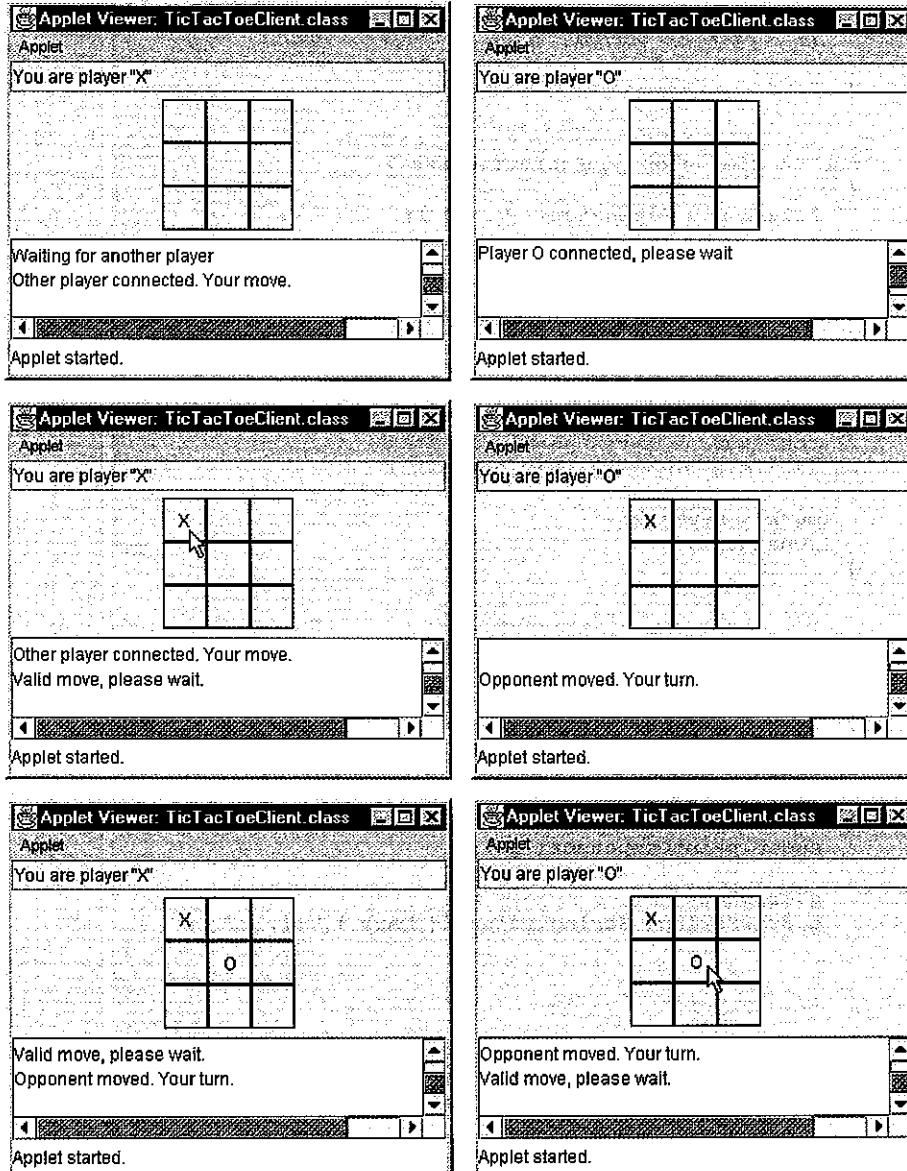


Fig. 17.10 Exemplos de saídas do programa cliente/servidor do jogo-da-velha (parte 1 de 2).

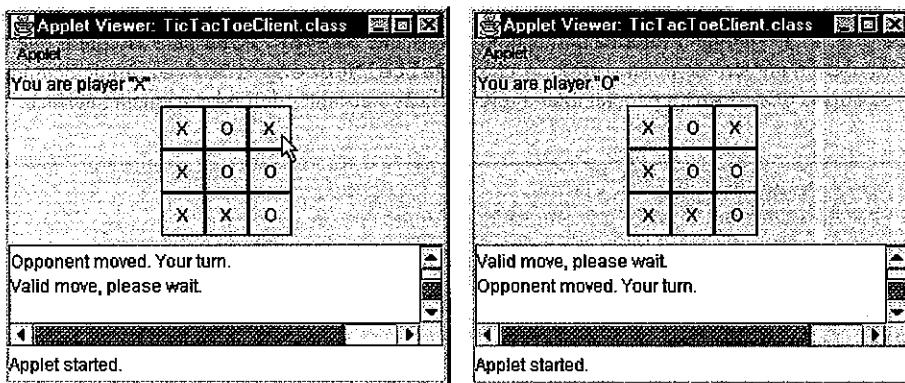


Fig. 17.10 Exemplos de saídas do programa cliente/servidor do jogo-da-velha (parte 2 de 2).

17.9 Segurança e a rede

Por mais ansiosos que estejamos para escrever uma grande variedade de aplicativos poderosos baseados em redes, nossos esforços podem ser obstruídos por limitações impostas a Java devido a preocupações com a segurança. Muitos navegadores da Web, como o Netscape *Communicator* e o Microsoft *Internet Explorer*, proibem, por *default*, *applets* Java de fazer processamento de arquivos nas máquinas em que são executados. Pense no assunto. Projeta-se um *applet* Java para ser enviado para seu navegador através de um documento HTML que pode ser baixado de qualquer servidor da Web no mundo. Com freqüência, você saberá muito pouco sobre as origens dos *applets* Java que serão executados em seu sistema. Permitir a esses *applets* livre acesso aos seus arquivos pode ser desastroso.

Uma situação mais sutil ocorre quando se limita as máquinas com as quais os *applets* em execução podem se conectar. Para construir aplicativos verdadeiramente colaboradores, idealmente gostaríamos de ter nossos *applets* comunicando-se com máquinas quase em qualquer lugar. O gerenciador de segurança de Java em um navegador da Web freqüentemente restringe um *applet* de modo que ele possa se comunicar apenas com a máquina da qual originalmente foi baixado.

Essas restrições podem parecer muito severas. Entretanto, a Java Security API agora fornece recursos para *applets* assinados que permitirão que os navegadores determinem se um *applet* é baixado de uma *fonte confiável*. Nos casos em que um *applet* é confiável, ele pode receber acesso adicional ao computador em que está sendo executado. Os recursos da Java Security API e os recursos adicionais para redes são discutidos em nosso texto *Advanced Java 2 Platform How to Program*.

17.10 Servidor e cliente de bate-papo DeitelMessenger

As salas de bate-papo se tornaram bastante comuns na Internet. Elas oferecem um lugar central em que os usuários podem falar uns com os outros através de mensagens curtas de texto. Cada participante em uma sala de bate-papo pode ver todas as mensagens que os outros usuários enviam e cada usuário pode divulgar mensagens na sala. Esta seção apresenta nosso estudo de caso fundamental para redes que integra muitos dos recursos de Java para redes, *multithreading* e GUI Swing que aprendemos até agora, para construir um sistema de bate-papo *on-line*. Também apresentamos o *multicast*, que permite a um aplicativo enviar *DatagramPackets* para grupos de clientes. Depois de ler esta seção, você será capaz de construir aplicativos importantes para redes.

17.10.1 DeitelMessengerServer e classes de suporte

O *DeitelMessengerServer* (Fig. 17.11) é o coração do sistema de bate-papo *on-line*. Os clientes de bate-papo podem participar de uma conversa se conectando ao *DeitelMessengerServer*. O método *startServer* (linhas 19 a 54) dispara o *DeitelMessengerServer*. As linhas 25 e 26 criam um *ServerSocket* para acei-

tar conexões que chegam através da rede. Lembre-se de que o construtor `ServerSocket` recebe como primeiro argumento a porta na qual o servidor deve esperar que as conexões cheguem. A interface `SocketMessengerConstants` (Fig. 17.12) define o valor da porta como a constante `SERVER_PORT` para assegurar que o servidor e seus clientes usem o número de porta correto. A classe `DeitelMessengerServer` implementa a interface `SocketMessengerConstants` para facilitar a referência às constantes definidas naquela interface.

```

1 // DeitelMessengerServer.java
2 // DeitelMessengerServer é um servidor de bate-papo com
3 // múltiplas threads, baseado em soquetes e pacotes.
4 package com.deitel.messenger.sockets.server;
5
6 // Pacotes do núcleo de Java
7 import java.util.*;
8 import java.net.*;
9 import java.io.*;
10
11 // Pacotes Deitel
12 import com.deitel.messenger.*;
13 import com.deitel.messenger.sockets.*;
14
15 public class DeitelMessengerServer implements MessageListener,
16     SocketMessengerConstants {
17
18     // inicia o servidor de bate-papo
19     public void startServer()
20     {
21         // cria o servidor e administra novos clientes
22         try {
23
24             // cria o ServerSocket para as conexões que chegam
25             ServerSocket serverSocket =
26                 new ServerSocket( SERVER_PORT, 100 );
27
28             System.out.println( "Server listening on port " +
29                 SERVER_PORT + " ..." );
30
31             // espera clientes continuamente
32             while ( true ) {
33
34                 // aceita nova conexão de cliente
35                 Socket clientSocket = serverSocket.accept();
36
37                 // cria nova ReceivingThread para
38                 // receber mensagens do cliente
39                 new ReceivingThread( this, clientSocket ).start();
40
41                 // imprime informações sobre a conexão
42                 System.out.println( "Connection received from: " +
43                     clientSocket.getInetAddress() );
44
45             } // fim do while
46
47         } // fim do try
48
49         // trata de exceções durante a criação do servidor e de conexão de clientes
50         catch ( IOException ioException ) {
51             ioException.printStackTrace();
52         }
53

```

Fig. 17.11 Aplicativo `DeitelMessengerServer` para administrar uma sala de bate-papo (parte 1 de 2).

```

54     } // fim do método startServer
55
56     // quando uma nova mensagem é recebida, envia a mensagem para todos os clientes
57     public void messageReceived( String from, String message )
58     {
59         // cria String contendo a mensagem inteira
60         String completeMessage = from + MESSAGE_SEPARATOR + message;
61
62         // cria e inicia a MulticastSendingThread para
63         // enviar novas mensagens para todos os clientes
64         new MulticastSendingThread(
65             completeMessage.getBytes() ).start();
66     }
67
68     // inicia o servidor
69     public static void main ( String args[] )
70     {
71         new DeitelMessengerServer().startServer();
72     }
73 }

```

```

Server listening on port 5000 ...
Connection received from: SEANSANTRY/xxx.xxx.xxx.xxx
Connection received from: PJD/XXX.XXX.XXX.XXX

```

Fig. 17.11 Aplicativo `DeitelMessengerServer` para administrar uma sala de bate-papo (parte 2 de 2).

As linhas 32 a 45 esperam continuamente novas conexões de clientes. A linha 35 invoca o método `accept` da classe `ServerSocket` para esperar e aceitar uma nova conexão de cliente. A linha 39 cria e inicia uma nova `ReceivingThread` para o cliente. A classe `ReceivingThread` (Fig. 17.4) é uma subclasse de `Thread` que espera novas mensagens que chegam de um cliente particular. O primeiro argumento para o construtor `ReceivingThread` é um `MessageListener` (Fig. 17.13), para o qual as mensagens do cliente devem ser entregues. A classe `DeitelMessengerServer` implementa a interface `MessageListener` (linha 15) e, portanto, pode passar a referência `this` para o construtor `ReceivingThread`.

O método `messageReceived` (linhas 57 a 66) é exigido pela interface `MessageListener`. Quando cada `ReceivingThread` recebe uma nova mensagem de um cliente, a `ReceivingThread` passa a mensagem para um `MessageListener` através do método `messageReceived`. A linha 60 concatena o `String from` com o separador `>>>` e o corpo da mensagem. As linhas 64 e 65 criam e iniciam (com `start`) uma nova `MulticastSendingThread` para enviar `completeMessage` para todos os clientes que estão esperando. A classe `MulticastSendingThread` (Fig. 17.15) utiliza *multicast* como um mecanismo eficiente para enviar uma mensagem para múltiplos clientes. Discutimos os detalhes de *multicasting* em seguida. O método `main` (linhas 69 a 72) cria uma nova instância de `DeitelMessengerServer` e inicia o servidor.

A interface `SocketMessengerConstants` (Fig. 17.12) declara constantes para uso nas várias classes que formam o sistema de mensagens Deitel. As classes podem acessar estas constantes `static` quer fazendo referência às constantes através da interface `SocketMessengerConstants` (por exemplo, `SocketMessengerConstants.SERVER_PORT`) ou implementando a interface e fazendo referência às constantes diretamente.

A linha 9 define a constante `String MULTICAST_ADDRESS`, que contém o endereço para o qual uma `MulticastSendingThread` (Fig. 17.15) deve enviar mensagens. Este endereço é um dos endereços reservados para *multicast*, que descreveremos em breve. A linha 12 define a constante inteira `MULTICAST_LISTENING_PORT` – a porta através da qual os clientes devem esperar novas mensagens. A linha 15 define a constante inteira `MULTICAST_SENDING_PORT` – a porta para a qual uma `MulticastSendingThread` deve enviar novas mensagens no `MULTICAST_ADDRESS`. A linha 18 define a constante inteira `SERVER_PORT` – a porta na qual `DeitelMessengerServer` espera as conexões que chegam de clientes. A linha 21 define a constante `String DISCONNECT_STRING`, que é o `String` que um cliente envia para o `DeitelMessengerServer` quando o usuário de-

seja sair da sala de bate-papo. A linha 24 define a constante **String MESSAGE_SEPARATOR**, que separa o nome do usuário do corpo da mensagem. A linha 27 especifica o tamanho máximo da mensagem, em *bytes*.

```

1 // SocketMessengerConstants.java
2 // SocketMessengerConstants define constantes para os números
3 // de portas e endereços de multicast em DeitelMessenger
4 package com.deitel.messenger.sockets;
5
6 public interface SocketMessengerConstants {
7
8     // endereço para datagramas de multicast
9     public static final String MULTICAST_ADDRESS = "230.0.0.1";
10
11    // porta para esperar por datagramas de multicast datagrams
12    public static final int MULTICAST_LISTENING_PORT = 5555;
13
14    // porta para enviar datagramas de multicast
15    public static final int MULTICAST_SENDING_PORT = 5554;
16
17    // porta para conexões com soquetes ao DeitelMessengerServer
18    public static final int SERVER_PORT = 5000;
19
20    // String que indica desconectar
21    public static final String DISCONNECT_STRING = "DISCONNECT";
22
23    // String que separa o nome do usuário do corpo da mensagem
24    public static final String MESSAGE_SEPARATOR = ">>>";
25
26    // tamanho da mensagem (em bytes)
27    public static final int MESSAGE_SIZE = 512;
28 }
```

Fig. 17.12 `SocketMessengerConstants` declara constantes para uso ao longo de todo o `DeitelMessengerServer` e o `DeitelMessenger`

Muitas classes diferentes no sistema de mensagens Deitel recebem mensagens. Por exemplo, `DeitelMessengerServer` recebe mensagens de clientes e envia estas mensagens para todos os participantes da sala de bate-papo. Como veremos, a interface com o usuário para cada cliente também recebe mensagens e exibe estas mensagens para os usuários. Cada uma das classes que recebem mensagens implementa a interface `MessageListener` (Fig. 17.13). A interface declara o método `messageReceived`, que permite que uma classe que a implementa receba mensagens de bate-papo. O método `messageReceived` recebe dois argumentos `String` que representam o nome do usuário que enviou a mensagem e o corpo da mensagem, respectivamente.

O `DeitelMessengerServer` utiliza instâncias da classe `ReceivingThread` (Fig. 17.14) para esperar novas mensagens de cada cliente. A classe `ReceivingThread` estende a classe `Thread`. Isto permite que o `DeitelMessengerServer` crie um objeto da classe `ReceivingThread` para cada cliente, para tratar mensagens de múltiplos clientes de uma só vez. Quando o `DeitelMessengerServer` recebe uma nova conexão de cliente, o `DeitelMessengerServer` cria uma nova `ReceivingThread` para o cliente e depois continua esperando novas conexões de clientes. A `ReceivingThread` espera novas mensagens de um único cliente e passa aquelas mensagens de volta através do método `messageReceived`.

```

1 // MessageListener.java
2 // MessageListener é uma interface para classes que
3 // querem receber novas mensagens de bate-papo.
4 package com.deitel.messenger;
```

Fig. 17.13 Interface `MessageListener` que define o método `messageReceived` para receber novas mensagens de bate-papo (parte 1 de 2).

```

5   public interface MessageListener {
6
7     // recebe nova mensagem de bate-papo
8     public void messageReceived( String from, String message );
9
10 }

```

Fig. 17.13 Interface `MessageListener` que define o método `messageReceived` para receber novas mensagens de bate-papo (parte 2 de 2).

```

1 // ReceivingThread.java
2 // ReceivingThread é uma Thread que espera mensagens
3 // de um cliente particular e envia mensagens para um
4 // MessageListener.
5 package com.deitel.messenger.sockets.server;
6
7 // Pacotes do núcleo de Java
8 import java.io.*;
9 import java.net.*;
10 import java.util.StringTokenizer;
11
12 // Pacotes Deitel
13 import com.deitel.messenger.*;
14 import com.deitel.messenger.sockets.*;
15
16 public class ReceivingThread extends Thread implements
17     SocketMessengerConstants {
18
19     private BufferedReader input;
20     private MessageListener messageListener;
21     private boolean keepListening = true;
22
23     // construtor de ReceivingThread
24     public ReceivingThread( MessageListener listener,
25         Socket clientSocket )
26     {
27         // invoca o construtor da superclasse para dar nome à Thread
28         super( "ReceivingThread: " + clientSocket );
29
30         // configura o ouvinte para o qual novas mensagens devem ser enviadas
31         messageListener = listener;
32
33         // configura tempo máximo de espera para leitura do clientSocket
34         // e cria um BufferedReader para ler mensagens que chegam
35         try {
36             clientSocket.setSoTimeout( 5000 );
37
38             input = new BufferedReader( new InputStreamReader(
39                 clientSocket.getInputStream() ) );
40         }
41
42         // trata de exceções durante a criação do BufferedReader
43         catch ( IOException ioException ) {
44             ioException.printStackTrace();
45         }
46

```

Fig. 17.14 `ReceivingThread` para esperar novas mensagens de clientes do `DeitelMessengerServer` em Threads separadas (parte 1 de 3).

```

47     } // fim do construtor de ReceivingThread
48
49     // espera novas mensagens e as envia para o MessageListener
50     public void run()
51     {
52         String message;
53
54         // espera novas mensagens até que seja parado
55         while ( keepListening ) {
56
57             // lê mensagem do BufferedReader
58             try {
59                 message = input.readLine();
60             }
61
62             // trata de exceção quando o tempo máximo de leitura é esgotado
63             catch ( InterruptedException interruptedIOException ) {
64
65                 // continua para a próxima iteração para se manter na espera
66                 continue;
67             }
68
69             // trata de exceções durante a leitura de mensagem
70             catch ( IOException ioException ) {
71                 ioException.printStackTrace();
72                 break;
73             }
74
75             // assegura que a mensagem não esteja vazia
76             if ( message != null ) {
77
78                 // separa a mensagens em tokens para recuperar
79                 // o nome do usuário e o corpo da mensagem
80                 StringTokenizer tokenizer =
81                     new StringTokenizer( message, MESSAGE_SEPARATOR );
82
83                 // ignora mensagens que não contêm um nome
84                 // de usuário e um corpo de mensagem
85                 if ( tokenizer.countTokens() == 2 ) {
86
87                     // envia mensagem para o MessageListener
88                     messageListener.messageReceived(
89                         tokenizer.nextToken(),           // nome do usuário
90                         tokenizer.nextToken() );      // corpo da mensagem
91                 }
92
93                 else
94
95                     // se recebeu mensagem de desconexão, pára de esperar mensagens
96                     if ( message.equalsIgnoreCase( MESSAGE_SEPARATOR +
97                         DISCONNECT_STRING ) ) {
98
99                         stopListening();
100                    }
101
102                } // fim do if
103
104            } // fim do while

```

Fig. 17.14 ReceivingThread para esperar novas mensagens de clientes do DeitelMessenger-Server em Threads separadas (parte 2 de 3).

```

105
106    // fecha o BufferedReader (também fecha o Socket)
107    try {
108        input.close();
109    }
110
111    // trata de exceção durante o fechamento do BufferedReader
112    catch ( IOException ioException ) {
113        ioException.printStackTrace();
114    }
115
116 } // fim do método run
117
118 // pára de esperar mensagens que chegam
119 public void stopListening()
120 {
121     keepListening = false;
122 }
123 }
```

Fig. 17.14 ReceivingThread para esperar novas mensagens de clientes do DeitelMessenger-Server em Threads separadas (parte 3 de 3).

O construtor **ReceivingThread** (linhas 24 a 47) recebe como primeiro argumento um **MessageListener**. A **ReceivingThread** irá entregar mensagens para este **MessageListener** invocando o método **messageReceived** da interface **MessageListener**. O argumento **Socket** para o construtor **ReceivingThread** é a conexão para um cliente em particular. A linha 28 invoca o construtor **Thread** para fornecer um único nome para cada instância de **ReceivingThread**. Dar nomes às **ReceivingThreads** desta maneira pode ser útil durante a depuração do aplicativo. A linha 31 configura o **MessageListener** para o qual a **ReceivingThread** deve entregar novas mensagens. A linha 36 invoca o método **setSoTimeout** da classe **Socket** com um argumento inteiro de 5000 milissegundos. A leitura de dados de um soquete é uma *chamada bloqueante* – a *thread* atual é colocada no estado bloqueada (Fig. 15.1) enquanto a *thread* espera que a operação de leitura seja completada. O método **setSoTimeout** especifica que, se nenhum dado for recebido decorrido o número de milissegundos especificado, o **Socket** deve disparar uma **InterruptedException**, a qual pode ser capturada pela *thread* atual, e então continuar a ser executada. Esta técnica evita que a *thread* atual fique em *deadlock* se mais nenhum dado estiver disponível a partir do **Socket**. As linhas 38 e 39 criam um novo **BufferedReader** para o **InputStream** do **clientSocket**. A **ReceivingThread** usa este **BufferedReader** para ler novas mensagens do cliente.

O método **run** (linhas 50 a 116) espera continuamente novas mensagens do cliente. As linhas 55 a 104 repetem um laço enquanto a variável boolean **keepListening** for igual a **true**. A linha 59 invoca o método **readLine** de **BufferedReader** para ler uma linha de dados do cliente. Se mais de 5000 milissegundos se passam sem a leitura de nenhum dado, o método **readLine** dispara uma **InterruptedException**, o que indica que o tempo máximo configurado na linha 36 se esgotou. A linha 66 usa a palavra-chave **continue** para passar para a próxima iteração do laço **while** para continuar a esperar por mensagens. As linhas 70 a 73 capturam uma **IOException**, que indica um problema mais sério no método **readLine**. A linha 71 imprime um monitoramento da pilha para ajudar na depuração do aplicativo, e a linha 72 usa a palavra-chave **break** para terminar o laço **while**.

Quando um cliente envia uma mensagem para o servidor, ele separa o nome do usuário do corpo da mensagem com o string **MESSAGE_SEPARATOR**. Se nenhuma exceção é disparada durante a leitura dos dados do cliente e a mensagem não for **null** (linha 76), as linhas 80 e 81 criam um novo **StringTokenizer**. Este **StringTokenizer** separa cada mensagem em duas *tokens* delimitadas por **MESSAGE_SEPARATOR**. A primeira *token* é o nome do usuário que enviou; a segunda *token* é a mensagem. A linha 85 verifica se a quantidade de *tokens* está correta e as linhas 88 a 90 invocam o método **messageReceived** da interface **MessageListener** para entregar a nova mensagem para o **MessageListener** registrado. Se o **StringTokenizer** não produz duas *tokens*, as linhas 96 e 97 verificam a mensagem para ver se ela coincide com a constante **DISCONNECT_STRING**, o que indicaria que o usuário deseja sair da sala de bate-papo. Se os **Strings** coincidirem, a linha 99 invoca o método **stopListening** de **ReceivingThread** para terminar a **ReceivingThread**.

O método `stopListening` (linhas 119 a 122) configura a variável booleana `keepListening` para `false`. Isto faz com que a condição do laço `while` na linha 55 falhe e faz a `ReceivingThread` fechar o `Socket` cliente (linha 108). então, o método `run` retorna, o que termina a execução da `ReceivingThread`.

A `MulticastSendingThread` (Fig. 17.15) entrega `DatagramPackets` que contêm mensagens de bate-papo para um grupo de clientes. O *multicast* é uma maneira eficiente de enviar dados para muitos clientes sem a sobrecarga de enviar aqueles dados para todos os *hosts* na Internet. Para entender o *multicast*, vejamos uma analogia do mundo real – o relacionamento entre o editor de uma revista e os assinantes dela. O editor da revista produz uma revista e a oferece para o distribuidor. Os clientes interessados naquela revista obtêm uma assinatura e começam a receber a revista pelo correio a partir do distribuidor. Esta comunicação é bastante diferente de uma transmissão de televisão. Quando uma emissora de televisão produz um programa, a estação transmite o programa para toda uma área geográfica ou até mesmo para todo o mundo, através de satélites. Transmitir um programa de televisão para 10.000 espectadores não é mais caro para a emissora de televisão do que transmitir o programa para 100.000 espectadores – o sinal de rádio que carrega a transmissão atinge uma área ampla. Entretanto, imprimir e entregar uma revista para 10.000 leitores seria muito mais caro do que imprimir e entregar a revista para 100 leitores. A maioria das editoras de revistas não conseguiria continuar no ramo se tivessem de enviar suas revistas para todo o mundo; assim, as editoras fazem *multicast* de sua revista para um grupo de assinantes.

Usando *multicast*, um aplicativo pode “publicar” `DatagramPackets` para serem entregues para outros aplicativos – os “assinantes”. Um aplicativo faz *multicast* dos `DatagramPackets` enviando os `DatagramPackets` para um *endereço de multicast*, que é um endereço IP reservado para *multicast*, no intervalo de 224.0.0.0 a 239.255.255.255. Os clientes que desejam receber estes `DatagramPackets` podem se conectar ao endereço de *multicast* apropriado para se juntar ao grupo de assinantes – o *grupo de multicast*. Quando um aplicativo envia um `DatagramPacket` para o endereço de *multicast*, cada cliente no grupo de *multicast* recebe o `DatagramPacket`. Os `DatagramPackets` de *multicast*, assim como os `DatagramPackets` de *unicast* (Fig. 17.7), não são confiáveis – não há garantia de que os pacotes cheguem ao destino. Além disso, não é garantida a ordem na qual os clientes recebem os datagramas.

A classe `MulticastSendingThread` estende a classe `Thread` para permitir que o `DeitelMessengerServer` envie mensagens de *multicast* através de uma *thread* separada. Cada vez que o `DeitelMessengerServer` precisa fazer *multicast* de uma mensagem, o servidor cria uma nova `MulticastSendingThread` com o conteúdo da mensagem e inicia a *thread*. O construtor `MulticastSendingThread` (linhas 20 a 26) recebe como argumento um *array* de `bytes` que contém a mensagem.

```

1 // MulticastSendingThread.java
2 // MulticastSendingThread é uma Thread que distribui uma
3 // mensagem de bate-papo com um datagrama de multicast.
4 package com.deitel.messenger.sockets.server;
5
6 // Pacotes do núcleo de Java
7 import java.io.*;
8 import java.net.*;
9
10 // Pacotes Deitel
11 import com.deitel.messenger.sockets.*;
12
13 public class MulticastSendingThread extends Thread
14     implements SocketMessengerConstants {
15
16     // dados da mensagem
17     private byte[] messageBytes;
18
19     // construtor MulticastSendingThread
20     public MulticastSendingThread( byte[] bytes )
21     {

```

Fig. 17.15 `MulticastSendingThread` para entregar mensagens enviadas para um grupo de *multicast* através de `DatagramPackets` (parte 1 de 2).

```

22     // invoca o construtor da superclasse para dar nome à Thread
23     super( "MulticastSendingThread" );
24
25     messageBytes = bytes;
26 }
27
28 // entrega a mensagem para o MULTICAST_ADDRESS através do DatagramSocket
29 public void run()
30 {
31     // entrega a mensagem
32     try {
33
34         // cria o DatagramSocket para enviar a mensagem
35         DatagramSocket socket =
36             new DatagramSocket( MULTICAST_SENDING_PORT );
37
38         // usa o InetAddress reservado para grupo de multicast
39         InetAddress group = InetAddress.getByName(
40             MULTICAST_ADDRESS );
41
42         // cria o DatagramPacket que contém a mensagem
43         DatagramPacket packet = new DatagramPacket(
44             messageBytes, messageBytes.length, group,
45             MULTICAST_LISTENING_PORT );
46
47         // envia o pacote para um grupo de multicast e fecha o soquete
48         socket.send( packet );
49         socket.close();
50     }
51
52     // trata de exceções durante a entrega da mensagem
53     catch ( IOException ioException ) {
54         ioException.printStackTrace();
55     }
56
57 } // fim do método run
58 }
```

Fig. 17.15 MulticastSendingThread para entregar mensagens enviadas para um grupo de *multicast* através de DatagramPackets (parte 2 de 2).

O método `run` (linhas 29 a 57) entrega a mensagem para o endereço de *multicast*. As linhas 35 e 36 criam um novo `DatagramSocket`. Lembre-se de que, no exemplo de rede de pacotes, usamos `DatagramSockets` para enviar `DatagramPackets` de *unicast* – pacotes enviados de um *host* diretamente para outro *host*. Entregar `DatagramPackets` com *multicast* é exatamente igual, exceto pelo fato de que o endereço para o qual os `DatagramPackets` são enviados é um endereço no intervalo de 224.0.0.0 a 239.255.255.255. As linhas 39 e 40 criam um objeto `InetAddress` para o endereço de *multicast*, que é definido como uma constante na interface `SocketMessengerConstants`. As linhas 43 a 45 criam o `DatagramPacket` que contém a mensagem. O primeiro argumento para o construtor `DatagramPacket` é o *array* de `bytes` que contém a mensagem. O segundo argumento é o comprimento do *array* de `bytes`. O terceiro argumento especifica o `InetAddress` para o qual o pacote deve ser enviado e o último argumento especifica o número da porta através da qual o pacote deve ser entregue para o endereço de *multicast*. A linha 48 invoca o método `send` da classe `DatagramSocket` para enviar o `DatagramPacket`. Quando o `DatagramPacket` é enviado para o endereço de *multicast*, todos os clientes que esperam mensagens daquele endereço de *multicast* na porta apropriada recebem o `DatagramPacket`. A linha 49 fecha o `DatagramSocket` e o método `run` retorna, terminando a `MulticastSendingThread`.

17.10.2 Cliente DeitelMessenger e classes de suporte

O cliente para o **DeitelMessengerServer** consiste em diversas partes. A classe que implementa a interface **MessageManager** (Fig. 17.16) administra a comunicação com o servidor. A subclasse de **Thread** espera mensagens no endereço de *multicast* do **DeitelMessengerServer**. Outra subclasse de **Thread** envia mensagens do cliente para o **DeitelMessengerServer**. A subclasse de **JFrame** fornece uma GUI para o cliente.

A interface **MessageManager** (Fig. 17.16) define métodos para administrar a comunicação com o **DeitelMessengerServer**. Definimos esta interface para abstrair a funcionalidade básica que um cliente precisa para interagir com um servidor de bate-papo do mecanismo de comunicação subjacente necessário para se comunicar com aquele servidor de bate-papo. Esta abstração nos permite fornecer implementações de **MessageManager** que usam outros protocolos de rede para implementar os detalhes de comunicação. Por exemplo, se queremos nos conectar a um servidor de bate-papo diferente, que não usa **DatagramPackets** de *multicast*, podemos implementar a interface **MessageManager** com os protocolos de rede apropriados para este servidor de mensagens alternativo. Não precisaríamos modificar nenhum outro código no cliente, porque os outros componentes do cliente se referem somente à interface **MessageManager** e não a alguma implementação particular de **MessageManager**. Do mesmo modo, os métodos da interface **MessageManager** fazem referência a outros componentes do cliente somente através da interface **MessageListener**. Portanto, outros componentes do cliente podem mudar sem necessitar de mudanças no **MessageManager** ou em suas implementações. O método **connect** (linha 10) conecta o **MessageManager** ao **DeitelMessengerServer** e roteia as mensagens que chegam para o **MessageListener** apropriado. O método **disconnect** (linha 14) desconecta o **MessageManager** do **DeitelMessengerServer** e pára de entregar mensagens para o **MessageListener** especificado. O método **sendMessage** (linha 17) envia uma nova mensagem para o **DeitelMessengerServer**.

A classe **SocketMessageManager** (Fig. 17.17) implementa a interface **MessageManager** (linha 16), usando **Sockets** e **MulticastSockets** para se comunicar com o **DeitelMessengerServer** e receber as mensagens que chegam. A linha 20 declara o **Socket** que **SocketMessageManager** utiliza para se conectar e enviar mensagens para o **DeitelMessengerServer**. A linha 26 declara a **PacketReceivingThread** (Fig. 17.19) que espera as novas mensagens que chegam. O indicador **boolean connected** (linha 29) indica se o **SocketMessageManager** está conectado ao **DeitelMessengerServer**.

```

1 // MessageManager.java
2 // MessageManager é uma interface para objetos capaz de
3 // administrar as comunicações com um servidor de mensagens.
4 package com.deitel.messenger;
5
6 public interface MessageManager {
7
8     // conecta-se ao servidor de mensagens e roteia as mensagens
9     // que chegam para um MessageListener determinado
10    public void connect( MessageListener listener );
11
12    // desconecta-se do servidor de mensagens e pára de rotear
13    // as mensagens que chegam para o MessageListener dado
14    public void disconnect( MessageListener listener );
15
16    // envia mensagem para o servidor de mensagens
17    public void sendMessage( String from, String message );
18 }

```

Fig. 17.16 Interface **MessageManager** que define métodos para se comunicar com um **DeitelMessengerServer**.

```

1 // SocketMessageManager.java
2 // SocketMessageManager é uma implementação de MessageManager
3 // para comunicação com um DeitelMessengerServer com Sockets
4 // e MulticastSockets.
5 package com.deitel.messenger.sockets.client;
6
7 // Pacotes do núcleo de Java
8 import java.util.*;
9 import java.net.*;
10 import java.io.*;
11
12 // Pacotes Deitel
13 import com.deitel.messenger.*;
14 import com.deitel.messenger.sockets.*;
15
16 public class SocketMessageManager implements MessageManager,
17     SocketMessengerConstants {
18
19     // Socket para mensagens enviadas
20     private Socket clientSocket;
21
22     // endereço do DeitelMessengerServer
23     private String serverAddress;
24
25     // Thread para receber mensagens de multicast
26     private PacketReceivingThread receivingThread;
27
28     // indicador que indica o estado da conexão
29     private boolean connected = false;
30
31     // construtor SocketMessageManager
32     public SocketMessageManager( String address )
33     {
34         serverAddress = address;
35     }
36
37     // conecta-se ao servidor e envia mensagens para o MessageListener especificado
38     public void connect( MessageListener listener )
39     {
40         // se já está conectado, retorna imediatamente
41         if ( connected )
42             return;
43
44         // abre conexão ao DeitelMessengerServer com Socket
45         try {
46             clientSocket = new Socket(
47                 InetAddress.getByName( serverAddress ), SERVER_PORT );
48
49             // cria Thread para receber mensagens que chegam
50             receivingThread = new PacketReceivingThread( listener );
51             receivingThread.start();
52
53             // atualiza indicador de conectado
54             connected = true;
55
56         } // fim do try
57
58         // trata exceção ocorrida durante a conexão com o servidor

```

Fig. 17.17 Implementação da interface MessageManager em SocketMessageManager para comunicação através de Sockets e DatagramPackets de multicast (parte 1 de 2).

```

59         catch ( IOException ioException ) {
60             ioException.printStackTrace();
61         }
62     } // fim do método connect
63
64     // desconecta-se do servidor e cancela registro do MessageListener especificado
65     public void disconnect( MessageListener listener )
66     {
67         // se não está conectado, retorna imediatamente
68         if ( !connected )
69             return;
70
71         // faz parar a thread que espera mensagens e desconecta-se do servidor
72         try {
73
74             // notifica o servidor de que o cliente está se desconectando
75             Thread disconnectThread = new SendingThread(
76                 clientSocket, "", DISCONNECT_STRING );
77             disconnectThread.start();
78
79             // espera 10 segundos para a mensagem de desconexão ser enviada
80             disconnectThread.join( 10000 );
81
82             // faz parar a receivingThread e remove o MessageListener especificado
83             receivingThread.stopListening();
84
85             // fecha o Socket de saída
86             clientSocket.close();
87
88         } // fim do try
89
90         // trata exceção ocorrida durante desconexão do servidor
91         catch ( IOException ioException ) {
92             ioException.printStackTrace();
93         }
94
95         // trata exceção ocorrida durante junção com a disconnectThread
96         catch ( InterruptedException interruptedException ) {
97             interruptedException.printStackTrace();
98         }
99
100        // atualiza indicador de conectado
101        connected = false;
102
103    } // fim do método disconnect
104
105    // envia mensagem para o servidor
106    public void sendMessage( String from, String message )
107    {
108        // se não está conectado, retorna imediatamente
109        if ( !connected )
110            return;
111
112        // cria e inicia uma nova SendingThread para entregar a mensagem
113        new SendingThread( clientSocket, from, message ).start();
114    }
115
116 }

```

Fig. 17.17 Implementação da interface `MessageManager` em `SocketMessageManager` para comunicação através de `Sockets` e `DatagramPackets` de *multicast* (parte 2 de 2).

O construtor `SocketMessageManager` (linhas 32 a 35) recebe o endereço do `DeitelMessengerServer` ao qual o `SocketMessageManager` deve se conectar. O método `connect` (linhas 38 a 63) conecta o `SocketMessageManager` ao `DeitelMessengerServer`. Se o `SocketMessageManager` estava conectado antes, a linha 42 retorna do método `connect`. As linhas 46 e 47 criam um novo `Socket` para se comunicar com o `DeitelMessengerServer`. A linha 47 cria um objeto `InetAddress` para o endereço do servidor e usa a constante `SERVER_PORT` para especificar a porta à qual o cliente deve se conectar. A linha 50 cria uma nova `PacketReceivingThread`, que espera as mensagens de *multicast* que chegam do `DeitelMessengerServer`. A linha 51 inicia a `PacketReceivingThread`. A linha 54 atualiza a variável booleana `connected` para indicar que o `SocketMessageManager` está conectado ao servidor.

O método `disconnect` (linhas 66 a 104) termina a conexão do `SocketMessageManager` ao `DeitelMessengerServer`. Se o `SocketMessageManager` não estiver conectado, a linha 70 retorna do método `disconnect`. As linhas 76 e 77 criam uma nova `SendingThread` (Fig. 17.18) para enviar `DISCONNECT_STRING` para o `DeitelMessengerServer`. A classe `SendingThread` entrega uma mensagem para o `DeitelMessengerServer` através da conexão por `Socket` do `SocketMessageManager`. A linha 78 inicia a `SendingThread` para entregar a mensagem. A linha 81 invoca o método `join` de `SendingThread` (herdado de `Thread`) para esperar que a mensagem de desconexão seja entregue. O argumento inteiro 10000 especifica que a *thread* atual deve esperar só 10 segundos para executar `join` para a `SendingThread` antes de continuar. Uma vez que a mensagem de desconexão tenha sido entregue, a linha 84 invoca o método `stopListening` da classe `PacketReceivingThread` para parar de receber mensagens de bate-papo que chegam. A linha 87 fecha a conexão por `Socket` ao `DeitelMessengerServer`.

O método `sendMessage` (linhas 107 a 115) envia uma mensagem destinada ao `DeitelMessengerServer`. Se o `SocketMessageManager` não estiver conectado, a linha 111 retorna do método `sendMessage`. A linha 114 cria e inicia uma nova instância de `SendingThread` (Fig. 17.18) para entregar a nova mensagem em uma *thread* de execução separada.

A classe `SendingThread` (Fig. 17.18) estende a classe `Thread` para entregar mensagens enviadas para o `DeitelMessengerServer` em uma *thread* de execução separada. O construtor de `SendingThread` (linhas 21 a 31) recebe como argumentos o `Socket` através do qual deve enviar a mensagem, o `userName` de quem veio a mensagem e o corpo da `message`. A linha 30 concatena o `userName`, `MESSAGE_SEPARATOR` e `message`, para construir a `messageToSend`. A constante `MESSAGE_SEPARATOR` permite que o receptor da mensagem separe a mensagem em duas partes – o nome do usuário que enviou e o corpo da mensagem – com um `StringTokenizer`.

```

1 // SendingThread.java
2 // SendingThread envia uma mensagem para o servidor de bate-papo
3 // em uma Thread separada.
4 package com.deitel.messenger.sockets.client;
5
6 // Pacotes do núcleo de Java
7 import java.io.*;
8 import java.net.*;
9
10 // Pacotes Deitel
11 import com.deitel.messenger.sockets.*;
12
13 public class SendingThread extends Thread
14     implements SocketMessengerConstants {
15
16     // Socket através do qual a mensagem deve ser enviada
17     private Socket clientSocket;
18     private String messageToSend;
19
20     // construtor SendingThread
21     public SendingThread( Socket socket, String userName,
22         String message )
23     {

```

Fig. 17.18 `SendingThread` para entregar mensagens enviadas para o `DeitelMessengerServer` (parte 1 de 2).

```

24     // invoca o construtor da superclasse para dar nome à Thread
25     super( "SendingThread: " + socket );
26
27     clientSocket = socket;
28
29     // constrói a mensagem a ser enviada
30     messageToSend = userName + MESSAGE_SEPARATOR + message;
31 }
32
33     // envia a mensagem e sai da Thread
34     public void run()
35     {
36         // envia a mensagem e descarrega PrintWriter
37         try {
38             PrintWriter writer =
39                 new PrintWriter( clientSocket.getOutputStream() );
40             writer.println( messageToSend );
41             writer.flush();
42         }
43
44         // trata exceção ocorrida durante o envio de mensagem
45         catch ( IOException ioException ) {
46             ioException.printStackTrace();
47         }
48     }
49 }
50 } // fim do método run

```

Fig. 17.18 SendingThread para entregar mensagens enviadas para o DeitelMessengerServer (parte 1 de 2).

O método `run` (linhas 34 a 49) entrega a mensagem completa ao `DeitelMessengerServer`, usando o `Socket` fornecido para o construtor `SendingThread`. As linhas 38 e 39 criam um novo `PrintWriter` para o `OutputStream` do `clientSocket`. A linha 40 invoca o método `println` da classe `PrintWriter` para enviar a mensagem. A linha 41 invoca o método `flush` da classe `PrintWriter` para assegurar que a mensagem seja enviada imediatamente. Observe que a classe `SendingThread` não fecha o `clientSocket`. A classe `SocketMessageManager` usa uma nova instância da classe `SendingThread` para cada mensagem que o cliente envia, de modo que o `clientSocket` deve permanecer aberto até que o usuário se desconecte do `DeitelMessengerServer`.

A classe `PacketReceivingThread` estende a classe `Thread` para permitir que o `SocketMessageManager` espere mensagens que chegam em uma *thread* de execução separada. A linha 19 declara o `MessageListener` ao qual a `PacketReceivingThread` irá entregar as mensagens que chegam. A linha 22 declara um `MulticastSocket`, que permite à `PacketReceivingThread` receber `DatagramPackets` de *multicast*. A linha 25 declara uma referência para `InetAddress` para o endereço de *multicast* para o qual o `DeitelMessengerServer` envia novas mensagens de bate-papo. O `MulticastSocket` se conecta a este `InetAddress` para esperar as mensagens de bate-papo que chegam.

```

1 // PacketReceivingThread.java
2 // PacketReceivingThread espera por DatagramPackets
3 // que contêm mensagens de um DeitelMessengerServer.
4 package com.deitel.messenger.sockets.client;
5
6 // Pacotes do núcleo de Java
7 import java.io.*;
8 import java.net.*;
9 import java.util.*;

```

Fig. 17.19 PacketReceivingThread para esperar por novas mensagens de *multicast* do DeitelMessengerServer em uma Thread separada (parte 1 de 4).

```

10 // Pacotes Deitel
11 import com.deitel.messenger.*;
12 import com.deitel.messenger.sockets.*;
13
14 public class PacketReceivingThread extends Thread
15     implements SocketMessengerConstants {
16
17     // MessageListener ao qual as mensagens devem ser entregues
18     private MessageListener messageListener;
19
20     // MulticastSocket para receber mensagens transmitidas
21     private MulticastSocket multicastSocket;
22
23     // InetAddress do grupo de mensagens
24     private InetAddress multicastGroup;
25
26     // indicador para terminar a PacketReceivingThread
27     private boolean keepListening = true;
28
29     // construtor PacketReceivingThread
30     public PacketReceivingThread( MessageListener listener )
31     {
32         // invoca o construtor da superclasse para dar nome à Thread
33         super( "PacketReceivingThread" );
34
35         // configura o MessageListener
36         messageListener = listener;
37
38         // conecta o MulticastSocket ao endereço e à porta de multicast
39         try {
40             multicastSocket =
41                 new MulticastSocket( MULTICAST_LISTENING_PORT );
42
43             multicastGroup =
44                 InetAddress.getByName( MULTICAST_ADDRESS );
45
46             // junta-se ao grupo de multicast para receber mensagens
47             multicastSocket.joinGroup( multicastGroup );
48
49             // ajusta tempo máximo de 5 segundos para esperar novos pacotes
50             multicastSocket.setSoTimeout( 5000 );
51         }
52
53
54         // trata exceção ocorrida durante conexão ao endereço de multicast
55         catch ( IOException ioException ) {
56             ioException.printStackTrace();
57         }
58
59     } // fim do construtor PacketReceivingThread
60
61     // espera as mensagens do grupo de multicast
62     public void run()
63     {
64         // espera as mensagens até ser parada
65         while ( keepListening ) {
66
67             // cria buffer para mensagens que chegam
68             byte[] buffer = new byte[ MESSAGE_SIZE ];

```

Fig. 17.19 PacketReceivingThread para esperar por novas mensagens de *multicast* do Deitel-MessengerServer em uma Thread separada (parte 2 de 4).

```

69
70     // cria DatagramPacket para mensagem que chega
71     DatagramPacket packet = new DatagramPacket( buffer,
72         MESSAGE_SIZE );
73
74     // recebe novo DatagramPacket (chamada bloqueante)
75     try {
76         multicastSocket.receive( packet );
77     }
78
79     // trata exceção quando esgotado tempo de espera na recepção
80     catch ( InterruptedIOException interruptedIOException ) {
81
82         // continua para a próxima iteração, para continuar esperando
83         continue;
84     }
85
86     // trata de exceção durante a leitura de pacote do grupo de multicast
87     catch ( IOException ioException ) {
88         ioException.printStackTrace();
89         break;
90     }
91
92     // coloca dados da mensagem em um String
93     String message = new String( packet.getData() );
94
95     // assegura que a mensagem não está vazia
96     if ( message != null ) {
97
98         // elimina espaço em branco extra do fim da mensagem
99         message = message.trim();
100
101        // separa a mensagem em tokens para recuperar
102        // o nome do usuário e o corpo da mensagem
103        StringTokenizer tokenizer =
104            new StringTokenizer( message, MESSAGE_SEPARATOR );
105
106        // ignora mensagens que não contêm um nome
107        // de usuário e um corpo de mensagem
108        if ( tokenizer.countTokens() == 2 ) {
109
110            // envia a mensagem para o MessageListener
111            messageListener.messageReceived(
112                tokenizer.nextToken(),           // nome do usuário
113                tokenizer.nextToken() );      // corpo da mensagem
114        }
115
116    } // fim do if
117
118 } // fim do while
119
120 // sai do grupo de multicast e fecha o MulticastSocket
121 try {
122     multicastSocket.leaveGroup( multicastGroup );
123     multicastSocket.close();
124 }
125
126 // trata exceção durante a leitura de pacote do grupo de multicast
127 catch ( IOException ioException ) {

```

Fig. 17.19 PacketReceivingThread para esperar por novas mensagens de *multicast* do Deitel-MessengerServer em uma Thread separada (parte 3 de 4).

```

128         ioException.printStackTrace();
129     }
130
131 } // fim do método run
132
133 // para de esperar por novas mensagens
134 public void stopListening()
135 {
136     // termina a Thread
137     keepListening = false;
138 }
139 }
```

Fig. 17.19 PacketReceivingThread para esperar por novas mensagens de *multicast* do DeitelMessengerServer em uma Thread separada (parte 4 de 4).

O construtor **PacketReceivingThread** (linhas 31 a 59) recebe como argumento o **MessageListener** para o qual a **PacketReceivingThread** deve entregar as mensagens que chegam. Lembre-se de que a interface **MessageListener** define um único método **messageReceived**. Quando a **PacketReceivingThread** recebe uma nova mensagem de bate-papo através do **MulticastSocket**, a **PacketReceivingThread** invoca o método **messageReceived** para entregar a nova mensagem para o **MessageListener**.

As linhas 41 e 42 criam um novo **MulticastSocket** e passam para o construtor **MulticastSocket** a constante **MULTICAST_LISTENING_PORT** da interface **SocketMessengerConstants**. Este argumento especifica a porta na qual o **MulticastSocket** deve esperar pelas mensagens de bate-papo que chegam. As linhas 44 e 45 criam um objeto **InetAddress** para o **MULTICAST_ADDRESS**, para o qual o **DeitelMessengerServer** transmite novas mensagens de bate-papo. A linha 48 invoca o método **joinGroup** da classe **MulticastSocket** para registrar o **MulticastSocket** para receber mensagens enviadas para **MULTICAST_ADDRESS**. A linha 51 invoca o método **setSoTimeout** para especificar que, se nenhum dado for recebido dentro de 5000 milissegundos, o **MulticastSocket** deve disparar uma **InterruptedException**, a qual a *thread* atual pode capturar e, então, continuar a ser executada. Esta abordagem evita que a **PacketReceivingThread** entre em *deadlock* quando estiver esperando pelos dados que chegam. Além disso, se o **MulticastSocket** nunca terminasse, o laço **while** não seria capaz de verificar a variável **keepListening** e, portanto, impediria que a **PacketReceivingThread** parasse se **keepListening** fosse ajustada para **false**.

O método **run** (linhas 62 a 131) espera pelas mensagens *multicast* que chegam. A linha 68 cria um *array* de bytes no qual se deve armazenar os dados do **DatagramPacket** que chega. As linhas 71 e 72 criam um **DatagramPacket** para armazenar a mensagem que chega. A linha 76 invoca o método **receive** da classe **MulticastSocket** com o pacote **DatagramPacket** como argumento. Esta é uma chamada bloqueante que lê um pacote que está chegando do endereço de *multicast*. Se passarem 5000 milissegundos sem recepção de um pacote, o método **receive** dispara uma **InterruptedException**, porque configuramos previamente um tempo máximo de espera de 5000 milissegundos (linha 51). A linha 83 usa a palavra-chave **continue** para prosseguir para a próxima iteração do laço **while**, para continuar a esperar pelas mensagens que chegam. Para outras **IOExceptions**, a linha 89 sai do laço **while** com **break** para terminar a **PacketReceivingThread**.

A linha 93 invoca o método **getData** da classe **DatagramPacket** para recuperar os dados da mensagem. A linha 99 invoca o método **trim** da classe **String** para remover os espaços em branco extras do fim da mensagem. Lembre-se de que **DatagramPackets** têm tamanho fixo – 512 bytes neste exemplo – assim, se a mensagem é mais curta do que 512 bytes, haverá espaço em branco extra após a mensagem. As linhas 103 e 104 criam um **StringTokenizer** para separar o corpo da mensagem do nome do usuário que enviou a mensagem. A linha 108 verifica se a quantidade de *tokens* está correta. As linhas 111 a 113 invocam o método **messageReceived** da interface **MessageListener** para entregar a mensagem que chega ao **MessageListener** da **PacketReceivingThread**.

Se o programa invoca o método **stopListening** (linhas 134 a 138), o laço **while** no método **run** (linhas 62 a 118) termina. A linha 122 invoca o método **leaveGroup** da classe **MulticastSocket** para parar de receber mensagens do endereço de *multicast*. A linha 123 invoca o método **close** da classe **MulticastSocket** para fechar o **MulticastSocket**. A **PacketReceivingThread** então termina quando ocorre o retorno do método **run**.

A classe **ClientGUI** (Fig. 17.20) estende a classe **JFrame** para criar uma GUI para um usuário que envia e recebe mensagens de bate-papo. A GUI consiste em uma **JTextArea** para exibir mensagens que chegam (linha

22), uma **JTextArea** para digitar novas mensagens (linha 23), **JButtons** e **JMenuItems** para se conectar ao servidor e se desconectar dele (linhas 26 a 29) e um **JButton** para enviar mensagens (linha 32). A GUI também contém um **JLabel** que indica se o cliente está conectado ou desconectado.

```

1 // ClientGUI.java
2 // ClientGUI fornece uma interface com o usuário para enviar e receber
3 // mensagens de DeitelMessengerServer e para ele.
4 package com.deitel.messenger;
5
6 // Pacotes do núcleo de Java
7 import java.io.*;
8 import java.net.*;
9 import java.awt.*;
10 import java.awt.event.*;
11
12 // Pacotes de extensão de Java
13 import javax.swing.*;
14 import javax.swing.border.*;
15
16 public class ClientGUI extends JFrame {
17
18     // JMenu para conectar/desconectar servidor
19     private JMenu serverMenu;
20
21     // JTextAreas para exibir e digitar mensagens
22     private JTextArea messageArea;
23     private JTextArea inputArea;
24
25     // JButtons e JMenuItems para conectar e desconectar
26     private JButton connectButton;
27     private JMenuItem connectMenuItem;
28     private JButton disconnectButton;
29     private JMenuItem disconnectMenuItem;
30
31     // JButton para enviar mensagens
32     private JButton sendButton;
33
34     // JLabel para exibir o estado da conexão
35     private JLabel statusBar;
36
37     // userName para adicionar a mensagens enviadas
38     private String userName;
39
40     // MessageManager para se comunicar com o servidor
41     private MessageManager messageManager;
42
43     // MessageListener para receber mensagens que chegam
44     private MessageListener messageListener;
45
46     // construtor ClientGUI
47     public ClientGUI( MessageManager manager )
48     {
49         super( "Deitel Messenger" );
50
51         // configura o MessageManager
52         messageManager = manager;
53
54         // cria MyMessageListener para receber mensagens

```

Fig. 17.20 Subclasse ClientGUI de **JFrame** que apresenta uma GUI para visualizar e enviar mensagens de bate-papo (parte 1 de 5).

```

55     messageListener = new MyMessageListener();
56
57     // cria o Jmenu File
58     serverMenu = new JMenu( "Server" );
59     serverMenu.setMnemonic( 'S' );
60     JMenuBar menuBar = new JMenuBar();
61     menuBar.add( serverMenu );
62     setJMenuBar( menuBar );
63
64     // cria ImageIcon para os botões de conectar
65     Icon connectIcon = new ImageIcon(
66         getClass().getResource( "images/Connect.gif" ) );
67
68     // cria connectButton e connectMenuItem
69     connectButton = new JButton( "Connect", connectIcon );
70     connectMenuItem = new JMenuItem( "Connect", connectIcon );
71     connectMenuItem.setMnemonic( 'C' );
72
73     // cria ConnectListener para os botões de conectar
74     ActionListener connectListener = new ConnectListener();
75     connectButton.addActionListener( connectListener );
76     connectMenuItem.addActionListener( connectListener );
77
78     // cria ImageIcon para os botões de desconectar
79     Icon disconnectIcon = new ImageIcon(
80         getClass().getResource( "images/Disconnect.gif" ) );
81
82     // cria disconnectButton e disconnectMenuItem
83     disconnectButton = new JButton( "Disconnect",
84         disconnectIcon );
85     disconnectMenuItem = new JMenuItem( "Disconnect",
86         disconnectIcon );
87     disconnectMenuItem.setMnemonic( 'D' );
88
89     // desabilita os botões de desconectar
90     disconnectButton.setEnabled( false );
91     disconnectMenuItem.setEnabled( false );
92
93     // cria DisconnectListener para os botões de desconectar
94     ActionListener disconnectListener =
95         new DisconnectListener();
96     disconnectButton.addActionListener( disconnectListener );
97     disconnectMenuItem.addActionListener( disconnectListener );
98
99     // adiciona os JMenuItems connect e disconnect ao fileMenu
100    serverMenu.add( connectMenuItem );
101    serverMenu.add( disconnectMenuItem );
102
103    // adiciona os JButtons connect e disconnect ao buttonPanel
104    JPanel buttonPanel = new JPanel();
105    buttonPanel.add( connectButton );
106    buttonPanel.add( disconnectButton );
107
108    // cria a JTextArea para exibir mensagens
109    messageArea = new JTextArea();
110
111    // desabilita a edição e a mudança automática de linha
112    messageArea.setEditable( false );
113    messageArea.setWrapStyleWord( true );

```

Fig. 17.20 Subclasse ClientGUI de JFrame que apresenta uma GUI para visualizar e enviar mensagens de bate-papo (parte 2 de 5).

```

114     messageArea.setLineWrap( true );
115
116     // coloca a messageArea no JScrollPane para permitir rolagem
117     JPanel messagePanel = new JPanel();
118     messagePanel.setLayout( new BorderLayout( 10, 10 ) );
119     messagePanel.add( new JScrollPane( messageArea ),
120         BorderLayout.CENTER );
121
122     // cria JTextArea para digitar novas mensagens
123     inputArea = new JTextArea( 4, 20 );
124     inputArea.setWrapStyleWord( true );
125     inputArea.setLineWrap( true );
126     inputArea.setEditable( false );
127
128     // cria Icon para sendButton
129     Icon sendIcon = new ImageIcon(
130         getClass().getResource( "images/Send.gif" ) );
131
132     // cria sendButton e o desabilita
133     sendButton = new JButton( "Send", sendIcon );
134     sendButton.setEnabled( false );
135
136     // cria ActionListener para sendButton
137     sendButton.addActionListener(
138         new ActionListener() {
139
140             // envia nova mensagem quando o usuário ativa o sendButton
141             public void actionPerformed( ActionEvent event )
142             {
143                 messageManager.sendMessage( userName,
144                     inputArea.getText());
145
146                 // limpa a inputArea
147                 inputArea.setText( "" );
148             }
149         } // fim do ActionListener
150     );
151
152     // coloca a inputArea e o sendButton em um BoxLayout
153     // e adiciona Box ao messagePanel
154     Box box = new Box( BoxLayout.X_AXIS );
155     box.add( new JScrollPane( inputArea ) );
156     box.add( sendButton );
157     messagePanel.add( box, BorderLayout.SOUTH );
158
159     // cria JLabel para a statusBar com borda reentrante
160     statusBar = new JLabel( "Not Connected" );
161     statusBar.setBorder(
162         new BevelBorder( BevelBorder.LOWERED ) );
163
164     // dispõe os componentes no JFrame
165     Container container = getContentPane();
166     container.add( buttonPanel, BorderLayout.NORTH );
167     container.add( messagePanel, BorderLayout.CENTER );
168     container.add( statusBar, BorderLayout.SOUTH );
169
170     // adiciona o WindowListener para se desconectar quando o usuário sair
171     addWindowListener (
172         new WindowAdapter () {

```

Fig. 17.20 Subclasse ClientGUI de JFrame que apresenta uma GUI para visualizar e enviar mensagens de bate-papo (parte 3 de 5).

```

173
174      // desconecta-se do servidor e sai do aplicativo
175      public void windowClosing ( WindowEvent event )
176      {
177          messageManager.disconnect( messageListener );
178          System.exit( 0 );
179      }
180  };
181  };
182
183 } // fim do construtor ClientGUI
184
185 // ConnectListener espera os pedidos do usuário
186 // para se conectar ao DeitelMessengerServer
187 private class ConnectListener implements ActionListener {
188
189     // conecta-se ao servidor e habilita/desabilita componentes da GUI
190     public void actionPerformed( ActionEvent event )
191     {
192         // conecta-se ao servidor e roteia mensagens
193         // para o messageListener
194         messageManager.connect( messageListener );
195
196         // pede para o usuário digitar seu nome
197         userName = JOptionPane.showInputDialog(
198             ClientGUI.this, "Enter user name:" );
199
200         // limpa a messageArea
201         messageArea.setText( "" );
202
203         // atualiza componentes da GUI
204         connectButton.setEnabled( false );
205         connectMenuItem.setEnabled( false );
206         disconnectButton.setEnabled( true );
207         disconnectMenuItem.setEnabled( true );
208         sendButton.setEnabled( true );
209         inputArea.setEditable( true );
210         inputArea.requestFocus();
211         statusBar.setText( "Connected: " + userName );
212     }
213
214 } // fim da classe interna ConnectListener
215
216 // DisconnectListener espera os pedidos do usuário
217 // para se desconectar do DeitelMessengerServer
218 private class DisconnectListener implements ActionListener {
219
220     // desconecta-se do servidor e habilita/desabilita componentes da GUI
221     public void actionPerformed( ActionEvent event )
222     {
223         // desconecta-se do servidor e pára de rotear mensagens
224         // para o messageListener
225         messageManager.disconnect( messageListener );
226
227         // atualiza componentes da GUI
228         sendButton.setEnabled( false );
229         disconnectButton.setEnabled( false );
230         disconnectMenuItem.setEnabled( false );
231         inputArea.setEditable( false );

```

Fig. 17.20 Subclasse ClientGUI de JFrame que apresenta uma GUI para visualizar e enviar mensagens de bate-papo (parte 4 de 5).

```

232     connectButton.setEnabled( true );
233     connectMenuItem.setEnabled( true );
234     statusBar.setText( "Not Connected" );
235 }
236
237 } // fim da classe interna DisconnectListener
238
239 // MyMessageListener espera novas mensagens do
240 // MessageManager e exibe as mensagens na messageArea
241 // usando um MessageDisplayer.
242 private class MyMessageListener implements MessageListener {
243
244     // quando recebidas, exibe mensagens na messageArea
245     public void messageReceived( String from, String message )
246     {
247         // anexa a mensagem com MessageDisplayer e invokeLater,
248         // assegurando que messageArea seja segura para o acesso de threads
249         SwingUtilities.invokeLater(
250             new MessageDisplayer( from, message ) );
251
252     } // fim do método messageReceived
253
254 } // fim da classe interna MyMessageListener
255
256 // MessageDisplayer exibe uma nova mensagem anexando a
257 // mensagem à JTextArea messageArea. Este objeto
258 // Runnable deve ser executado somente na thread Event,
259 // porque ele modifica um componente Swing ativo.
260 private class MessageDisplayer implements Runnable {
261
262     private String fromUser;
263     private String messageBody;
264
265     // construtor MessageDisplayer
266     public MessageDisplayer( String from, String body )
267     {
268         fromUser = from;
269         messageBody = body;
270     }
271
272     // exibe nova mensagem em messageArea
273     public void run()
274     {
275         // anexa nova mensagem
276         messageArea.append( "\n" + fromUser + "> " +
277             messageBody );
278
279         // move caret para o fim da messageArea para assegurar
280         // que a nova mensagem seja visivel na tela
281         messageArea.setCaretPosition(
282             messageArea.getText().length() );
283     }
284
285 } // fim da classe interna MessageDisplayer
286 }

```

Fig. 17.20 Subclasse ClientGUI de JFrame que apresenta uma GUI para visualizar e enviar mensagens de bate-papo (parte 5 de 5).

ClientGUI usa um **MessageManager** (linha 41) para tratar toda a comunicação com o servidor de bate-papo. Lembre-se de que **MessageManager** é uma interface e, portanto, permite que ClientGUI use qualquer implemen-

tação de `MessageManager` sem necessidade de mudar qualquer código em `ClientGUI`. A classe `ClientGUI` também usa um `MessageListener` (linha 44) para receber as mensagens que chegam do `MessageManager`.

O construtor `ClientGUI` (linhas 47 a 183) recebe como argumento o `MessageManager` para se comunicar com o `DeitelMessengerServer`. A linha 52 configura o `MessageManager` do `ClientGUI`. A linha 55 cria uma instância de `MyMessageListener`, a qual implementa a interface `MessageListener`. As linhas 58 a 62 criam um menu `Server` que contém `JMenuItem`s para se conectar ao servidor de bate-papo e se desconectar dele. As linhas 65 a 66 criam um `ImageIcon` para `connectButton` e `connectMenuItem`.

A linha 66 invoca o método `getClass` (herdado da classe `Object`) para recuperar o objeto `Class` que representa a definição da classe `ClientGUI`. A linha 66 então invoca o método `getResource` da classe `Class` para carregar a imagem de conectar. A máquina virtual de Java carrega as definições de classes na memória usando um *carregador de classes*. O método `getResource` de `Class` usa o carregador de classe do objeto `Class` para especificar o endereço de um recurso, como um arquivo de imagem. Especificar a localização de recursos desta maneira permite que se evitem caminhos codificados internamente ou absolutos, que podem tornar os programas mais difíceis de instalar. Usar as técnicas descritas aqui permite a um *applet* ou aplicativo carregar arquivos de endereços que sejam relativos ao endereço do arquivo `.class` para uma determinada classe.

As linhas 69 a 70 criam `connectButton` e `connectMenuItem`, cada um com o rótulo "Connect" e o `Icon connectIcon`. A linha 71 invoca o método `setMnemonic` da classe `JMenuItem` para configurar o caractere mnemônico para acesso pelo teclado ao `connectMenuItem`. A linha 74 cria uma instância da classe interna `private ConnectListener`, que implementa a interface `ActionListener` para tratar `ActionEvents` de `connectButton` e `connectMenuItem`. As linhas 75 a 76 adicionam `connectListener` como um `ActionListener` para `connectButton` e `connectMenuItem`.

As linhas 79 e 80 criam um `ImageIcon` para os componentes `disconnectButton` e `disconnectMenuItem`. As linhas 83 a 86 criam `disconnectButton` e `disconnectMenuItem`, cada um com o rótulo "Disconnect" e o `Icon disconnectIcon`.

A linha 87 invoca o método `setMnemonic` da classe `JMenuItem` para permitir o acesso através do teclado ao `disconnectMenuItem`. As linhas 90 e 91 invocam o método `setEnabled` da classe `JButton` e da classe `JMenuItem` com um argumento `false` para desabilitar `disconnectButton` e `disconnectMenuItem`. Isso evita que o usuário tente se desconectar do servidor porque o cliente ainda não está conectado. As linhas 94 e 95 criam uma instância da classe interna `private DisconnectListener`, a qual implementa a interface `ActionListener` para tratar de `ActionEvents` de `disconnectButton` e `disconnectMenuItem`. As linhas 96 e 97 adicionam `disconnectListener` como um `ActionListener` para os componentes `disconnectButton` e `disconnectMenuItem`.

As linhas 100 e 101 adicionam `connectMenuItem` e `disconnectMenuItem` ao `JMenu Server`. As linhas 104 a 106 criam um `JPanel` e adicionam `connectButton` e `disconnectButton` àquele `JPanel`. A linha 109 cria a `JTextArea messageArea`, na qual o cliente exibe as mensagens que chegam. A linha 112 invoca o método `setEnabled` com um argumento `false`, para desabilitar a edição do texto em `messageArea`. As linhas 113 e 114 invocam os métodos `setWrapStyleWord` e `setLineWrap` da classe `JTextArea` para permitir o *wrapping* de palavras em `messageArea`. Se uma mensagem é mais longa do que a largura da `messageArea`, a `messageArea` irá passar para a linha seguinte o texto após a última palavra que cabe em cada linha, facilitando a leitura das mensagens mais longas. As linhas 117 a 120 criam um `JPanel` para a `messageArea` e adicionam a `messageArea` ao `JPanel` em um `JScrollPane`. O `JScrollPane` adiciona barras de rolagem à `messageArea` para permitir que o usuário role as mensagens que excedem o tamanho da `messageArea`.

A linha 123 cria a `inputArea JTextArea` para digitação de novas mensagens. Os argumentos para o construtor `JTextArea` especificam uma `JTextArea` de quatro linhas que tem 20 caracteres de largura. As linhas 124 e 125 permitem o *wrapping* de palavras e de linhas, e a linha 126 desabilita a edição da `inputArea`. Quando o cliente se conecta ao servidor de bate-papo, `ConnectListener` habilita `inputArea` para permitir que o usuário digite novas mensagens.

A linha 129 cria um `ImageIcon` para `sendButton`. A linha 133 cria `sendButton`, no qual o usuário pode clicar para enviar uma mensagem que o usuário digitou. A linha 134 desabilita `sendButton`; o `ConnectListener` habilita o `sendButton` quando o cliente se conecta ao servidor de bate-papo. As linhas 137 a 150 adicionam um `ActionListener` a `sendButton`. As linhas 143 e 144 invocam o método `sendMessage` da interface `MessageManager` com o `userName` e o texto da `inputArea` como argumentos. Esta instrução envia o nome do usuário e qualquer texto que o usuário digitou na `inputArea` para o `DeitelMessengerServer`.

como uma nova mensagem de bate-papo. A linha 147 invoca o método `setText` da classe `JTextArea` com um argumento `String` vazio para limpar a `inputArea` para a próxima mensagem.

As linhas 154 a 157 usam um `BoxLayout` para dispor a `inputArea` e `sendButton`. A linha 155 coloca `inputArea` em um `JScrollPane` para permitir a rolagem de mensagens longas. A linha 157 adiciona a `Box` que contém `inputArea` e `sendButton` à região `SOUTH` do `messagePanel`. As linhas 160 a 162 criam o `JLabel` `statusBar`. Este `JLabel` exibe se o usuário está conectado ou não ao servidor de bate-papo. As linhas 161 e 162 invocam o método `setBorder` da classe `JLabel` e criam uma nova `BevelBorder` do tipo `BevelBorder.LOWERED`. Esta borda faz o `JLabel` parecer pressionado, como é comum com a barra de estado em muitos aplicativos. As linhas 165 a 168 dispõem o `buttonPanel`, o `messagePanel` e a `statusBar` no `JFrame` de `ClientGUI`.

As linhas 171 a 181 adicionam um `WindowListener` ao `JFrame` `ClientGUI`. A linha 177 invoca o método `disconnect` da interface `MessageManager` para se desconectar do servidor de bate-papo caso o usuário saia enquanto ainda estiver conectado.

A classe interna `ConnectListener` (linhas 187 a 214) trata de eventos de `connectButton` e `connectMenuItem`. A linha 194 invoca o método `connect` da classe `MessageManager` para se conectar com o servidor de bate-papo. A linha 194 passa como argumento para o método `connect` o `MessageListener` para o qual novas mensagens devem ser entregues. As linhas 197 e 198 pedem ao usuário para digitar um nome de usuário, e a linha 201 limpa a `JTextArea messageArea`. As linhas 204 a 209 habilitam os componentes para se desconectar do servidor e para enviar mensagens e desabilitam os componentes para se conectar ao servidor. A linha 210 invoca o método `requestFocus` da classe `JTextArea` para colocar o cursor de entrada de texto na `inputArea` para que o usuário possa começar a digitar uma mensagem mais facilmente.

A classe interna `DisconnectListener` (linhas 218 a 237) trata de eventos de `disconnectButton` e `disconnectMenuItem`. A linha 225 invoca o método `disconnect` da classe `MessageManager` para se desconectar do servidor de bate-papo. As linhas 228 a 234 desabilitam os componentes para enviar mensagens e os componentes para desconectar e então habilitam os componentes para se conectar ao servidor de bate-papo.

A classe interna `MyMessageListener` (linhas 242 a 254) implementa a interface `MessageListener` para receber as mensagens que chegam do `MessageManager`. Quando uma nova mensagem é recebida, o programa invoca o método `messageReceived` (linhas 242 a 252) com o nome do usuário do remetente e o corpo da mensagem. As linhas 249 e 250 invocam o método `static invokeLater` da classe `SwingUtilities` com uma nova instância de `MessageDisplayer` para anexar a nova mensagem à `messageArea`. Lembre-se, do Capítulo 15, de que os componentes Swing devem ser acessados somente a partir da *thread* de despacho de eventos. O método `messageReceived` é invocado pela `PacketReceivingThread` na classe `SocketMessageManager` e portanto não pode anexar o texto da mensagem à `messageArea` diretamente, pois isto aconteceria na `PacketReceivingThread`, não na *thread* de despacho de eventos.

A classe interna `MessageDisplayer` (linhas 260 a 285) implementa a interface `Runnable` para fornecer uma maneira segura em termos de *threads* de anexar texto à `JTextArea messageArea`. O construtor `MessageDisplayer` (linhas 266 a 270) recebe como argumentos o nome do usuário e a mensagem a enviar. O método `run` (linhas 273 a 283) anexa o nome do usuário, ">" e `messageBody` à `messageArea`. As linhas 281 e 282 invocam o método `setCaretPosition` da classe `JTextArea` para rolar a `messageArea` até o fundo, para exibir a mensagem recebida mais recentemente. Instâncias da classe `MessageDisplayer` devem ser executadas somente como parte da *thread* de despacho de eventos, para assegurar acesso seguro em termos de *threads* ao componente Swing `messageArea`.

A classe `DeitelMessenger` (Fig. 17.21) dispara o cliente para o `DeitelMessengerServer`. As linhas 18 a 21 criam um novo `SocketMessageManager` para se conectar ao `DeitelMessengerServer` com o endereço IP especificado como argumento de linha de comando para o aplicativo. As linhas 24 a 27 criam um `ClientGUI` para o `MessageManager`, configuraram o tamanho do `ClientGUI` e tornam o `ClientGUI` visível.

```

1 // DeitelMessenger.java
2 // DeitelMessenger é um aplicativo de bate-papo que usa
3 // uma ClientGUI e um SocketMessageManager para se
4 // comunicar com o DeitelMessengerServer.

```

Fig. 17.21 Aplicativo `DeitelMessenger` para participar de uma sessão de bate-papo do `DeitelMessengerServer` (parte 1 de 3).

```

5  package com.deitel.messenger.sockets.client;
6
7  // Pacotes Deitel
8  import com.deitel.messenger.*;
9
10 public class DeitelMessenger {
11
12     // executa o aplicativo
13     public static void main( String args[] )
14     {
15         MessageManager messageManager;
16
17         // cria um novo DeitelMessenger
18         if ( args.length == 0 )
19             messageManager = new SocketMessageManager( "localhost" );
20         else
21             messageManager = new SocketMessageManager( args[ 0 ] );
22
23         // cria a GUI para o SocketMessageManager
24         ClientGUI clientGUI = new ClientGUI( messageManager );
25         clientGUI.setSize( 300, 400 );
26         clientGUI.setResizable( false );
27         clientGUI.setVisible( true );
28     }
29 }
```

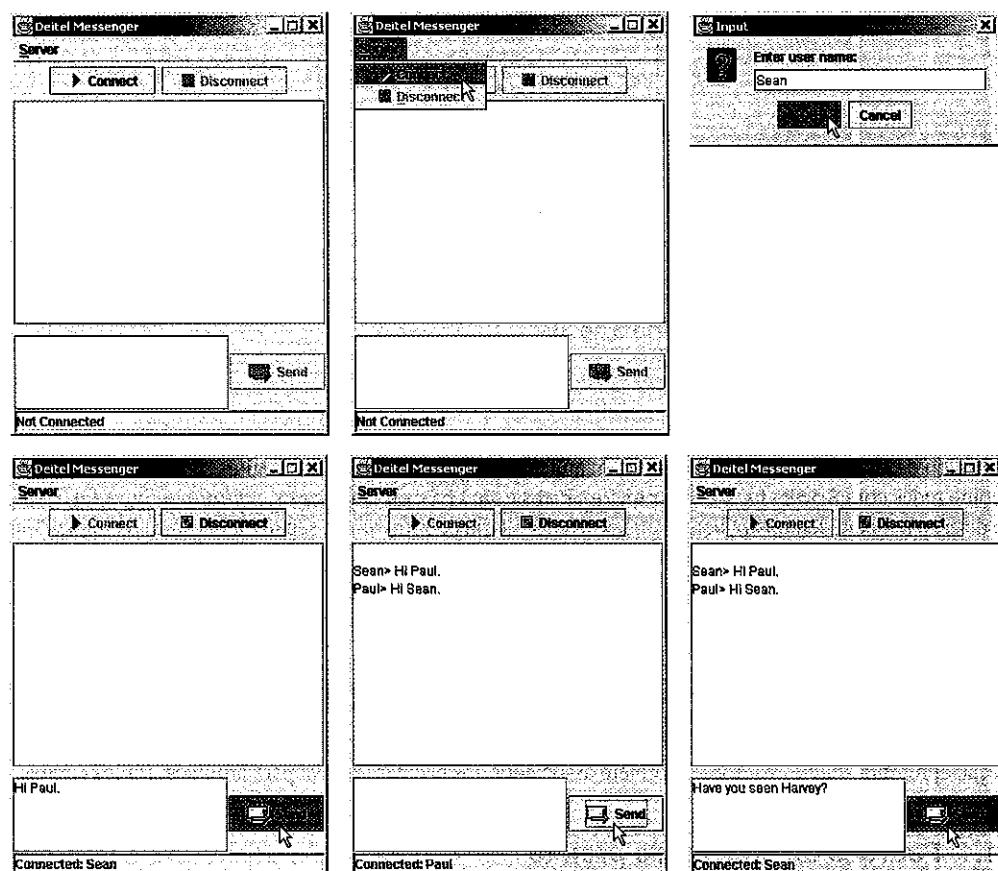


Fig. 17.21 Aplicativo DeitelMessenger para participar de uma sessão de bate-papo do DeitelMessengerServer (parte 2 de 3).

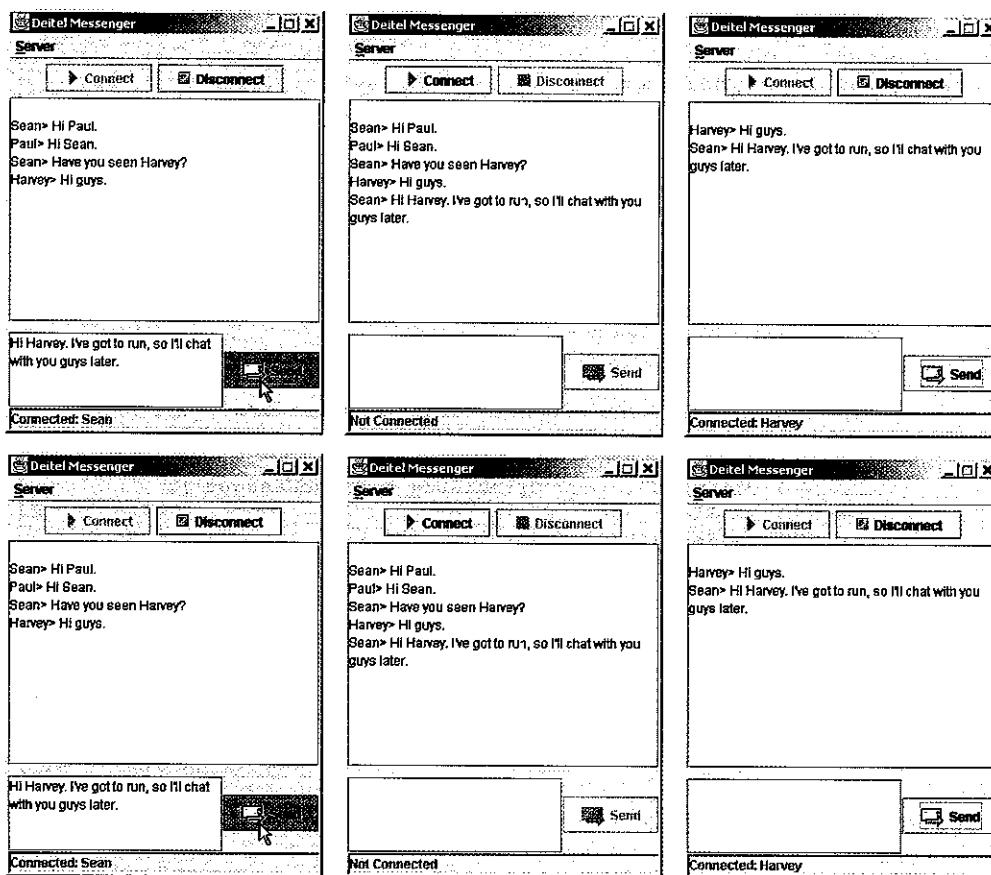


Fig. 17.21 Aplicativo DeitelMessenger para participar de uma sessão de bate-papo do DeitelMessengerServer (parte 3 de 3).

O estudo de caso do Deitel Messenger é um aplicativo significativo que usa muitos recursos intermediários de Java, como redes com **Sockets**, **DatagramPackets** e **MulticastSockets**, **multithreading** e **GUI Swing**. O estudo de caso também demonstra boas práticas de engenharia de *software* ao separar a interface da implementação, permitindo que os desenvolvedores construam **MessageManagers** para diferentes protocolos de redes e **MessageListeners** que fornecem diferentes interfaces com o usuário. Você agora deve ser capaz de aplicar estas técnicas a seus próprios projetos em Java.

17.11 (Opcional) Descobrindo padrões de projeto: padrões de projeto utilizados nos pacotes `java.io` e `java.net`

Esta seção apresenta aqueles padrões de projeto associados aos pacotes de Java para arquivos, fluxos e redes.

17.11.1 Padrões de criação de projeto

Continuamos agora nossa discussão dos padrões de criação.

Abstract Factory

Assim como o padrão de projeto *Factory Method*, o padrão de projeto *Abstract Factory* permite que um sistema determine a subclasse para a qual se deve instanciar um objeto durante a execução. Freqüentemente, esta subclasse não é conhecida durante o desenvolvimento. Entretanto, o *Abstract Factory* usa um objeto conhecido como uma fábrica que usa uma interface para instanciar objetos. A fábrica cria um produto; neste caso, o produto é um objeto de uma subclasse determinada durante a execução.

A biblioteca de soquetes de Java no pacote `java.net` usa o padrão de projeto *Abstract Factory*. O soquete descreve uma conexão, ou um fluxo de dados, entre dois computadores. A classe `Socket` faz referência a um objeto da subclasse `SocketImpl` (Seção 17.5). A classe `Socket` também contém uma referência `static` para um objeto que implementa a interface `SocketImplFactory`. O construtor `Socket` invoca o método `createSocketImpl` da interface `SocketFactory` para criar o objeto `SocketImpl`. O objeto que implementa a interface `SocketFactory` é a fábrica e o objeto de uma subclasse `SocketImpl` é o produto feito por ela. O sistema não pode especificar a subclasse `SocketImpl` da qual se deve instanciar a não ser durante a execução, porque o sistema não tem conhecimento de qual tipo de implementação de `Socket` é necessário (por exemplo, um soquete configurado para os requisitos de segurança da rede de área local). O método `createSocketImpl` decide de qual subclasse de `SocketImpl` se deve instanciar o objeto durante a execução.

17.11.2 Padrões estruturais de projeto

Esta seção encerra nossa discussão sobre os padrões estruturais.

Decorator

Examinemos a classe `CreateSequentialFile` (Fig. 16.6). As linhas 127 e 128 desta classe permitem que um objeto `ObjectOutputStream`, que grava objetos em um arquivo, assuma as responsabilidades de um objeto `FileOutputStream`, que fornece métodos para gravar `bytes` em arquivos. A classe `CreateSequentialFile` parece “encadear” os objetos – o objeto `FileOutputStream` é o argumento para o construtor do `ObjectOutputStream`. O fato de o objeto `ObjectOutputStream` poder assumir o comportamento de um `FileOutputStream` dinamicamente evita a necessidade de se criar uma classe separada chamada `ObjectFileOutputStream`, que iria implementar os comportamentos de ambas as classes.

As linhas 127 e 128 da classe `CreateSequentialFile` mostram um exemplo do padrão de projeto *Decorator*, que permite que um objeto assuma responsabilidades adicionais dinamicamente. Usando este padrão, os projetistas não precisam criar classes separadas desnecessárias para adicionar responsabilidades aos objetos de uma determinada classe.

Consideremos um exemplo mais complexo para descobrir como o padrão de projeto *Decorator* pode simplificar a estrutura de um sistema. Suponha que desejemos melhorar o desempenho de E/S do exemplo anterior usando um `BufferedOutputStream`. Usando o padrão de projeto *Decorator*, escreveríamos

```
output = new ObjectOutputStream(
    new BufferedOutputStream(
        new FileOutputStream( fileName ) ) );
```

Podemos encadear objetos desta maneira, porque `ObjectOutputStream`, `BufferedOutputStream` e `FileOutputStream` estendem a superclasse abstrata `OutputStream` e cada construtor de subclasse recebe um objeto `OutputStream` como parâmetro. Se os objetos de fluxo no pacote `java.io` não utilizassem o padrão de projeto *Decorator* (isto é, não satisfizessem estes dois requisitos), o pacote `java.io` teria que fornecer classes `BufferedFileOutputStream`, `ObjectBufferedOutputStream`, `ObjectBufferedFileOutputStream` e `ObjectFileOutputStream`. Imagine quantas classes teríamos que ter criado se tivéssemos encadeado ainda mais objetos de fluxo sem aplicar o padrão *Decorator*.

Facade

Ao dirigir um carro, você sabe que pressionar o pedal do acelerador acelera o carro, mas você ignora a maneira exata como o pedal faz o carro acelerar. Este princípio é a base do padrão de projeto *Facade*, que permite que um objeto – chamado de *objeto fachada* – forneça uma interface simples para os comportamentos de um *subsistema* – um agregado de objetos que assume em conjunto a responsabilidade principal do sistema. O pedal do acelerador, por exemplo, é o objeto de fachada para o subsistema de aceleração do carro, o volante é o objeto fachada para o subsistema de direção e o freio é o objeto fachada para o subsistema de desaceleração. O *objeto cliente* usa o objeto facha-

da para acessar os objetos por trás da fachada. O cliente permanece sem saber como os objetos por trás da fachada vão atender às responsabilidades, de modo que a complexidade do subsistema é escondida do cliente. Quando você pressiona o acelerador, você age como um objeto cliente. O padrão de projeto *Facade* reduz a complexidade do sistema porque o cliente interage com apenas um objeto (a fachada). Este padrão isola os desenvolvedores de aplicativos das complexidades do subsistema. Os desenvolvedores precisam estar familiarizados só com as operações do objeto fachada, e não com as operações mais detalhadas de todo o subsistema.

No pacote `java.net`, o objeto da classe `URL` é um objeto fachada. Este objeto contém uma referência para um objeto `InetAddress` que especifica o endereço IP do computador *host*. O objeto fachada `URL` também faz referência a um objeto da classe `URLStreamHandler` que abre a conexão URL. O objeto cliente que usa o objeto fachada `URL` acessa os objetos `InetAddress` e `URLStreamHandler` através do objeto fachada. Entretanto, o objeto cliente não sabe como os objetos por trás do objeto fachada `URL` cumprem suas responsabilidades.

17.11.3 Padrões de arquitetura de projeto

Os padrões de projeto permitem que os desenvolvedores projetem partes específicas dos sistemas, como abstrair a instânciação de objetos ou agregar classes em grandes estruturas. Os padrões de projeto também incentivam o baixo acoplamento entre objetos. Os padrões de arquitetura incentivam o baixo acoplamento entre subsistemas. Estes padrões especificam todos os subsistemas de um sistema e como eles interagem uns com os outros¹. Apresentamos os populares padrões de arquitetura *Model-View-Controller* e *Layers*.

MVC

Imagine o projeto de um editor de texto simples. Neste programa, o usuário insere texto pelo teclado e formata este texto com o mouse. Nossa programa armazena este texto e as informações de formato em uma série de estruturas de dados, e depois exibe estas informações no vídeo para o usuário ler o que está sendo digitado.

Este programa adota o padrão de arquitetura *Model-View-Controller* (*MVC*), que separa os dados do aplicativo (contidos no *modelo*), dos componentes gráficos de apresentação (a *visão*) e da lógica de processamento de entrada (o *controlador*)². A Fig. 17.22 mostra os relacionamentos entre componentes no *MVC*.

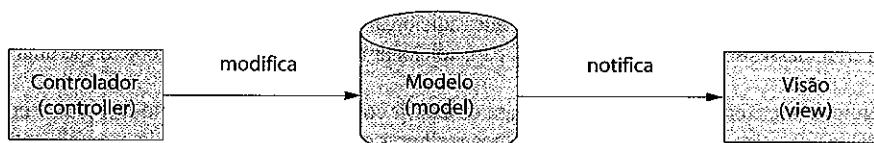


Fig. 17.22 Arquitetura *Model-View-Controller*.

O controlador implementa a lógica para processar as entradas do usuário. O modelo contém os dados do aplicativo, e a visão apresenta os dados armazenados no modelo. Quando o usuário fornece alguma entrada, o controlador modifica o modelo com a entrada dada. O modelo contém os dados do aplicativo. A respeito do exemplo do editor de texto, ele pode conter só os caracteres que compõem o documento. Quando o modelo se altera, ele notifica a visão da alteração para que ela possa atualizar sua apresentação com os dados alterados. A visão, em um processador de palavras, pode exibir caracteres usando uma determinada fonte, com um determinado tamanho, etc.

O *MVC* não restringe uma aplicação a uma única visão e um único controlador. Em um programa mais sofisticado (por exemplo, um processador de textos), podem existir duas visões de um modelo de documento. Uma visão pode exibir os tópicos do documento e a outra pode exibir o documento completo. O processador de texto também pode implementar vários controladores – um para tratar da entrada pelo teclado e outro para manipular a seleção pelo mouse. Se

¹ R. Hartman. "Building on Patterns". *Application Development Trends*. May 2001, p.19-26.

² A Seção 13.17 também discutiu a arquitetura *Model-View-Controller* e sua relevância no estudo de caso de simulação do elevador.

ambos os controladores fazem uma alteração no modelo, tanto a visão de tópicos quanto a janela de visualizar impressão vão mostrar a alteração imediatamente quando o modelo notificar todas as visões sobre as alterações.

Uma vantagem fundamental do padrão de arquitetura MVC é que os desenvolvedores podem modificar cada componente individualmente sem ter que modificar os outros componentes. Por exemplo, os desenvolvedores podem modificar a visão que exibe os tópicos do documento, mas os desenvolvedores não precisam modificar o modelo ou outras visões ou controladores.

Layers

Imagine o projeto da Fig. 17.23, que apresenta a estrutura básica de um *aplicativo de três camadas*, no qual cada camada contém um único componente do sistema.

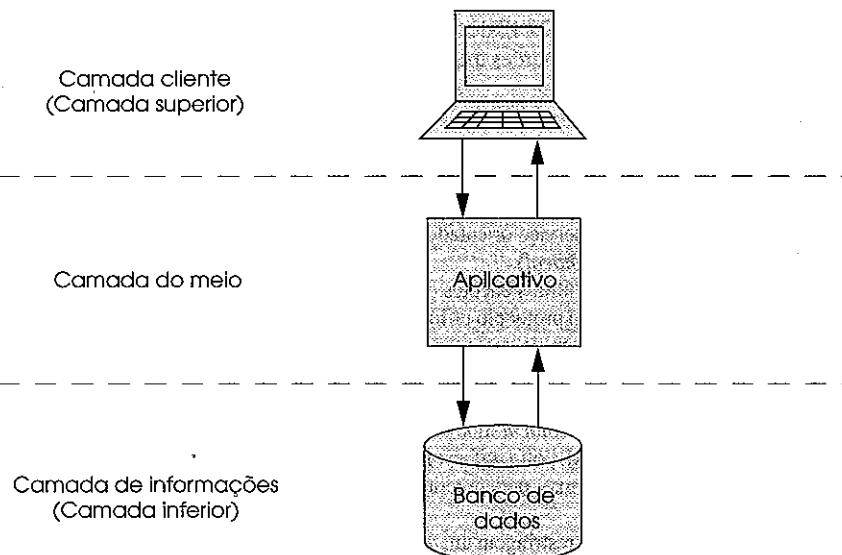


Fig. 17.23 Modelo de aplicativo em três camadas.

A *camada de informações* (também chamada de “camada inferior”) mantém os dados para o aplicativo geralmente armazenando estes dados em um banco de dados. A camada de informações para uma loja *on-line* pode conter informações sobre produtos, como descrições, preços e quantidades em estoque, e informações sobre clientes, como nomes de usuários, endereços de cobrança e números de cartões de crédito. A *camada do meio* atua como intermediária entre a camada de informações e a camada cliente. A camada do meio processa as requisições da camada cliente, lê dados do banco de dados e grava dados nele. A camada do meio, então, processa os dados da camada de informações e apresenta o conteúdo à camada cliente. Este processamento é a lógica de negócios do aplicativo, que realiza tarefas como recuperar dados da camada de informações, assegurar que os dados são confiáveis antes de atualizar o banco de dados, e apresentar dados para a camada cliente. Por exemplo, a lógica de negócios associada com a camada do meio, para a loja *on-line*, pode verificar o cartão de crédito de um cliente com o emissor do cartão de crédito antes que o almoxarifado despache o pedido do cliente. Esta lógica de negócios, então, pode armazenar (ou recuperar) as informações sobre crédito no banco de dados e notificar a camada cliente de que a verificação foi bem-sucedida.

A *camada cliente* (também chamada de “camada superior”) é a interface com o usuário do aplicativo, como um navegador da Web padrão. Os usuários interagem diretamente com o aplicativo através da interface com o usuário.

A camada cliente interage com a camada do meio para fazer requisições e recuperar dados da camada de informações. A camada cliente, então, exibe os dados obtidos da camada do meio.

A Fig. 17.23 é uma implementação do *padrão de arquitetura Layers*, o qual divide a funcionalidade em camadas separadas. Cada camada contém um conjunto de responsabilidades do sistema e só depende dos serviços da camada imediatamente abaixo. Na Fig. 17.23, cada camada corresponde a um nível. Este padrão de arquitetura é útil porque o projetista pode modificar um nível sem ter que modificar os outros níveis. Por exemplo, o projetista pode modificar a camada de informações da Fig. 17.23 para acomodar um produto de banco de dados em particular, mas o projetista não tem que modificar nem a camada cliente nem a camada do meio.

17.11.4 Conclusão

Nesta seção de “Descobrindo padrões de projetos”, discutimos como os pacotes `java.io` e `java.net` se beneficiam de padrões de projeto específicos e como os desenvolvedores podem integrar os padrões de projeto com aplicativos de redes/arquivos em Java. Também apresentamos os padrões de arquitetura *Model-View-Controller* e *Layers*, os quais atribuem funcionalidades do sistema a subsistemas separados. Esses padrões facilitam aos desenvolvedores o projeto de um sistema. Na Seção 21.12 de “Descobrindo padrões de projeto”, concluímos nossa apresentação de padrões de projeto discutindo os padrões usados no pacote `java.util`.

Resumo

- Java fornece soquetes de fluxo e soquetes de datagrama. Com soquetes de fluxo o processo estabelece uma conexão com outro processo. Enquanto a conexão está ativa, os dados fluem entre os processos em fluxos contínuos. Dizemos que os soquetes de fluxo fornecem um serviço orientado para conexão. O protocolo utilizado para transmissão é o conhecido TCP (*Transmission Control Protocol*).
- Com soquetes de datagrama, são transmitidos pacotes de informações. Esse não é o protocolo certo para os usuários rotineiros porque, ao contrário do TCP, o protocolo utilizado, UDP (*User Datagram Protocol*) é um serviço sem conexão e não garante que os pacotes cheguem de qualquer forma particular. Na verdade, os pacotes podem ser perdidos, duplicados e até chegar fora de seqüência. Então, com o UDP, é exigida do usuário uma programação extra significativa para lidar com esses problemas (se o usuário optar por fazer isso).
- O protocolo HTTP (*Hypertext Transfer Protocol*) que forma a base da World Wide Web utiliza URIs (*Uniform Resource Identifiers*, também chamados de URLs ou *Uniform Resource Locators*) para localizar dados na Internet. Os URIs comuns representam arquivos ou diretórios e podem representar tarefas complexas como pesquisas em banco de dados e pesquisas na Internet.
- Os navegadores da Web freqüentemente restringem um *applet* de modo que ele possa se comunicar somente com a máquina da qual ele foi originalmente baixado.
- Uma `Hashtable` armazena pares chave/valor. O programa usa uma chave para armazenar e recuperar um valor associado na `Hashtable`. O método `put` de `Hashtable` recebe dois argumentos – uma chave e seu valor associado – e coloca o valor na `Hashtable` em uma localização determinada pela chave. O método `get` de `Hashtable` recebe um argumento – uma chave – e recupera o valor (como uma referência a `Object`) associado com a chave.
- O `Vector` é um *array* de `Objects` dinamicamente redimensionável. O método `addElement` de `Vector` adiciona um novo elemento ao fim do `Vector`.
- O método `getAppletContext` de `Applet` devolve uma referência a um objeto `AppletContext` que representa o ambiente do *applet* (isto é, o navegador em que o *applet* está sendo executado).
- O método `showDocument` de `AppletContext` recebe um objeto `URL` como argumento e o passa para o `AppletContext` (isto é, o navegador), que exibe o recurso de World Wide Web associado com esse `URL`.
- Uma segunda versão do método `showDocument` de `AppletContext` permite a um *applet* especificar a *frame-alvo* em que o recurso da Web deve ser exibido. Entre as *frames-alvo* especiais incluem-se `_blank` (exibe o conteúdo do URI especificado em uma nova janela de navegador da Web), `_self` (exibe o conteúdo do URI especificado na mesma *frame* que o *applet*) e `_top` (o navegador deve remover as *frames* atuais e depois exibir o conteúdo do URI especificado na janela atual).
- O método `setCursor` de `Component` altera o cursor do mouse quando o cursor é posicionado sobre um componente GUI específico. O construtor `Cursor` recebe um inteiro que indica o tipo de cursor (como `Cursor.WAIT_CURSOR` ou `Cursor.DEFAULT_CURSOR`).
- O método `setPage` de `JEditorPane` baixa o documento especificado por seu argumento e o exibe no `JEditorPane`.

- Em geral, o documento HTML contém *hyperlinks* – texto, imagens ou componentes GUI que, ao serem clicados, levam a um outro documento na Web. Se um documento HTML é exibido em um **JEditorPane** e o usuário clica em um *hyperlink*, o **JEditorPane** gera um **HyperlinkEvent** (pacote `javax.swing.event`) e notifica todos os **HyperlinkListeners** registrados (pacote `javax.swing.event`) desse evento.
- O método `getEventType` de **HyperlinkEvent** determina o tipo do **HyperlinkEvent**. A classe **HyperlinkEvent** contém a classe `public static` interna **EventType**, que define três tipos de evento de *hyperlink*: **ACTIVATED** (o usuário clicou em um *hyperlink*), **ENTERED** (o usuário moveu o mouse sobre um *hyperlink*) e **EXITED** (o usuário moveu o mouse para fora de um *hyperlink*).
- O método `getURL` de **HyperlinkEvent** obtém o URL representado pelo *hyperlink*.
- As conexões baseadas em fluxos são gerenciadas com objetos **Socket**.
- O objeto **ServerSocket** estabelece a porta em que um servidor espera conexões de clientes. O segundo argumento para o construtor **ServerSocket** especifica o número de clientes que podem esperar uma conexão e ser processados pelo servidor. Se a fila de clientes estiver cheia, as conexões de clientes são recusadas. O método `accept` de **ServerSocket** espera indefinidamente (isto é, fica bloqueado) por uma conexão de um cliente e devolve um objeto **Socket** quando uma conexão é estabelecida.
- O método `getOutputStream` de **Socket** obtém uma referência para o **OutputStream** associado com um **Socket**. O método `getInputStream` de **Socket** obtém uma referência para o **InputStream** associado com o **Socket**.
- Quando a transmissão através de uma conexão de **Socket** está completa, o servidor fecha a conexão chamando o método `close` do **Socket**.
- O objeto **Socket** conecta um cliente a um servidor especificando o nome de servidor e o número da porta quando o objeto **Socket** é criado. Uma falha na tentativa de conexão dispara uma **IOException**.
- Quando o método `read` de **InputStream** devolve `-1`, o fluxo detecta que o fim de fluxo foi atingido.
- Ocorre uma **EOFException** quando um **ObjectInputStream** tenta ler o valor de um fluxo em que o fim de fluxo é detectado.
- O método `getByName` de **InetAddress** devolve um objeto **InetAddress** que contém o nome de *host* do computador para o qual o **String** nome de *host* ou o **String** endereço na Internet é especificado como argumento.
- O método `getLocalHost` de **InetAddress** devolve um objeto **InetAddress** que contém o nome de *host* do computador local que está executando o programa.
- A porta em que um cliente se conecta a um servidor às vezes é chamada de *ponto de handshake*.
- A transmissão orientada para conexão é como o sistema de telefonia – você disca e recebe uma *conexão* para o telefone da pessoa com quem você deseja se comunicar. A conexão é mantida enquanto durar a sua chamada telefônica, mesmo quando você não está conversando.
- A transmissão sem conexão com *datagramas* é semelhante ao envio de correspondência através do serviço postal. A mensagem grande que não cabe em um envelope pode ser dividida em pedaços de mensagem separados que são colocados em envelopes separados, numerados sequencialmente. Todas as cartas são remetidas de uma vez. As cartas podem chegar em ordem, fora da ordem ou não chegar.
- Os objetos **DatagramPacket** armazenam pacotes de dados a enviar ou armazenam pacotes de dados recebidos por um aplicativo. **DatagramSockets** enviam e recebem **DatagramPackets**.
- O construtor **DatagramSocket** que não recebe argumentos vincula o aplicativo a uma porta escolhida pelo computador no qual o programa é executado. O construtor **DatagramSocket** que recebe um argumento inteiro como número de porta vincula o aplicativo à porta especificada. Se um construtor **DatagramSocket** não consegue vincular o aplicativo a uma porta, ocorre uma **SocketException**.
- O método `receive` de **DatagramSocket** bloqueia (espera) até um pacote chegar, depois armazena o pacote em seu argumento.
- O método `getAddress` de **DatagramPacket** devolve um objeto **InetAddress** que contém informações sobre o computador *host* do qual o pacote foi enviado.
- O método `getPort` de **DatagramPacket** devolve um inteiro que especifica o número da porta através da qual o *host* enviou o **DatagramPacket**.
- O método `getLength` de **DatagramPacket** devolve um inteiro que representa o número de *bytes* de dados em um **DatagramPacket**.
- O método `getData` de **DatagramPacket** devolve um *array* de *bytes* que contém os dados em um **DatagramPacket**.
- O construtor **DatagramPacket** para um pacote a ser enviado recebe quatro argumentos – o *array* de *bytes* a ser enviado, o número de *bytes* a enviar, o endereço do cliente para o qual o pacote será enviado e o número da porta em que o cliente está esperando receber os pacotes.
- O método `send` de **DatagramSocket** envia um **DatagramPacket** para fora através da rede.

- Se ocorrer um erro ao receber ou enviar um **DatagramPacket**, ocorre uma **IOException**.
- A leitura de dados de um **Socket** é uma chamada bloqueante – a *thread* atual é colocada no estado bloqueada enquanto a *thread* espera que a operação de leitura seja completada. O método **setSoTimeout** especifica que, se nenhum dado for recebido após decorrido o número de milissegundos especificado, o **Socket** deve emitir uma **InterruptedException**, a qual pode ser capturada pela *thread* atual, e então continuar sendo executada. Isto evita que a *thread* atual entre em *deadlock* se não houver mais dados disponíveis do **Socket**.
- O *multicast* é uma maneira eficiente de enviar dados para muitos clientes sem a sobrecarga de transmitir aqueles dados para cada *host* na Internet.
- Usando *multicast*, um aplicativo pode “publicar” **DatagramPackets** para serem enviados para outros aplicativos – os “subscritores”.
- O aplicativo faz o *multicast* de **DatagramPackets** enviando os **DatagramPackets** para um endereço de *multicast* – um endereço IP no intervalo de 224.0.0.0 a 239.255.255.255, reservado para *multicast*.
- Os clientes que desejam receber **DatagramPackets** podem se conectar ao endereço de *multicast* para se associar ao grupo de *multicast* que irá receber os **DatagramPackets** publicados.
- **DatagramPackets** recebidos por *multicast* não são confiáveis – não há garantia de que os pacotes cheguem a qualquer destino. Além disso, a ordem na qual os clientes recebem os datagramas não é garantida.
- O construtor **MulticastSocket** recebe como argumento a porta à qual o **MulticastSocket** deve se conectar para receber **DatagramPackets** que estão chegando. O método **joinGroup** da classe **MulticastSocket** recebe como argumento o **InetAddress** do grupo de *multicast* ao qual se associar.
- O método **receive** da classe **MulticastSocket** lê um **DatagramPacket** que está chegando de um endereço de *multicast*.

Terminologia

<i>abrir um soquete</i>	<i>espera uma conexão</i>
<i>aceitar uma conexão</i>	<i>fechar uma conexão</i>
<i>classe BindException</i>	<i>grupo de multicast</i>
<i>classe ConnectException</i>	<i>host</i>
<i>classe Cursor</i>	<i>Hyperlink.EventType.ACTIVATED</i>
<i>classe DatagramPacket</i>	<i>Hyperlink.EventType.ENTERED</i>
<i>classe DatagramSocket</i>	<i>Hyperlink.EventType.EXITED</i>
<i>classe Hashtable</i>	<i>interface AppletContext</i>
<i>classe Hyperlink.EventType</i>	<i>interface HyperlinkListener</i>
<i>classe HyperlinkEvent</i>	<i>Internet</i>
<i>classe InetAddress</i>	<i>Java Security API</i>
<i>classe InterruptedIOException</i>	<i>ler de um soquete</i>
<i>classe IOException</i>	<i>método accept da classe ServerSocket</i>
<i>classe JEditorPane</i>	<i>método addElement da classe Vector</i>
<i>classe MalformedURLException</i>	<i>método close da classe Socket</i>
<i>classe MulticastSocket</i>	<i>método get da classe Hashtable</i>
<i>classe ServerSocket</i>	<i>método getAddress de DatagramPacket</i>
<i>classe Socket</i>	<i>método getAppletContext da classe Applet</i>
<i>classe URL</i>	<i>método getByName de InetAddress</i>
<i>classe Vector</i>	<i>método getData da classe DatagramPacket</i>
<i>cliente</i>	<i>método getEventType</i>
<i>cliente se conecta a um servidor</i>	<i>método getInputStream da classe Socket</i>
<i>comprimento de pacote</i>	<i>método getLength de DatagramPacket</i>
<i>computação colaboradora</i>	<i>método getLocalHost</i>
<i>comunicação baseada em soquetes</i>	<i>método getLocalHost de InetAddress</i>
<i>conectar-se a um site da World Wide Web</i>	<i>método getOutputStream da classe Socket</i>
<i>conectar-se a uma porta</i>	<i>método getPort da classe DatagramPacket</i>
<i>conexão</i>	<i>método getPredefinedCursor de Cursor</i>
<i>Cursor.DEFAULT_CURSOR</i>	<i>método getURL da classe HyperlinkEvent</i>
<i>Cursor.WAIT_CURSOR</i>	<i>método hyperlinkUpdate</i>
<i>datagrama</i>	<i>método joinGroup da classe MulticastSocket</i>
<i>endereço de multicast</i>	<i>método leaveGroup da classe MulticastSocket</i>
<i>endereço na Internet</i>	<i>método put da classe Hashtable</i>
<i>espera um pacote</i>	<i>método receive da classe DatagramSocket</i>

método <code>send</code> da classe <code>DatagramSocket</code>	serviço orientado para conexão
método <code>setCursor</code> da classe <code>Component</code>	serviço sem conexão
método <code>setPage</code> da classe <code>JEditorPane</code>	servidor
método <code>setSoTimeout</code> da classe <code>Socket</code>	servidor com múltiplas threads
método <code>showDocument</code> de <code>AppletContext</code>	servidor da Web
<i>multicast</i>	sistemas heterogêneos de computador
<i>navegador da Web</i>	<code>SocketException</code>
<i>número de porta em um servidor</i>	solicitação de conexão
<i>pacote</i>	soquete
<i>pacote java.net</i>	soquete de datagrama
<i>pacotes duplicados</i>	soquete de fluxo
<i>pacotes fora de seqüência</i>	soquete do lado do cliente
<i>pacotes perdidos</i>	soquete do lado do servidor
<i>par chave/valor</i>	TCP (Transmission Control Protocol)
<i>ponto de handshake</i>	transmissão sem conexão com datagramas
<i>porta</i>	UDP (User Datagram Protocol)
<i>programação de redes</i>	<code>UnknownHostException</code>
<i>recusar uma conexão</i>	URI (Uniform Resource Identifier)
<i>redes</i>	URL (Uniform Resource Locator)
<i>redes de computadores</i>	vincular a uma porta
<i>registrar um número de porta disponível</i>	World Wide Web
<i>relacionamento cliente/servidor</i>	

Exercícios de auto-revisão

- 17.1** Preencha as lacunas em cada uma das frases seguintes:
- A exceção _____ ocorre quando há um erro de entrada/saída ao se fechar um soquete.
 - A exceção _____ ocorre quando um endereço de servidor indicado por um cliente não pode ser resolvido.
 - Se um construtor `DatagramSocket` não consegue configurar um `DatagramSocket` adequadamente, ocorre uma exceção do tipo _____.
 - Muitas das classes de rede de Java estão contidas no pacote _____.
 - A classe _____ vincula o aplicativo a uma porta para transmissão de datagramas.
 - Um objeto da classe _____ contém um endereço da Internet.
 - Os dois tipos de soquetes que discutimos neste capítulo são soquetes _____ e soquetes _____.
 - O acrônimo URL significa _____.
 - O acrônimo URI significa _____.
 - O protocolo fundamental que forma a base da World Wide Web é _____.
 - O método _____ de `AppletContext` recebe um objeto URL como argumento e exibe em um navegador o recurso da World Wide Web associado com esse URL.
 - O método `getLocalHost` de `InetAddress` devolve um objeto _____ que contém o nome de host local do computador em que o programa está sendo executado.
 - O método _____ da classe `MulticastSocket` inscreve o `MulticastSocket` em um grupo de *multicast*.
 - O construtor URL determina se o `String` passado como argumento representa um *Uniform Resource Identifier* válido. Se for, o objeto URL é inicializado para conter o URI; caso contrário, ocorre uma exceção do tipo _____.
- 17.2** Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- O aplicativo que usa *multicast* transmite `DatagramPackets` para todos os *hosts* na Internet.
 - O UDP é um protocolo orientado para conexão.
 - Com soquetes de fluxo, o processo estabelece uma conexão com outro processo.
 - O servidor espera em uma porta por conexões de um cliente.
 - A transmissão de pacotes de datagramas através de uma rede é confiável – garante-se que os pacotes chegarão na seqüência.
 - Por razões de segurança, muitos navegadores da Web como o Netscape Communicator permitem que os *applets* Java façam processamento de arquivos apenas nas máquinas em que são executados.

- g) Os navegadores da Web freqüentemente restringem um *applet* para que ele possa apenas se comunicar com a máquina da qual ele foi baixado originalmente.
- h) Os endereços IP no intervalo de 224.0.0.0 a 239.255.255.255 são reservados para *multicast*.

Respostas aos exercícios de auto-revisão

17.1 a) `IOException`. b) `UnknownHostException`. c) `SocketException`. d) `java.net`. e) `DatagramSocket`. f) `InetAddress`. g) fluxo, datagrama. h) Uniform Resource Locator. i) *Universal Resource Identifier*. j) `http`. k) `showDocument`. l) `InetAddress`. m) `joinGroup`. n) `MalformedURLException`.

17.2 a) Falsa; o *multicast* envia `DatagramPackets` somente para *hosts* que tenham se associado ao grupo de *multicast*. b) Falsa; o UDP é um protocolo sem conexão e o TCP é um protocolo orientado para conexão. c) Verdadeira. d) Verdadeira. e) Falsa; os pacotes podem ser perdidos e os pacotes podem chegar fora de ordem. f) Falsa; a maioria dos navegadores impede que os *applets* façam processamento de arquivos na máquina do cliente. g) Verdadeira. h) Verdadeira.

Exercícios

- 17.3** Estabeleça a distinção entre serviços de rede orientados para conexão e sem conexão.
- 17.4** Como um cliente determina o nome de *host* do computador cliente?
- 17.5** Sob quais circunstâncias uma `SocketException` seria disparada?
- 17.6** Como um cliente pode obter uma linha de texto de um servidor?
- 17.7** Descreva como um cliente se conecta a um servidor.
- 17.8** Descreva como um servidor envia dados para um cliente.
- 17.9** Descreva como preparar um servidor para receber uma solicitação de conexão com base em fluxos de um único cliente.
- 17.10** Descreva como preparar um servidor para receber solicitações de conexão de múltiplos clientes se cada cliente que se conecta deve ser processado em paralelo com todos os outros clientes conectados.
- 17.11** Como um servidor espera por conexões em uma porta?
- 17.12** O que determina quantas solicitações de conexão de clientes podem esperar em uma fila para se conectar a um servidor?
- 17.13** Como descrito no texto, que razões podem fazer com que um servidor recuse uma solicitação de conexão de um cliente?
- 17.14** Utilize uma conexão de soquetes para permitir que um cliente especifique um nome de arquivo e fazer o servidor enviar o conteúdo do arquivo ou indicar que o arquivo não existe.
- 17.15** Modifique o Exercício 17.14 para permitir que o cliente modifique o conteúdo do arquivo e envie o arquivo de volta ao servidor para armazenamento. O usuário pode editar o arquivo em uma `JTextArea` e, depois, clicar no botão *salvar alterações* para enviar o arquivo de volta para o servidor.
- 17.16** Modifique o programa da Fig. 17.2 para permitir aos usuários adicionar seus próprios *sites* à lista e removê-los dela.
- 17.17** Os servidores com múltiplas *threads* são bem conhecidos hoje, especialmente por causa da utilização crescente de servidores com *multiprocessing*. Modifique o aplicativo servidor simples apresentado na Seção 17.6 para ser um servidor com múltiplas *threads*. Depois utilize vários aplicativos clientes e faça cada um deles se conectar ao servidor simultaneamente. Use um `Vector` para armazenar as *threads* clientes. `Vector` oferece diversos métodos que podem ser úteis neste exercício. O método `size` determina a quantidade de elementos em um `Vector`. O método `elementAt` devolve o elemento na posição especificada (como uma referência para `Object`). O método `add` coloca um novo elemento no fim do `Vector`. O método `remove` apaga seu argumento do `Vector`. O método `lastElement` devolve uma referência para `Object` para o último objeto que você inseriu no `Vector`.
- 17.18** No texto, apresentamos um programa de jogo-da-velha controlado por um servidor com múltiplas *threads*. Desenvolva um programa de jogo de damas modelado com base no programa de jogo-da-velha. Os dois usuários devem fazer jogadas alternadamente. O programa deve mediar as jogadas dos jogadores, determinando de quem é a vez e permitindo apenas movimentos válidos. Os próprios jogadores determinarão quando o jogo acabou.

- 17.19** Desenvolva um programa de jogo de xadrez modelado com base no programa de damas do Exercício 17.18.
- 17.20** Desenvolva um programa de jogo de cartas “vinte e um” em que o aplicativo servidor dá as cartas para cada um dos *applets* clientes. O servidor deve distribuir cartas adicionais (de acordo com as regras do jogo) para cada jogador quando solicitado.
- 17.21** Desenvolva um jogo de pôquer em que o aplicativo servidor dá as cartas para cada um dos *applets* clientes. O servidor deve distribuir cartas adicionais (de acordo com as regras do jogo) para cada jogador quando solicitado.
- 17.22** (*Modificações no Programa Tic-Tac-Toe com múltiplas threads*) Os programas das Figs. 17.8 e 17.9 implementaram uma versão cliente/servidor com múltiplas *threads* do jogo-da-velha. Nossa objetivo ao desenvolver esse jogo foi demonstrar um servidor com múltiplas *threads* que pode processar múltiplas conexões de clientes ao mesmo tempo. O servidor no exemplo é na realidade um mediador entre os dois *applets* clientes – ele certifica-se de que cada jogada é válida e que cada cliente jogue na ordem adequada. O servidor não determina quem ganhou ou quem perdeu nem se houve um empate. Além disso, não há opções para permitir que um novo jogo seja jogado ou para terminar um jogo em andamento.

A seguir há uma lista de modificações sugeridas para o aplicativo e o *applet* com múltiplas *threads* do jogo-da-velha.

- Modifique a classe **TicTacToeServer** para testar se há um vencedor, um perdedor ou um empate em cada jogada durante o jogo. Envie uma mensagem para cada *applet* cliente indicando o resultado do jogo quando ele acabar.
- Modifique a classe **TicTacToeClient** para exibir um botão que, ao ser clicado, permita ao cliente jogar outro jogo. O botão deve apenas ser habilitado quando um jogo terminar. Observe que tanto a classe **TicTacToeClient** como a classe **TicTacToeServer** devem ser modificadas para restaurar o tabuleiro e todas as informações de estado. Além disso, o outro **TicTacToeClient** deve ser notificado de que um novo jogo está para iniciar, de modo que o tabuleiro e o estado possam ser restaurados.
- Modifique a classe **TicTacToeClient** para fornecer um botão que permita que um cliente termine o programa a qualquer momento. Quando o usuário clica no botão, o servidor e o outro cliente devem ser notificados. O servidor então deve esperar uma conexão de um outro cliente de modo que um novo jogo possa começar.
- Modifique a classe **TicTacToeClient** e a classe **TicTacToeServer** de modo que o vencedor de um jogo possa escolher a marca X ou 0 para o próximo jogo. *Lembre-se: X sempre começa primeiro.*
- Se você quiser ser ambicioso, permita que um cliente jogue contra o servidor enquanto o servidor espera por uma conexão de outro cliente.

- 17.23** (*Jogo-da-velha 3-D com múltiplas threads*) Modifique o programa cliente/servidor do jogo-da-velha com múltiplas *threads* para implementar uma versão tridimensional 4 por 4 por 4 do jogo. Implemente o aplicativo servidor para mediar entre os dois clientes. Exiba o tabuleiro tridimensional como quatro tabuleiros que contêm quatro linhas e quatro colunas cada um. Se você quiser ser ambicioso, experimente as seguintes modificações:
- Desenhe o tabuleiro de uma maneira tridimensional.
 - Permita ao servidor testar se há um ganhador, um perdedor ou um empate. Cuidado! Há muitas maneiras possíveis de ganhar em um tabuleiro 4 por 4 por 4!

- 17.24** (*Código Morse em rede*) Modifique a solução do Exercício 10.27 para permitir que dois *applets* troquem mensagens em Código Morse entre si através de um aplicativo servidor com múltiplas *threads*. Cada *applet* deve permitir que o usuário digite caracteres normais em **JTextAreas**, traduza os caracteres para Código Morse e envie a mensagem codificada, através do servidor, para o outro cliente. Quando as mensagens são recebidas, elas devem ser decodificadas e exibidas como caracteres normais e como Código Morse. O *applet* deve ter duas **JTextAreas**: uma para exibir as mensagens do outro cliente e uma para digitação.

Multimídia: imagens, animação, áudio e vídeo

Objetivos

- Entender como obter e exibir imagens.
- Criar animações a partir de sequências de imagens.
- Personalizar um *applet* de animação com parâmetros de *applet* especificados no documento HTML do *applet*.
- Criar mapas de imagem.
- Ser capaz de obter, reproduzir, repetir e parar sons com um **AudioClip**.

A roda que faz mais barulho ... é a que obtém a graxa.

John Billings (Henry Wheeler Shaw)

Utilizaremos um sinal que eu testei e considerei de grande alcance e fácil de gritar. Uaa-huu!

Zane Grey

Há uma dança natural no movimento de um peixe de aquário.

Walt Disney

Entre o movimento e o ato está a sombra.

Thomas Stearns Eliot, *The Hollow Men*



Sumário do capítulo

- 18.1 Introdução**
- 18.2 Carregando, exibindo e dimensionando imagens**
- 18.3 Animando uma série de imagens**
- 18.4 Personalizando o *Logo Animator* através de parâmetros de applet**
- 18.5 Mapas de imagens**
- 18.6 Carregando e reproduzindo clipes de áudio**
- 18.7 Recursos na Internet e na World Wide Web**

[Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão]

[Exercícios]

18.1 Introdução

Bem-vindo ao que pode ser a maior revolução na história da computação. Os que entraram neste campo décadas atrás estavam principalmente interessados em utilizar os computadores para fazer cálculos aritméticos em alta velocidade. Mas com o desenvolvimento do campo da informática, estamos começando a perceber que a capacidade de manipulação de dados dos computadores é agora igualmente importante. A “nova onda” de Java é a *multimídia*, o uso de *som*, *imagens*, *imagens gráficas* e *vídeo* para fazer os aplicativos “ganharem vida”. Atualmente muitas pessoas consideram o monitor bidimensional colorido a “última palavra” em multimídia. Mas nesta década, esperamos o advento de todo tipo de aplicativos tridimensionais sofisticados. Os programadores Java já podem usar a *Java3D API* para criar aplicativos gráficos substanciais. Discutimos a Java3D API em nosso livro *Advanced Java 2 Platform How to Program*.

A programação com multimídia oferece muitos desafios novos. A área já é enorme e crescerá rapidamente. As pessoas estão se apressando em equipar seus computadores para multimídia. A maioria dos computadores novos vendidos hoje em dia já estão “prontos para multimídia”, incluindo unidades de CD ou DVD, placas de áudio e às vezes recursos especiais de vídeo.

Entre os usuários fãs dos recursos gráficos, as imagens gráficas bidimensionais não são mais suficientes. Agora muitas pessoas querem gráficos coloridos tridimensionais de alta resolução. Imagens verdadeiramente tridimensionais podem tornar-se disponíveis dentro da próxima década. Imagine ter televisão tridimensional de ultra-alta-resolução do tipo “*home theater*”. Eventos esportivos e espetáculos parecerão acontecer ao vivo na sua própria sala de estar! Estudantes de medicina em todo o mundo verão operações sendo realizadas a milhares de quilômetros de distância como se estivessem ocorrendo na mesma sala. As pessoas serão capazes de aprender como dirigir em suas casas, com simuladores de direção extremamente realistas, antes de pegarem no volante. As possibilidades são empolgantes e intermináveis.

A multimídia exige um extraordinário poder de computação. Até recentemente, os computadores economicamente acessíveis com esse tipo de poder não estavam disponíveis. Os processadores ultra-rápidos atuais, como o SPARC Ultra da Sun Microsystems, o Pentium e o Itanium da Intel, o Alfa da Compaq Computer Corporation e o R8000 da MIPS/Silicon Graphics (entre outros) possibilitaram a multimídia eficiente. O computador e os setores de comunicações serão os principais beneficiários da revolução multimídia. Os usuários estarão dispostos a pagar por processadores mais rápidos, mais memória e mais largura de banda para as comunicações que suportam aplicativos multimídia exigentes. Ironicamente, os usuários podem não precisar pagar mais caro à medida que a competição feroz nesses setores força a baixa dos preços.

Precisamos de linguagens de programação que facilitem a criação de aplicativos multimídia. A maioria das linguagens de programação não tem capacidade de multimídia incorporada. Mas Java fornece extensos recursos para multimídia que permitem desenvolver imediatamente aplicativos de multimídia poderosos.

Este capítulo apresenta uma série de exemplos de “código ativo” que abrangem diversos recursos interessantes de multimídia de que você vai necessitar para construir aplicativos úteis, incluindo:

1. princípios básicos da manipulação de imagens

2. criação de animações suaves
3. personalização de um *applet* animado através de parâmetros fornecidos a partir do arquivo de HTML que invoca o *applet*
4. reprodução de arquivos de áudio com a interface **AudioClip**
5. criação de mapas de imagem capazes de detectar quando o cursor está sobre eles mesmo sem o clicar do mouse

Continuaremos nossa abordagem dos recursos de multimídia de Java no Capítulo 22, no qual discutiremos as APIs *Java Media Framework (JMF)* e *Java Sound*. A JMF e o Java Sound permitem que os programas em Java reproduzam e gravem áudio e vídeo. A JMF também permite enviar fluxos de áudio e vídeo – denominados *streaming media* – através de uma rede ou pela Internet. Os exercícios deste capítulo e do Capítulo 22 sugerem dezenas de projetos desafiadores e interessantes e até mencionam algumas “idéias de milhões de dólares” que podem ajudá-lo a construir sua fortuna! Quando estávamos criando esses exercícios, parecia que as idéias simplesmente fluíam sem parar. A multimídia parece estimular a criatividade de maneiras que não experimentamos com a capacidade dos computadores “convencionais”.

18.2 Carregando, exibindo e dimensionando imagens

Os recursos de multimídia de Java incluem gráficos, imagens, animações, sons e vídeo. Iniciamos nossa discussão de multimídia com imagens.

O *applet* da Fig. 18.1 demonstra o carregamento de uma **Image** (pacote `java.awt`) e o carregamento de um **ImageIcon** (pacote `javax.swing`). O *applet* exibe a **Image** em seu tamanho original e ampliada para um tamanho maior utilizando duas versões do método **drawImage** de **Graphics**. O *applet* também desenha o **ImageIcon** com o método **paintIcon**. A classe **ImageIcon** é mais fácil de usar do que **Image**, porque seu construtor pode receber argumentos de vários formatos diferentes, incluindo um *array byte* que contém os *bytes* de uma imagem, uma **Image** já carregada na memória, um **String** representando a posição de uma imagem e o URL representando a localização de uma imagem.

```

1 // Fig. 18.1: LoadImageAndScale.java
2 // Carrega uma imagem e a exibe em seu tamanho original e a redimensiona
3 // para duas vezes a largura e altura originais.
4 // Carrega e exibe a mesma imagem como um ImageIcon.
5
6 // Pacotes do núcleo de Java
7 import java.applet.Applet;
8 import java.awt.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 public class LoadImageAndScale extends JApplet {
14     private Image logo1;
15     private ImageIcon logo2;
16
17     // carrega a imagem quando o applet é carregado
18     public void init()
19     {
20         logo1 = getImage( getDocumentBase(), "logo.gif" );
21         logo2 = new ImageIcon( "logo.gif" );
22     }
23
24     // exibe a imagem
25     public void paint( Graphics g )

```

Fig. 18.1 Carregando e exibindo uma imagem em um *applet* (parte 1 de 2).

```

26      {
27          // desenha a imagem original
28          g.drawImage( logo1, 0, 0, this );
29
30          // desenha a imagem redimensionada para caber na largura
31          // do applet e altura do applet menos 120 pixels
32          g.drawImage( logo1, 0, 120,
33                      getWidth(), getHeight() - 120, this );
34
35          // desenha ícone usando seu método paintIcon
36          logo2.paintIcon( this, g, 180, 0 );
37      }
38
39 } // fim da classe LoadImageAndScale

```



Fig. 18.1 Carregando e exibindo uma imagem em um *applet* (parte 2 de 2).

As linhas 14 e 15 declaram uma referência `Image` e uma referência `ImageIcon`, respectivamente. A classe `Image` é uma classe `abstract`; portanto, você não pode criar um objeto da classe `Image` diretamente. Em vez disso, o *applet* deve chamar um método que faz com que o contêiner de *applets* carregue e devolva a `Image` para uso no programa. A classe `Applet` (a superclasse de `JApplet`) fornece um método que faz justamente isso. A linha 20 no método `init` do *applet* utiliza o método `getImage` de `Applet` para carregar uma `Image` no *applet*. Essa versão de `getImage` recebe dois argumentos – uma localização em que a imagem está armazenada e o nome do arquivo que contém a imagem. No primeiro argumento, o método `getDocumentBase` de `Applet` devolve o URL representando a localização da imagem na Internet (ou em seu computador, se o *applet* foi carregado a partir do seu computador). O programa supõe que a imagem está armazenada no mesmo diretório que o arquivo HTML que invocou o *applet*. O método `getDocumentBase` devolve a localização do arquivo HTML na Internet como objeto da classe `URL`. O segundo argumento especifica um nome de arquivo de imagem. Java suporta vários formatos de imagem, incluindo *Graphics Interchange Format (GIF)*, *Joint Photographic Experts Group (JPEG)* e *Portable Network Graphics (PNG)*. Os nomes de arquivo para cada um desses tipos terminam com `.gif`, `.jpg` (ou `.jpeg`) e `.png`, respectivamente.



Dica de portabilidade 18.1

A classe `Image` é uma classe `abstract`, assim os objetos `Image` não podem ser criados diretamente. Para obter independência de plataforma, a implementação de Java em cada plataforma fornece sua própria subclasse de `Image` para armazenar as informações da imagem.

A linha 20 invoca o método `getImage` para configurar a carga da imagem a partir do computador local (ou baixar a imagem da Internet). Quando a imagem é exigida pelo programa, a imagem é carregada em uma *thread* de execução separada. Isso permite que o programa continue a execução enquanto a imagem estiver sendo carregada. [Nota: se o arquivo solicitado não estiver disponível, o método `getImage` não indica um erro.]

A classe `ImageIcon` não é uma classe `abstract`; portanto, ela pode criar um objeto `ImageIcon`. A linha 21 no método `init` do *applet* cria um objeto `ImageIcon` que carrega a mesma imagem `logo.gif`. A classe `ImageIcon` fornece vários construtores que permitem que os programas inicializem objetos `ImageIcon` com imagens do computador local ou com imagens armazenadas na Internet.

O método `paint` (linhas 25 a 37) do *applet* exibe as imagens. A linha 28 utiliza o método `drawImage` de `Graphics` para exibir uma `Image`. O método `drawImage` recebe quatro argumentos. O primeiro argumento é uma referência ao objeto `Image` a ser exibido (`logo1`). O segundo e terceiro argumentos são as coordenadas *x* e *y* em que a imagem deve ser exibida no *applet*; as coordenadas indicam o canto superior esquerdo da imagem. O último argumento é uma referência a um objeto `ImageObserver`. Normalmente, o `ImageObserver` é o objeto no qual o programa exibe a imagem. O `ImageObserver` pode ser qualquer objeto que implementa a interface `ImageObserver`. A interface `ImageObserver` é implementada pela classe `Component` (uma das superclasses indiretas da classe `Applet`). Portanto, todo os `Components` (incluindo nosso *applet*) são `ImageObservers`. O argumento `ImageObserver` é importante ao exibir imagens grandes que exigem um longo tempo para baixar da Internet. É possível que um programa execute o código que exibe a imagem antes de a imagem ser completamente baixada. O `ImageObserver` é notificado para atualizar a imagem exibida à medida que o restante da imagem é carregado. Quando você executar esse *applet*, observe atentamente como partes da imagem são exibidas enquanto a imagem é carregada. [Nota: em computadores mais rápidos, você pode não perceber esse efeito.]

As linhas 32 e 33 utilizam outra versão do método `drawImage` de `Graphics` para dar saída a uma versão *ampliada* da imagem. O quarto e quinto argumentos especificam a *largura* e a *altura* da imagem para fins de exibição. O método `drawImage` redimensiona a imagem para se ajustar à largura e à altura especificadas. Neste exemplo, o quarto argumento indica que a largura da imagem redimensionada deve ser a largura do *applet* e o quinto argumento indica que a altura deve ser a altura do *applet* menos 120 pixels. A linha 33 determina a largura e a altura do *applet* chamando os métodos `getWidth` e `getHeight` (herdados da classe `Component`).

A linha 36 utiliza o método `paintIcon` de `ImageIcon` para exibir a imagem. O método exige quatro argumentos – uma referência ao `Component` em que a imagem será exibida, uma referência ao objeto `Graphics` que será utilizado para exibir a imagem, a coordenada *x* do canto superior esquerdo da imagem e a coordenada *y* do canto superior esquerdo da imagem.

Se você comparar as duas técnicas para carga e exibição de imagens nesse exemplo, poderá ver que utilizar `ImageIcon` é mais simples. Você cria objetos da classe `ImageIcon` diretamente e não precisa utilizar uma referência `ImageObserver` quando exibir a imagem. Por essa razão, utilizamos a classe `ImageIcon` no restante do capítulo. [Nota: o método `paintIcon` da classe `ImageIcon` não permite o redimensionamento de uma imagem. Entretanto, a classe fornece o método `getImage` que retorna uma referência `Image` que pode ser utilizada com o método `drawImage` de `Graphics` para exibir uma imagem redimensionada.]

18.3 Animando uma série de imagens

O próximo exemplo demonstra a animação de uma série de imagens que estão armazenadas em um *array*. O aplicativo utiliza as mesmas técnicas para carregar e exibir `ImageIcons` mostradas na Fig. 18.1. A animação apresentada na Fig. 18.2 é projetada como uma subclasse de `JPanel` (chamada `LogoAnimator`) que pode ser anexada a uma janela de aplicativo ou possivelmente a um `JApplet`. A classe `LogoAnimator` também define um método `main` (definido nas linhas 96 a 117) para executar a animação como aplicativo. O método `main` define uma instância da classe `JFrame` e anexa um objeto `LogoAnimator` ao `JFrame` para exibir a animação.

```

1 // Fig. 18.2: LogoAnimator.java
2 // Animação de uma série de imagens
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;

```

Fig. 18.2 Animando uma série de imagens (parte 1 de 3).

```

6  import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class LogoAnimator extends JPanel
12     implements ActionListener {
13
14     protected ImageIcon images[];      // array de imagens
15
16     protected int totalImages = 30,    // quantidade de imagens
17                     currentImage = 0,   // índice da imagem corrente
18                     animationDelay = 50, // retardo em milissegundos
19                     width,           // largura da imagem
20                     height;          // altura da imagem
21
22     protected String imageName = "deitel"; // nome básico das imagens
23     protected Timer animationTimer; // Timer que faz funcionar a animação
24
25     // inicializa LogoAnimator carregando as imagens
26     public LogoAnimator()
27     {
28         initializeAnimation();
29     }
30
31     // inicializa a animação
32     protected void initializeAnimation()
33     {
34         images = new ImageIcon[ totalImages ];
35
36         // carrega as imagens
37         for ( int count = 0; count < images.length; ++count )
38             images[ count ] = new ImageIcon( getClass().getResource(
39                 "images/" + imageName + count + ".gif" ) );
40
41         width = images[ 0 ].getIconWidth(); // obtém a largura do ícone
42         height = images[ 0 ].getIconHeight(); // obtém a altura do ícone
43     }
44
45     // exibe a imagem atual
46     public void paintComponent( Graphics g )
47     {
48         super.paintComponent( g );
49
50         images[ currentImage ].paintIcon( this, g, 0, 0 );
51         currentImage = ( currentImage + 1 ) % totalImages;
52     }
53
54     // responde a evento do Timer
55     public void actionPerformed( ActionEvent actionEvent )
56     {
57         repaint(); // redesenha o animador
58     }
59
60     // inicia ou reinicia a animação
61     public void startAnimation()
62     {
63         if ( animationTimer == null ) {
64             currentImage = 0;
65             animationTimer = new Timer( animationDelay, this );

```

Fig. 18.2 Animando uma série de imagens (parte 2 de 3).

```

66         animationTimer.start();
67     }
68     else // continua a partir da última imagem exibida
69     if ( ! animationTimer.isRunning() )
70         animationTimer.restart();
71 }
72
73 // faz parar o timer da animação
74 public void stopAnimation()
75 {
76     animationTimer.stop();
77 }
78
79 // devolve o tamanho mínimo da animação
80 public Dimension getMinimumSize()
81 {
82     return getPreferredSize();
83 }
84
85 // devolve o tamanho preferido da animação
86 public Dimension getPreferredSize()
87 {
88     return new Dimension( width, height );
89 }
90
91 // executa a animação em um JFrame
92 public static void main( String args[] )
93 {
94     // cria LogoAnimator
95     LogoAnimator animation = new LogoAnimator();
96
97     // configura a janela
98     JFrame window = new JFrame( "Animator test" );
99
100    Container container = window.getContentPane();
101    container.add( animation );
102
103    window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
104
105    // dimensiona e exibe a janela
106    window.pack();
107    Insets insets = window.getInsets();
108
109    window.setSize( animation.getPreferredSize().width +
110        insets.left + insets.right,
111        animation.getPreferredSize().height +
112        insets.top + insets.bottom );
113
114    window.setVisible( true );
115    animation.startAnimation(); // inicia a animação
116
117 } // fim do método main
118
119 } // fim da classe LogoAnimator

```



Fig. 18.2 Animando uma série de imagens (parte 3 de 3).

A classe `LogoAnimator` mantém um *array* de `ImageIcons` que são carregados no método `initializeAnimation` (linhas 32 a 43), o qual é chamado a partir do construtor. À medida que a estrutura `for` (linhas 37 a 39) cria cada objeto `ImageIcon`, o construtor `ImageIcon` carrega uma das 30 imagens da animação. O argumento do construtor utiliza a concatenação de *strings* para montar o nome de arquivo a partir das partes "`images/`", `image-Name`, `count` e `".gif"`. Cada uma das imagens na animação está em um dos arquivos denominados `deitel#.gif`, onde `#` é um valor no intervalo de 0 a 29, especificado pela variável `count` de controle do laço. As linhas 41 e 42 determinam a largura e a altura da animação a partir do tamanho da primeira imagem no *array* `images`.

Dica de desempenho 18.1



É mais eficiente carregar os frames da animação como uma única imagem do que carregar cada imagem separadamente (pode-se usar um programa de pintura para combinar os frames da animação em uma imagem). Se as imagens estão sendo carregadas da Web, cada imagem carregada exige uma conexão separada com o site em que as imagens são armazenadas.

Dica de desempenho 18.2



Carregar todas as frames de uma animação como uma imagem grande pode forçar seu programa a esperar um tempo para começar a exibir a animação.

Depois que o construtor `LogoAnimator` carrega as imagens, o método `main` configura a janela na qual a animação irá aparecer e chama `startAnimation` (definido nas linhas 61 a 71) para começar a animação. A animação é controlada por uma instância da classe `Timer` (pacote `javax.swing`). Um objeto da classe `Timer` gera `ActionEvents` a um intervalo fixo em milissegundos (normalmente especificado como argumento para o construtor de `Timer`) e notifica todos os seus `ActionListeners` registrados de que o evento ocorreu. As linhas 63 a 67 determinam se a referência a `Timer animationTimer` é `null`. Se for, a linha 64 configura `currentImage` como 0 para indicar que a animação deve iniciar com a imagem no primeiro elemento do *array* `images`. A linha 65 atribui um novo objeto `Timer` a `animationTimer`. O construtor `Timer` recebe dois argumentos – o retardo em milissegundos (`animationDelay` é 50 nesse exemplo) e o `ActionListener` que responderá aos `ActionEvents` do `Timer`. A classe `LogoAnimator` implementa `ActionListener`, por isso, a linha 65 especifica `this` como `Listener`. A linha 66 inicia o objeto `Timer`. Uma vez iniciado, `animationTimer` gerará um `ActionEvent` a cada 50 milissegundos. As linhas 69 e 70 permitem que um programa reinicie uma animação que o programa interrompeu anteriormente. Por exemplo, para fazer uma animação “amigável para o navegador” em um *applet*, a animação deve ser parada quando o usuário mudar de página da Web. Se o usuário voltar à página da Web que contém a animação, o método `startAnimation` pode ser chamado para reiniciar a animação. A condição `if` na linha 73 utiliza o método `isRunning` de `Timer` para determinar se o `Timer` está sendo executado (isto é, gerando eventos). Se não estiver sendo executado, a linha 70 chama o método `restart` de `Timer` para indicar que `Timer` deve começar a gerar eventos novamente.

Em resposta a cada evento de `Timer` nesse exemplo, o programa chama o método `actionPerformed` (linhas 55 a 58). A linha 57 chama o método `repaint` de `LogoAnimator` para escalonar uma chamada ao método `update` do `LogoAnimator` (herdado da classe `JPanel`) que, por sua vez, chama o método `paintComponent` de `LogoAnimator` (linhas 46 a 52). Lembre-se de que qualquer subclasse de `JComponent` que faz desenhos deve fazer isso em seu método `paintComponent`. Como mencionado no Capítulo 13, a primeira instrução em qualquer método `paintComponent` deve ser uma chamada ao método `paintComponent` da superclasse para assegurar que os componentes Swing sejam exibidos corretamente.

As linhas 50 e 51 pintam o `ImageIcon` do elemento `currentImage` do *array* e preparam para exibição da próxima imagem, incrementando `currentImage` em 1. Observe o cálculo de módulo para assegurar que `currentImage` é colocada em 0 quando seu incremento ultrapassar 29 (o último subscrito do *array*).

O método `stopAnimation` (linhas 74 a 77) interrompe a animação com a linha 76, que chama o método `stop` de `Timer` para indicar que o `Timer` deve parar de gerar eventos. Isso evita que `actionPerformed` chame `repaint` para iniciar a pintura da próxima imagem do *array*.

Observação de engenharia de software 18.1



Ao criar uma animação para utilizar em um applet, forneça um mecanismo para desativar a animação quando o usuário navegar para uma nova página da Web separada da página em que o applet de animação reside.

Os métodos `getPreferredSize` (linhas 80 a 83) e `getPreferredSize` (linhas 86 a 89) sobrescrevem os métodos correspondentes herdados da classe `Component` e permitem que os gerenciadores de layout determinem o ta-

manho apropriado de um **LogoAnimator** em um leiaute. Nesse exemplo, as imagens têm 160 pixels de largura e 80 pixels de altura, assim o método **getPreferredSize** retorna um objeto **Dimension** contendo os números 160 e 80. O método **getMinimumSize** simplesmente chama **getPreferredSize** (uma prática comum de programação).

As linhas 107 a 112 de **main** dimensionam a janela do aplicativo com base no tamanho preferido de **LogoAnimators** e os *insets* da janela. Os *insets* especificam o número de pixels das bordas superior, inferior, esquerda e direita da janela. Usar os *insets* da janela assegura que a área do cliente da janela é grande o suficiente para exibir o **LogoAnimator** corretamente. A *área do cliente* da janela é a parte da janela na qual a janela exibe os componentes GUI. O programa obtém os *insets* da janela chamando o método **getInsets** na linha 107. O método devolve um objeto **Insets** que contém os membros de dados **public top, bottom, left e right**.

18.4 Personalizando LogoAnimator através de parâmetros de applet

Ao navegar pela World Wide Web, você irá se deparar freqüentemente com *applets* que são de domínio público – você pode utilizá-los gratuitamente em suas próprias páginas da Web (normalmente em troca do reconhecimento da autoria do *applet*). Um recurso comum de tais *applets* é a capacidade de personalizá-lo através de parâmetros fornecidos no arquivo HTML que invoca o *applet*. Por exemplo, o HTML

```
<html>
<applet code="LogoApplet.class" width=400 height=400>
<param name="totalimages" value="30">
<param name="imagename" value="deitel">
<param name="animationdelay" value="200">
</applet>
</html>
```

do arquivo **LogoApplet.html**, invoca o *applet* **LogoApplet** (Fig. 18.4) e especifica três parâmetros. As linhas com as *marcas param* devem aparecer entre as marcas **applet** inicial e final. Cada parâmetro tem um **name** e um **value**. O método **getParameter** de **Applet** devolve um **String** que representa o **value** associado com um nome de parâmetro específico. O argumento para **getParameter** é um **String** que contém o nome do parâmetro na marca **param**. Por exemplo, a instrução

```
parameter = getParameter( "animationdelay" );
```

obtém o valor 200 associado com o parâmetro **animationdelay** e o atribui à referência **String parameter**. Se não houver uma marca **param** que contém o nome de parâmetro especificado, **getParameter** devolve **null**.

A classe **LogoAnimator2** (Fig. 18.3) estende a classe **LogoAnimator** e adiciona um construtor que recebe três argumentos – o número total de imagens na animação, o retardo entre a exibição de imagens e o nome básico das imagens. A classe **LogoApplet** (Fig. 18.4) permite aos projetistas de páginas da Web personalizar a animação para utilizar suas próprias imagens. Três parâmetros são fornecidos no documento HMTL do *applet*. O parâmetro **animationdelay** é o número de milissegundos para “dormir” entre exibições de imagens. Esse valor será convertido em um inteiro e utilizado como valor para a variável de instância **sleepTime**. O parâmetro **imagename** é o nome básico das imagens a carregar. Esse **String** será atribuído à variável de instância **imageName**. O *applet* supõe que as imagens estão em um subdiretório chamado **images** que pode ser localizado no mesmo diretório que o *applet*. O *applet* também supõe que os nomes dos arquivos de imagem são numerados a partir de 0 (como na Fig. 18.2). O parâmetro **totalimages** representa o número total de imagens na animação. Seu valor será convertido em um inteiro e atribuído à variável de instância **totalImages**.

```
1 // Fig. 18.3: LogoAnimator2.java
2 // Animando uma série de imagens
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6
7 // Pacotes de extensão de Java
8 import javax.swing.*;
```

Fig. 18.3 Subclasse **LogoAnimator2** de **LogoAnimator** (Fig. 18.2) adiciona um construtor para personalizar a quantidade de imagens, o retardo na animação e o nome básico das imagens (parte 1 de 2).

```

9
10 public class LogoAnimator2 extends LogoAnimator {
11
12     // construtor default
13     public LogoAnimator2()
14     {
15         super();
16     }
17
18     // novo construtor para dar suporte à personalização
19     public LogoAnimator2( int count, int delay, String name )
20     {
21         totalImages = count;
22         animationDelay = delay;
23         imageName = name;
24
25         initializeAnimation();
26     }
27
28     // inicia a animação como aplicativo em sua própria janela
29     public static void main( String args[] )
30     {
31         // cria LogoAnimator
32         LogoAnimator2 animation = new LogoAnimator2();
33
34         // configura a janela
35         JFrame window = new JFrame( "Animator test" );
36
37         Container container = window.getContentPane();
38         container.add( animation );
39
40         window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
41
42         // dimensiona e exibe a janela
43         window.pack();
44         Insets insets = window.getInsets();
45
46         window.setSize( animation.getPreferredSize().width +
47                         insets.left + insets.right,
48                         animation.getPreferredSize().height +
49                         insets.top + insets.bottom );
50
51         window.setVisible( true );
52         animation.startAnimation();    // inicia a animação
53
54     } // fim do método main
55
56 } // fim da classe LogoAnimator2

```

Fig. 18.3 Subclasse LogoAnimator2 de LogoAnimator (Fig. 18.2) adiciona um construtor para personalizar a quantidade de imagens, o retardo na animação e o nome básico das imagens (parte 2 de 2).

```

1 // Fig. 18.4: LogoApplet.java
2 // Personalizando um applet através de parâmetros de HTML.
3 //
4 // O parâmetro de HTML "animationdelay" é um int que indica
5 // quantos milissegundos adormecer entre imagens (default 50).
6 //
7 // O parâmetro de HTML "imagename" é o nome básico das imagens

```

Fig. 18.4 Personalizando um *applet* animado através da marca de HTML `param` (parte 1 de 2).

```

8 // que serão exibidas (i.e., "deitel" é o nome básico para
9 // as imagens "deitel0.gif," "deitell.gif," etc.). O applet
10 // supõe que as imagens estão em um subdiretório "images"
11 // do diretório no qual o applet está armazenado.
12 //
13 // O parâmetro de HTML "totalimages" é um inteiro que representa
14 // a quantidade total de imagens na animação. O applet supõe
15 // que as imagens são numeradas de 0 a totalimages - 1 (default 30).
16 //
17 // Pacotes do núcleo de Java
18 import java.awt.*;
19 //
20 // Pacotes de extensão de Java
21 import javax.swing.*;
22 //
23 public class LogoApplet extends JApplet {
24
25     // obtém parâmetros do documento HTML e personaliza o applet
26     public void init()
27     {
28         String parameter;
29
30         // obtém retardo da animação do documento HTML
31         parameter = getParameter( "animationdelay" );
32
33         int animationDelay = ( parameter == null ?
34             50 : Integer.parseInt( parameter ) );
35
36         // obtém o nome básico das imagens do documento HTML
37         String imageName = getParameter( "imagename" );
38
39         // obtém a quantidade total de imagens do documento HTML
40         parameter = getParameter( "totalimages" );
41
42         int totalImages = ( parameter == null ?
43             0 : Integer.parseInt( parameter ) );
44
45         // cria instância de LogoAnimator
46         LogoAnimator2 animator;
47
48         if ( imageName == null || totalImages == 0 )
49             animator = new LogoAnimator2();
50         else
51             animator = new LogoAnimator2( totalImages,
52                 animationDelay, imageName );
53
54         // anexa o animador ao applet e inicia a animação
55         getContentPane().add( animator );
56         animator.startAnimation();
57
58     } // fim do método init
59
60 } // fim da classe LogoApplet

```

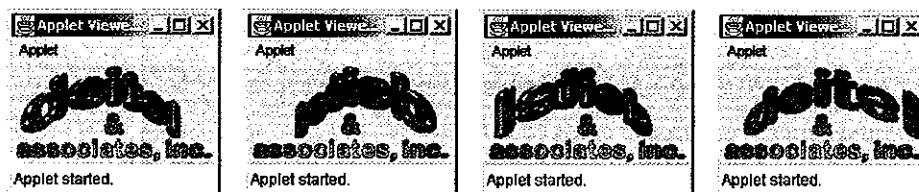


Fig. 18.4 Personalizando um applet animado através da marca de HTML param (parte 2 de 2).

A classe **LogoApplet** (Fig. 18.4) define um método **init** em que os três parâmetros de HTML são lidos com o método **getParameter** de **Applet** (linhas 31, 37 e 40). Depois que o *applet* lê estes parâmetros e os dois parâmetros inteiros são convertidos em valores **int**, a estrutura **if/else** nas linhas 48 a 51 cria um **LogoAnimator2** e chama seu construtor de três argumentos. Se o **imageName** é **null** ou **totalImages** for 0, o **applet** chama o construtor **default** **LogoAnimator2** e utiliza a animação **default**. Caso contrário, o **applet** passa **totalImages**, **animationDelay** e **imageName** para o construtor de três argumentos **LogoAnimator2** e o construtor utiliza esses argumentos para personalizar a animação. O construtor de três argumentos de **LogoAnimator2** invoca o método **initializeAnimation** de **LogoAnimator** para carregar as imagens e determinar a largura e a altura da animação.

18.5 Mapas de imagens

Os *mapas de imagens* são uma técnica comum usada para criar páginas da Web interativas. Trata-se de uma imagem que tem *áreas ativas* (*hot areas*) em que o usuário pode clicar para realizar uma tarefa como carregar uma página da Web diferente em um navegador. Quando o usuário posiciona o ponteiro do mouse sobre uma área ativa, normalmente uma mensagem descritiva é exibida na barra de estado do navegador ou em uma janela *pop-up*.

A Fig. 18.5 carrega uma imagem que contém diversos ícones comuns usados para as dicas ao longo deste livro. O programa permite que o usuário posicione o ponteiro do mouse sobre um ícone e exiba uma mensagem descritiva para o ícone. O tratador de eventos **mouseMoved** (linhas 42 a 46) obtém as coordenadas do mouse e as passa para o método **translateLocation** (linhas 63 a 76). O método **translateLocation** testa as coordenadas para determinar o ícone sobre qual o mouse estava posicionado quando ocorreu o evento **mouseMoved**. O método **translateLocation** então devolve uma mensagem que indica o que o ícone representa. Essa mensagem é exibida na barra de estado do **appletviewer** (ou navegador).

```

1 // Fig. 18.5: ImageMap.java
2 // Demonstrando um mapa de imagens.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class ImageMap extends JApplet {
12     private ImageIcon mapImage;
13
14     private String captions[] = { "Common Programming Error",
15         "Good Programming Practice",
16         "Graphical User Interface Tip", "Performance Tip",
17         "Portability Tip", "Software Engineering Observation",
18         "Testing and Debugging Tip" };
19
20     // configura ouvintes do mouse
21     public void init()
22     {
23         addMouseListener(
24
25             new MouseAdapter() {
26
27                 // indica quando o ponteiro do mouse sai da área do applet
28                 public void mouseExited( MouseEvent event )
29                 {

```

Fig. 18.5 Demonstrando um mapa de imagens (parte 1 de 3).

```

30         showStatus( "Pointer outside applet" );
31     }
32
33 } // fim da classe interna anônima
34
35 ); // fim da chamada para o método addMouseListener
36
37 addMouseMotionListener(
38
39     new MouseMotionAdapter() {
40
41         // determina o ícone sobre o qual o mouse aparece
42         public void mouseMoved( MouseEvent event )
43     {
44         showStatus( translateLocation(
45             event.getX(), event.getY() ) );
46     }
47
48 } // fim da classe interna anônima
49
50 ); // fim da chamada para o método addMouseMotionListener
51
52 mapImage = new ImageIcon( "icons.png" );
53
54 } // fim do método init
55
56 // exibe mapImage
57 public void paint( Graphics g )
58 {
59     mapImage.paintIcon( this, g, 0, 0 );
60 }
61
62 // devolve texto de dica com base nas coordenadas do mouse
63 public String translateLocation( int x, int y )
64 {
65     // se as coordenadas estiverem fora da imagem, retorna imediatamente
66     if ( x >= mapImage.getIconWidth() || 
67         y >= mapImage.getIconHeight() )
68         return "";
69
70     // determina o número do ícone (0 - 6)
71     int iconWidth = mapImage.getIconWidth() / 7;
72     int iconNumber = x / iconWidth;
73
74     // devolve o texto de dica apropriado para o ícone
75     return captions[ iconNumber ];
76 }
77
78 } // fim da classe ImageMap

```

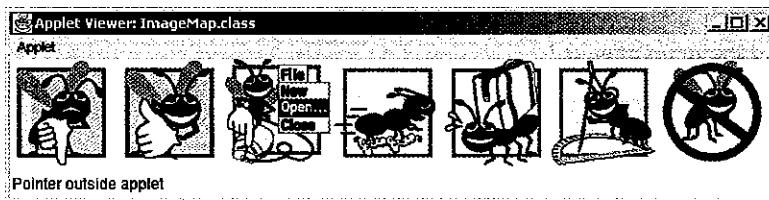


Fig. 18.5 Demonstrando um mapa de imagens (parte 2 de 3).



Fig. 18.5 Demonstrando um mapa de imagens (parte 3 de 3).

Clicar nesse *applet* não causará nenhuma ação. No Capítulo 17, discutimos as técnicas necessárias para carregar outra página da Web em um navegador através de URLs e da interface **AppletContext**. Usando essas técnicas, este *applet* pode associar cada ícone com um URL que o navegador exibe quando o usuário clica no ícone.

18.6 Carregando e reproduzindo clipes de áudio

Os programas Java podem manipular e reproduzir *clipes de áudio*. É fácil para os usuários capturar seus próprios clipes de áudio e existem muitos clipes disponíveis em produtos de *software* e através da Internet. O sistema precisa estar equipado com *hardware* de áudio (alto-falantes e uma placa de som) para ser capaz de reproduzir os clipes de áudio.

Java fornece diversos mecanismos para reproduzir sons em um *applet*. Os dois métodos mais simples são o método **play** de **Applet** e o método **play** da interface **AudioClip**. Se você quiser reproduzir o som apenas uma vez em um programa, o método **play** de **Applet** carrega o som e o reproduz uma vez; o som é marcado para coleta de lixo depois de ser reproduzido. O método **play** de **Applet** tem duas formas:

```
public void play( URL location, String soundFileName );
public void play( URL soundURL );
```

A primeira versão carrega o clipe de áudio armazenado no arquivo **soundFileName** a partir da **location** e reproduz o som. O primeiro argumento é normalmente uma chamada para o método **getDocumentBase** ou **getDocumentBase** do *applet*. O método **getDocumentBase** indica a localização do arquivo HTML que carregou o *applet* (se o *applet* estiver em um pacote, isto indica a localização do pacote ou do arquivo JAR que contém o pacote). O método **getDocumentBase** indica a localização do arquivo **.class** do *applet*. A segunda versão do método **play** recebe um **URL** que contém a localização e o nome de arquivo do clipe de áudio. A instrução

```
play( getDocumentBase(), "hi.au" );
```

carrega o clipe de áudio do arquivo **hi.au** e o reproduz uma vez.

A máquina de som que reproduz os clipes de áudio suporta diversos formatos de arquivos de áudio, incluindo o formato de arquivo *Sun Audio* (extensão **.au**), o *Windows Wave* (extensão **.wav**), o *Macintosh AIFF* (extensões **.aif** ou **.aiff**) e o *Musical Instrument Digital Interface (MIDI)* (extensões **.mid** ou **.rmi**). O Java Media Framework (JMF) e a API Java Sounds suportam outros formatos adicionais.

O programa da Fig. 18.6 demonstra a carga e reprodução de um **AudioClip** (pacote **java.applet**). Essa técnica é mais flexível que o método **play** de **Applet**. O *applet* pode usar um **AudioClip** para armazenar áudio para uso repetido ao longo de toda a execução do programa. O método **getAudioClip** de **Applet** tem duas formas que recebem os mesmos argumentos que o método **play** descrito anteriormente. O método **getAudioClip** devolve uma referência para um **AudioClip**. O **AudioClip** tem três métodos – **play**, **loop** e **stop**. O método **play** reproduz o áudio uma vez. O método **loop** repete continuamente o clipe de áudio como som de fundo. O método **stop** termina um clipe de áudio que atualmente está sendo reproduzido. No programa, cada um desses métodos é associado a um botão no *applet*.

As linhas 62 e 63 no método **init** do *applet* utilizam **getAudioClip** para carregar dois arquivos de áudio – um arquivo Windows Wave (**welcome.wav**) e um arquivo de Sun Audio (**hi.au**). O usuário pode selecionar qual clipe de áudio reproduzir a partir da **JComboBox chooseSound**. Repare que o método **stop** do *applet* é sobreescrito nas linhas 69 a 72. Quando o usuário muda de página na Web, o contêiner de *applets* chama o método **stop** do *applet*. Isso permite que o *applet* pare de reproduzir o clipe de áudio. Caso contrário, o clipe de áudio continuaria a ser reproduzido em segundo plano – mesmo se o *applet* não estiver sendo exibido no navegador. Isso não é realmente um problema, mas pode irritar o usuário se o clipe de áudio estiver sendo repetido. O método **stop** é colocado aqui para conveniência do usuário.

Boa prática de programação 18.1

Ao reproduzir clipes de áudio em um *applet* ou aplicativo, forneça um mecanismo para o usuário desativar o áudio.

```
1 // Fig. 18.6: LoadAudioAndPlay.java
2 // Carrega um clipe de áudio e o reproduz.
```

Fig. 18.6 Carregando e reproduzindo um **AudioClip** (parte 1 de 3).

```
3 // Pacotes do núcleo de Java
4 import java.applet.*;
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class LoadAudioAndPlay extends JApplet {
12     private AudioClip sound1, sound2, currentSound;
13     private JButton playSound, loopSound, stopSound;
14     private JComboBox chooseSound;
15
16     // carrega a imagem quando o applet começa a ser executado
17     public void init()
18     {
19         Container container = getContentPane();
20         container.setLayout( new FlowLayout() );
21
22         String choices[] = { "Welcome", "Hi" };
23         chooseSound = new JComboBox( choices );
24
25         chooseSound.addItemListener(
26             new ItemListener() {
27
28                 // faz parar o som e muda para o som selecionado pelo usuário
29                 public void itemStateChanged( ItemEvent e )
30                 {
31                     currentSound.stop();
32
33                     currentSound =
34                         chooseSound.getSelectedIndex() == 0 ?
35                         sound1 : sound2;
36                 }
37
38             } // fim da classe interna anônima
39
40     } // fim da chamada para o método addItemClickListener
41
42     container.add( chooseSound );
43
44     // configura o tratador de eventos de botões e os botões
45     ButtonHandler handler = new ButtonHandler();
46
47     playSound = new JButton( "Play" );
48     playSound.addActionListener( handler );
49     container.add( playSound );
50
51     loopSound = new JButton( "Loop" );
52     loopSound.addActionListener( handler );
53     container.add( loopSound );
54
55     stopSound = new JButton( "Stop" );
56     stopSound.addActionListener( handler );
57     container.add( stopSound );
58
59     // carrega sons e configura currentSound
60     sound1 = getAudioClip( getDocumentBase(), "welcome.wav" );
```

Fig. 18.6 Carregando e reproduzindo um AudioClip (parte 2 de 3).

```

63     sound2 = getAudioClip( getDocumentBase(), "hi.au" );
64     currentSound = sound1;
65
66 } // fim do método init
67
68 // faz parar o som quando o usuário muda de página da Web
69 public void stop()
70 {
71     currentSound.stop();
72 }
73
74 // classe interna privada para tratar de eventos de botão
75 private class ButtonHandler implements ActionListener {
76
77     // processa eventos dos botões play, loop e stop
78     public void actionPerformed( ActionEvent actionEvent )
79     {
80         if ( actionEvent.getSource() == playSound )
81             currentSound.play();
82
83         else if ( actionEvent.getSource() == loopSound )
84             currentSound.loop();
85
86         else if ( actionEvent.getSource() == stopSound )
87             currentSound.stop();
88     }
89 }
90
91 } // fim da classe LoadAudioAndPlay

```



Fig. 18.6 Carregando e reproduzindo um `AudioClip` (parte 3 de 3).

18.7 Recursos na Internet e na World Wide Web

Esta seção apresenta vários recursos na Internet e na Web para *sites* relacionados com multimídia (recursos adicionais são fornecidos no Capítulo 22).

www.nasa.gov/gallery/index.html

A NASA *Multimedia Gallery* contém uma grande variedade de imagens, clipes de áudio e videoclipes que você pode baixar e utilizar para testar seus programas de multimídia Java.

sunsite.sut.ac.jp/multimed/

A *Sunsite Japan Multimedia Collection* também fornece uma ampla variedade de imagens, clipes de áudio e videoclipes que você pode baixar para fins educacionais.

www.anbg.gov.au/anbg/index.html

O site da Web *Australian National Botanic Gardens* fornece *links* para muitos sons de animais. Experimente o link *Common Birds*.

www.thefreesite.com

O *TheFreeSite.com* tem *links* para sons e desenhos gráficos.

www.soundcentral.com

O *SoundCentral* fornece clipes de áudio nos formatos WAV, AU, AIFF e MIDI.

www.animationfactory.com

A *Animation Factory* fornece milhares de animações GIF para uso pessoal.

www.clipart.com

O *ClipArt.com* contém *links* para *sites* da Web que fornecem desenhos gráfitos.

www.pngart.com

O *PNGART.com* fornece mais de 50.000 imagens gratuitas no formato PNG, num esforço para ajudar este novo formato a se tornar popular.

developer.java.sun.com/developer/techDocs/hi/repository

O *Java Look-and-Feel Graphics Repository* fornece imagens-padrão para uso numa GUI Swing.

Resumo

- O método `getImage` de `Applet` carrega uma `Image`. A versão de `getImage` recebe dois argumentos – uma localização em que a imagem está armazenada e o nome do arquivo da imagem.
- O método `getDocumentBase` de `Applet` devolve a localização do arquivo HTML do *applet* na Internet como objeto da classe `URL` (pacote `java.net`).
- Java suporta diversos formatos de imagem, inclusive Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG) e Portable Network Graphics (PNG). Os nomes de arquivos para cada um desses tipos terminam em `.gif`, `.jpg` (ou `.jpeg`) ou `.png`, respectivamente.
- A classe `ImageIcon` fornece construtores que permitem que um objeto `ImageIcon` seja inicializado com uma imagem do computador local ou com uma imagem armazenada em um servidor da Web na Internet.
- O método de `drawImage` de `Graphics` recebe quatro argumentos – uma referência ao objeto `Image` em que a imagem está armazenada, as coordenadas *x* e *y* em que a imagem deve ser exibida e uma referência para um objeto `ImageObserver`.
- Outra versão do método `drawImage` de `Graphics` envia para a saída uma imagem *dimensionada*. O quarto e o quinto argumentos especificam a largura e a altura da imagem para fins de exibição.
- A interface `ImageObserver` é implementada pela classe `Component` (uma superclasse indireta de `Applet`). `ImageObservers` são notificados para atualizar uma imagem que foi exibida à medida que o restante da imagem é carregado.
- O método `paintIcon` de `ImageIcon` exibe a imagem do `ImageIcon`. O método requer quatro argumentos – uma referência para o `Component` em que a imagem será exibida, uma referência para o objeto `Graphics` utilizado para exibir a imagem, a coordenada *x* do canto superior esquerdo da imagem e a coordenada *y* do canto superior esquerdo da imagem.
- A método `paintIcon` da classe `ImageIcon` não permite o redimensionamento de uma imagem. A classe fornece o método `getImage` que devolve uma referência a uma `Image`, a qual pode ser utilizada com o método `drawImage` de `Graphics` para exibir uma versão redimensionada de uma imagem.
- Os objetos `Timer` geram `ActionEvents` a intervalos fixos em milissegundos e notificam seus `ActionListeners` registrados de que ocorreram eventos. O construtor `Timer` recebe dois argumentos – o retardo em milissegundos e o `ActionListener`. O método `start` de `Timer` indica que o `Timer` deve começar a gerar eventos. O método `stop` de `Timer` indica que o `Timer` deve parar de gerar eventos. O método `restart` de `Timer` indica que o `Timer` deve começar a gerar eventos novamente.
- Os *applets* podem ser personalizados através de parâmetros (a marca `<param>`) que são fornecidos pelo arquivo HTML que invoca o *applet*. As linhas da marca `<param>` devem aparecer entre a marca `applet` inicial e a marca `applet` final. Cada parâmetro tem um `name` e um `value`.
- O método `getParameter` de `Applet` obtém o `value` associado com um parâmetro específico e retorna o `value` como `String`. O argumento passado para `getParameter` é um `String` que contém o nome do parâmetro na marca `param`. Se não houver uma marca `param` que contém o parâmetro especificado, `getParameter` retorna `null`.
- O mapa de imagem é uma imagem que tem áreas ativas em que o usuário pode clicar para realizar uma tarefa, como carregar uma página da Web diferente em um navegador.
- O método `play` de `Applet` tem duas formas:

```
public void play( URL location, String soundFileName );
public void play( URL soundURL );
```

- Uma versão carrega o clipe de áudio armazenado no arquivo `soundFileName` na `location` e reproduz o som; a outra aceita um `URL` que contém a localização e o nome do arquivo do clipe de áudio.

- O método `getDocumentBase` de `Applet` indica a localização do arquivo HTML que carregou o *applet*. O método `getCodeBase` indica onde está localizado o arquivo `.class` para um *applet*.
- A máquina de som que reproduz clipes de áudio suporta diversos formatos de arquivos de áudio incluindo o formato de arquivo *Sun Audio* (extensão `.au`), o *Windows Wave* (extensão `.wav`), o *Macintosh AIFF* (extensão `.aif` ou `.aiff`) e o *Musical Instrument Digital Interface* (MIDI) (extensões `.mid` ou `.rmi`). O Java Media Framework (JMF) suporta outros formatos adicionais.
- O método `getAudioClip` de `Applet` tem duas formas que recebem os mesmos argumentos que o método `play`. O método `getAudioClip` retorna uma referência para um `AudioClip`. `AudioClips` têm três métodos – `play`, `loop` e `stop`. O método `play` reproduz o áudio uma vez. O método `loop` faz uma repetição contínua do clipe de áudio. O método `stop` termina um clipe de áudio que atualmente está sendo reproduzido.

Terminologia

<i>altura de uma imagem</i>	<i>Joint Photographic Experts Group (JPEG)</i>
<i>animação</i>	<i>largura de uma imagem</i>
<i>animando uma série de imagens</i>	<i>mapa de imagem</i>
<i>área ativa de um mapa de imagem</i>	<i>máquina de som</i>
<i>arquivo Macintosh AIFF (.aif ou .aiff)</i>	<i>marca param</i>
<i>arquivo Windows Wave (.wav)</i>	<i>método drawImage de Graphics</i>
<i>atributo nome da marca param</i>	<i>método getAudioClip de Applet</i>
<i>atributo value da marca param</i>	<i>método getCodeBase de Applet</i>
<i>botão de informações</i>	<i>método getDocumentBase de Applet</i>
<i>botão mute</i>	<i>método getHeight de Component</i>
<i>classe Image</i>	<i>método getIconHeight de ImageIcon</i>
<i>classe ImageIcon</i>	<i>método getIconWidth de ImageIcon</i>
<i>classe Timer</i>	<i>método getImage de Applet</i>
<i>clipe de áudio</i>	<i>método getImage de ImageIcon</i>
<i>controle de volume</i>	<i>método getParameter de Applet</i>
<i>escalonar uma imagem</i>	<i>método getWidth de Component</i>
<i>extensão de nome de arquivo .aif</i>	<i>método loop da interface AudioClip</i>
<i>extensão de nome de arquivo .aiff</i>	<i>método paintIcon da classe ImageIcon</i>
<i>extensão de nome de arquivo .au</i>	<i>método play da classe Applet</i>
<i>extensão de nome de arquivo .gif</i>	<i>método play da interface AudioClip</i>
<i>extensão de nome de arquivo .jpeg</i>	<i>método restart da classe Timer</i>
<i>extensão de nome de arquivo .mid</i>	<i>método start da classe Timer</i>
<i>extensão de nome de arquivo .rmi</i>	<i>método stop da classe Timer</i>
<i>extensão de nome de arquivo .wav</i>	<i>método stop da interface AudioClip</i>
<i>extensão de nome de arquivo .jpg</i>	<i>método update da classe Component</i>
<i>formato de arquivo Sun Audio (.au)</i>	<i>multimídia</i>
<i>gráfico</i>	<i>Musical Instrument Digital Interface (MIDI)</i>
<i>Graphics Interchange Format (GIF)</i>	<i>personalizar um applet</i>
<i>imagens</i>	<i>relação de troca espaço/tempo</i>
<i>imagens gráficas</i>	<i>som</i>
<i>interface ImageObserver</i>	

Exercícios de auto revisão

18.1 Preencha as lacunas em cada uma das frases seguintes:

- O método _____ `Applet` carrega uma imagem em um *applet*.
- O método _____ de `Applet` devolve a localização na Internet do arquivo HTML que invocou o *applet*, como um objeto da classe `URL`.
- O método _____ de `Graphics` exibe uma imagem em um *applet*.
- Java fornece dois mecanismos para reproduzir sons em um *applet* – o método `play` de `Applet` e o método `play` da interface _____.
- O _____ é uma imagem que tem *áreas ativas* em que o usuário pode clicar para realizar uma tarefa, como carregar uma página diferente da Web.
- O método _____ da classe `ImageIcon` exibe a imagem `ImageIcon`.
- Java suporta diversos formatos de imagem, incluindo _____, _____ e _____.

- 18.2** Diga se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- Um som será coletado como lixo logo que ele terminar de ser reproduzido.
 - A classe `ImageIcon` fornece construtores que permitem que um objeto `ImageIcon` seja inicializado sómente com uma imagem do computador local.
 - O método `getParameter` de `Applet` obtém o `value` associado com um parâmetro de HTML específico e devolve o `value` como um `String`.

Respostas aos exercícios de auto-revisão

18.1 a) `getImage`. b) `getDocumentBase`. c) `drawImage`. d) `AudioClip`. e) mapa de imagem. f) `paintIcon`. g) Graphics Interchange Format (GIF), Joint Photograph Experts Group (JPEG) e Portable Network Graphics (PNG).

18.2 a) Falsa; o som será marcado para coleta de lixo (se não for mencionado por um `AudioClip`) e será eliminado quando o coletor de lixo for capaz de ser executado. b) Falsa. `ImageIcon` pode carregar imagens da Internet também. c) Verdadeira.

Exercícios

- 18.3** Descreva como fazer uma animação “amigável para o navegador”.
- 18.4** Descreva os métodos Java para reproduzir e manipular clipes de áudio.
- 18.5** Como os *applets* Java podem ser personalizados com as informações de um arquivo HTML?
- 18.6** Explique como os mapas de imagem são utilizados. Liste exemplos nos quais são usados mapas em imagem.
- 18.7** (*Apagar aleatoriamente uma imagem*) Suponha que uma imagem seja exibida em uma área de tela retangular. Uma maneira de apagar a imagem é simplesmente configurar todos os *pixels* com a mesma cor imediatamente, mas isso resulta em um efeito visual desagradável. Escreva um programa Java que exibe uma imagem e depois a apaga, utilizando a geração de números aleatórios para selecionar *pixels* a apagar. Depois que a maior parte da imagem tiver sido apagada, apague todos os *pixels* restantes de uma vez. Você pode fazer referência a *pixels* isolados com uma linha que inicia e termina no mesmo ponto. Você poderia tentar diversas variantes desse problema. Por exemplo, você poderia exibir linhas aleatoriamente ou exibir formas aleatoriamente para apagar regiões da tela.
- 18.8** (*Flasher de texto*) Crie um programa Java que faça um texto piscar repetidamente na tela. Faça isso alternando a exibição do texto com a de uma imagem somente com a cor de fundo. Permita que o usuário controle a “velocidade de piscamento” e o padrão ou a cor de fundo.
- 18.9** (*Flasher de imagem*) Crie um programa Java que faça uma imagem piscar repetidamente na tela. Faça isso alternando a exibição da imagem com a de uma imagem somente com a cor de fundo.
- 18.10** (*Relógio digital*) Implemente um programa que exibe um relógio digital na tela. Você pode adicionar opções para redimensionar o relógio, para exibir o dia, o mês e o ano, para disparar um alarme, para reproduzir certos sons em horários designados e coisas parecidas.
- 18.11** (*Ampliador de imagem*) Crie um programa que permita ampliar (*zoom in*) ou reduzir (*zoom out*) uma imagem.

Seção especial: projetos desafiadores de multimídia

Os exercícios anteriores são relacionados com o texto e projetados para testar a compreensão de conceitos fundamentais de multimídia pelo leitor. Esta seção inclui um conjunto de projetos avançados de multimídia. O leitor achará que estes problemas são desafiadores mas, ainda assim, divertidos. Os problemas variam consideravelmente em grau de dificuldade. Alguns requerem uma ou duas horas de codificação e implementação de programa. Outros são úteis para trabalhos de laboratório que poderiam exigir duas ou três semanas de estudo e implementação. Alguns são projetos desafiadores para um semestre. [Nota: não são fornecidas soluções para estes exercícios.]

18.13 (*Animação*) Crie um programa de animação Java de uso geral. O programa deve permitir que o usuário especifique a sequência de *frames* a ser exibida, a velocidade com que as imagens são exibidas, os sons que devem ser reproduzidos enquanto a animação está sendo executada e assim por diante.

18.14 (*Limericks*) Modifique o programa de escrever *limericks* que você escreveu no Exercício 10.10 para cantar os *limericks* que o programa cria.

18.15 (*Transição aleatória entre diferentes imagens*) Aqui está um efeito visual interessante. Se você está exibindo uma imagem em uma determinada área na tela e quer fazer uma transição para outra imagem na mesma área de tela, armazene a nova imagem de tela em um *buffer* fora da tela e copie aleatoriamente *pixels* da nova imagem para a área de exibição sobrepondo os *pixels* anteriores nessas posições. Quando a maioria dos *pixels* tiver sido copiada, copie a nova imagem inteira para a área de exibição para certificar-se de que você está exibindo a nova imagem completa. Para implementar esse programa, você pode precisar utilizar as classes **PixelGrabber** e **MemoryImageSource** (veja a documentação de Java API para conhecer as descrições dessas classes). Você poderia tentar diversas variantes desse problema. Por exemplo, tente selecionar todos os *pixels* em uma linha reta ou uma forma selecionada aleatoriamente na nova imagem e sobrepor esses *pixels* às posições correspondentes na imagem antiga.

18.16 (*Áudio de fundo*) Adicione áudio de fundo para um de seus aplicativos favoritos utilizando o método **loop** da classe **AudioClip** para reproduzir o som no fundo enquanto interage com seu aplicativo na maneira normal.

18.17 (*Painel Marquee que desloca o texto*) Crie um programa Java que desloca caracteres pontilhados da direita para a esquerda (ou da esquerda para a direita se isso for apropriado para seu idioma) ao longo de um painel de exibição do tipo *Marquee*. Como opção, exiba o texto em um laço contínuo de modo que o texto reapareça por uma extremidade depois de sair pela outra.

18.18 (*Marquee que rola uma imagem*) Crie um programa Java que rola uma imagem ao longo de uma tela de *Marquee*.

18.19 (*Relógio analógico*) Crie um programa Java que exibe um relógio analógico com ponteiros de hora, minuto e segundos que se movem apropriadamente conforme o tempo passa.

18.20 (*Caleidoscópio dinâmico com áudio e gráfico*) Escreva um programa de caleidoscópio que exibe imagens gráficas refletidas para simular o brinquedo infantil popular. Incorpore efeitos de áudio que “espelhem” a mudança dinâmica de imagens no seu programa.

18.21 (*Gerador automático de quebra-cabeças*) Crie um gerador e manipulador de quebra-cabeças em Java. O usuário especifica uma imagem. O programa carrega e exibe a imagem. Depois, o programa divide a imagem em formas selecionadas aleatoriamente e embaralha as formas. O usuário então utiliza o mouse para mover as peças do quebra-cabeça para resolvê-lo. Adicione sons de áudio apropriados à medida que as peças são embaralhadas e encaixadas em seus respectivos lugares. Você poderia controlar a posição de cada peça e o lugar em que ela deve ficar, de modo a utilizar efeitos de áudio para ajudar o usuário a encaixar as peças nas posições corretas.

18.22 (*Gerando e percorrendo um labirinto*) Desenvolva um gerador de labirintos baseado em multimídia e um programa que percorra esse labirinto com base nos programas de labirintos escritos nos Exercícios 7.38 a 7.40. Deixe o usuário personalizar o labirinto especificando o número de linhas e colunas e indicando o nível de dificuldade. Faça um rato animado percorrer o labirinto. Utilize áudio para dramatizar o movimento do ratinho.

18.23 (*Caça-níqueis*) Desenvolva uma simulação de uma máquina caça-níqueis com multimídia. Crie três rodas giratórias. Coloque várias frutas e símbolos em cada roda. Utilize geração de números aleatórios de verdade para simular o giro de cada roda e a parada em um símbolo.

18.24 (*Corrida de cavalos*) Crie uma simulação em Java de uma corrida de cavalos. Tenha múltiplos competidores. Utilize áudios para um apresentador da corrida. Reproduza os áudios apropriados para indicar o estado correto de cada um dos competidores ao longo de toda a corrida. Utilize áudios para anunciar os resultados finais. Você pode tentar simular os tipos de jogo de corrida de cavalos que geralmente ocorrem nos parques de diversões. Cada jogador tem uma vez para usar o mouse e tem de realizar alguma manipulação baseada em suas habilidades com o mouse para avançar seus respectivos cavalos.

18.25 (*Jogo de malha*) Desenvolva uma simulação baseada em multimídia do jogo de malha (*Shuffleboard*). Utilize áudio e efeitos visuais apropriados.

18.26 (*Jogo de bilhar*) Crie uma simulação baseada em multimídia do jogo de bilhar. Cada jogador tem uma vez para usar o mouse a fim de posicionar o taco de bilhar e acertar o taco contra a bola no ângulo apropriado para tentar fazer as bolas de bilhar caírem nas caçapás. O programa deve manter uma contagem de pontos.

18.27 (*Artista*) Projete um programa de arte em Java que forneça para um artista uma grande variedade de recursos para desenhar, utilizar imagens, utilizar animações, etc., para criar uma exposição dinâmica de arte em multimídia.

18.28 (*Designer de fogos de artifício*) Crie um programa em Java que alguém poderia utilizar para criar um espetáculo de fogos de artifício. Crie diversas demonstrações de fogos de artifício. Depois orquestre a queima dos fogos de artifício para obter um efeito máximo.

18.29 (*Planejador de planta baixa*) Desenvolva um programa Java que ajude alguém a organizar os móveis em sua casa. Adicione recursos que ajudem a pessoa a alcançar a melhor disposição possível.

18.30 (*Palavras cruzadas*) Palavras cruzadas estão entre os passatempos mais populares. Desenvolva um programa de palavras cruzadas baseado em multimídia. O programa deve permitir que o jogador escreva e apague as palavras facilmente. Associe o programa a um grande dicionário computadorizado. O programa deve ser capaz de sugerir as palavras com base nas letras já preenchidas. Forneça outros recursos que facilitem o trabalho do entusiasta das palavras cruzadas.

18.31 (*Jogo do 15*) Escreva um programa em Java baseado em multimídia que permite que o usuário jogue o jogo do 15. Há um tabuleiro de 4 por 4 com um total de 16 posições. Uma das posições está vazia. As outras posições estão ocupadas por 15 peças quadradas numeradas de 1 a 15. Qualquer peça ao lado da posição atualmente vazia pode ser movida para essa posição clicando na peça. O programa deve criar o tabuleiro com as peças fora de ordem. O objetivo é organizar os ladrilhos na ordem seqüencial, linha por linha.

18.32 (*Testador de precisão de reação/tempo de reação*) Crie um programa Java que move uma forma criada aleatoriamente pela tela. O usuário move o mouse para capturar e clicar na forma. A velocidade e o tamanho da forma podem ser variados. Mantenha estatísticas sobre quanto tempo o usuário geralmente leva para capturar uma forma de um determinado tamanho. O usuário provavelmente terá mais dificuldade para capturar formas que se movem mais rápido e menores.

18.33 (*Arquivo de calendário/lembretes*) Crie um arquivo de calendário e de lembretes de uso geral utilizando áudio e imagens. Por exemplo, o programa deve cantar “Feliz Aniversário” quando você utilizá-lo em seu aniversário. Faça com que o programa exiba imagens e reproduza áudios associados com eventos importantes. Além disso, faça o programa lembrá-lo com antecedência destes eventos importantes. Seria interessante, por exemplo, fazer o programa lhe dar um aviso com uma semana de antecedência, de modo que você possa enviar um cartão de felicitações apropriado para essa pessoa especial.

18.34 (*Rotação de imagens*) Crie um programa Java que permita girar uma imagem em vários graus (até um máximo de 360 graus). O programa deve permitir que você especifique que deseja girar a imagem continuamente. O programa deve permitir que você ajuste a velocidade de rotação dinamicamente.

18.35 (*Colorindo fotografias e imagens P&B*) Crie um programa em Java que permita colorir uma fotografia em preto-e-branco. Forneça uma paleta de cores para selecionar cores. O programa deve permitir que você aplique cores diferentes a regiões diferentes da imagem.

18.36 (*Simulador Simpletron baseado em multimídia*) Modifique o simulador Simpletron que você desenvolveu nos exercícios dos capítulos anteriores para incluir recursos de multimídia. Adicione sons semelhantes aos de um computador para indicar que o Simpletron está executando instruções. Adicione um som de vidro quebrando quando ocorrer um erro fatal. Utilize luzes piscando para indicar quais as células de memória e/ou os registradores que estão sendo manipulados atualmente. Utilize outras técnicas de multimídia adequadas para tornar o simulador mais valioso como ferramenta educacional para usuários.

Estruturas de dados

Objetivos

- Ser capaz de formar estruturas de dados encadeadas com referências, classes auto-referenciais e recursão.
- Ser capaz de criar e manipular estruturas de dados dinâmicas como listas encadeadas, filas, pilhas e árvores binárias.
- Entender várias aplicações importantes de estruturas de dados encadeadas.
- Entender como criar estruturas de dados reutilizáveis com classes, herança e composição.

*Much that I bound, I could not free;
Much that I freed returned to me.*

Lee Wilson Dodd

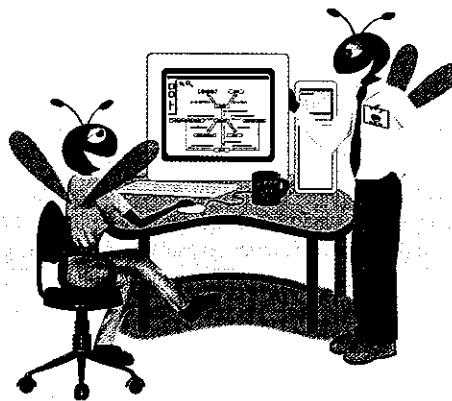
*'Will you walk a little faster?' said a whiting to a snail,
'There's a porpoise close behind us, and he's treading on
my tail.'*

Lewis Carroll

Há sempre espaço no ponto mais alto.
Daniel Webster

Prossiga – continue andando.
Thomas Morton

*I think that I shall never see
A poem lovely as a tree.*
Joyce Kilmer



Sumário do capítulo

- 19.1 Introdução
- 19.2 Classes auto-referenciais
- 19.3 Alocacão dinâmica de memória
- 19.4 Listas encadeadas
- 19.5 Pilhas
- 19.6 Filas
- 19.7 Árvores

Resumo • Termoologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão • Exercícios • Seção especial: construindo seu próprio compilador

19.1 Introdução

Estudamos *estruturas de dados* de tamanho fixo como *arrays* uni e bidimensionais. Este capítulo apresenta *estruturas de dados dinâmicas* que crescem e encolhem durante a execução. As *listas encadeadas* são coleções de itens de dados “colocados em fila” – as inserções e exclusões podem ser feitas em qualquer lugar em uma lista encadeada. As *pilhas* são importantes em compiladores e sistemas operacionais – as inserções e as exclusões são feitas somente em uma extremidade de uma pilha – sua *parte superior (topo)*. As *filas* representam filas de espera; as inserções são feitas na parte de trás (também conhecida como *cauda*) de uma fila e as exclusões são feitas a partir da parte da frente (também conhecida como *cabeça*) de uma fila. As *árvores binárias* facilitam a pesquisa e a classificação de dados em alta velocidade, a eliminação eficiente de itens de dados duplicados, a representação de diretórios de sistema de arquivos, a compilação de expressões em linguagem de máquina e muitas outras aplicações interessantes.

Discutiremos cada um dos principais tipos de estruturas de dados e implementaremos os programas que os criam e manipulam. Utilizamos classes, herança e composição para criar e empacotar essas estruturas de dados para que possam ser reutilizadas e mantidas. No Capítulo 20 e no Capítulo 21, discutimos as classes predefinidas de Java que implementam as estruturas de dados discutidas neste capítulo.

Os exemplos do capítulo são programas práticos que podem ser utilizados em cursos mais avançados e em aplicativos em empresas. Os exercícios incluem uma rica coleção de aplicativos úteis.

Sugerimos que você tente implementar o projeto principal descrito na seção especial intitulada “Construindo seu próprio compilador”. Você vem utilizando um compilador para traduzir seus programas Java em *bytecodes* para que você possa executar esses programas em seu computador. Nesse projeto, você realmente construirá seu próprio compilador. Este lerá um arquivo de instruções escrito em uma linguagem de alto nível simples mas poderosa, semelhante às versões iniciais da conhecida linguagem Basic. O compilador traduzirá essas instruções para um arquivo de instruções da Simpletron Machine Language (SML) – a SML é a linguagem que você aprendeu na seção especial do Capítulo 7, *Construindo seu próprio computador*. O programa simulador do Simpletron executará, então, o programa SML produzido por seu compilador! A implementação desse projeto utilizando uma abordagem orientada a objetos irá lhe fornecer uma oportunidade maravilhosa de praticar grande parte do que você aprendeu neste livro. A seção especial o conduz cuidadosamente através das especificações da linguagem de alto nível e descreve os algoritmos de que você precisará para converter cada tipo de instrução de linguagem de alto nível em instruções de linguagem de máquina. Se você gosta de desafios, pode tentar os muitos melhoramentos para compilador e para o simulador do Simpletron sugeridos nos exercícios.

19.2 Classes auto-referenciais

A classe *auto-referencial* contém uma variável de instância que faz referência a outro objeto do mesmo tipo de classe. Por exemplo, a definição

```
class Node {
    private int data;
```

```

private Node nextNode;

public Node( int data )
public void setData( int data )
public int getData()
public void setNext( Node next )
public Node getNext()
}
    
```

define a classe **Node** (nodo). Esse tipo tem duas variáveis de instância **private** – o inteiro **data** e a referência para **Node nextNode**. O membro **nextNode** faz referência a um objeto do tipo **Node**, um objeto do mesmo tipo que aquele sendo declarado aqui – daí o termo “classe auto-referencial”. O membro **nextNode** é um *link* – **nextNode** “vincula” um objeto de tipo **Node** a outro objeto do mesmo tipo. O tipo **Node** também tem cinco métodos: um construtor que recebe um inteiro para inicializar **data**, um método **setData** para configurar o valor **data**, um método **getData** para devolver o valor de **data**, um método **setNext** para configurar o valor de **nextNode** e um método **getNext** para devolver o valor do membro **nextNode**.

Os programas podem encadear objetos auto-referenciais para formar estruturas de dados úteis como listas, filas, pilhas e árvores. A Fig. 19.1 ilustra dois objetos auto-referenciais encadeados entre si para formar uma lista. A barra invertida – representando uma referência **null** – é colocada no membro de *link* do segundo objeto auto-referencial para indicar que o *link* não faz referência a outro objeto. A barra invertida é para fins de ilustração; ela não corresponde ao caractere de barra invertida em Java. A referência **null** normalmente indica o fim de uma estrutura de dados.



Erro comum de programação 19.1

Não configurar o link no último nodo de uma lista como null é um erro de lógica.

19.3 Alocação dinâmica de memória

Criar e manter estruturas de dados dinâmicas exige *alocação dinâmica de memória* – a capacidade de um programa obter mais espaço de memória durante a execução para armazenar novos nodos e liberar espaço não mais necessário. Como já aprendemos, os programas Java não liberam explicitamente memória alocada dinamicamente. Em vez disso, Java realiza a coleta de lixo automática sobre objetos aos quais não se faz mais referência em um programa.

O limite para alocação dinâmica de memória pode ser tão grande quanto a quantidade de memória física disponível no computador ou a quantidade de espaço disponível em disco em um sistema de memória virtual. Freqüentemente, os limites são muito menores porque a memória disponível do computador deve ser compartilhada entre muitos aplicativos.

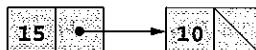


Fig. 19.1 Dois objetos de classe auto-referencial encadeados.

O operador **new** é essencial para alocação dinâmica de memória. O operador **new** recebe como operando o tipo do objeto que está sendo dinamicamente alocado e devolve uma referência para um objeto desse tipo recém-criado. Por exemplo, a instrução

```
Node nodeToAdd = new Node( 10 );
```

aloca a quantidade adequada de memória para armazenar um objeto **Node** e armazena uma referência para esse objeto em **nodeToAdd**. Se não houver memória disponível, **new** dispara um **OutOfMemoryError**. O 10 são os dados do objeto **Node**.

As seções a seguir discutem listas, pilhas, filas e árvores, que usam alocação dinâmica de memória e classes auto-referenciais para criar estruturas de dados dinâmicas.

19.4 Listas encadeadas

A *lista encadeada* é uma coleção linear (isto é, uma sequência) de objetos de classe auto-referencial, chamados de *nodos*, conectados por *links* de referência – daí o termo lista “encadeada”. O programa acessa uma lista encadeada através de uma referência ao primeiro nodo da lista. Ele acessa cada nodo subsequente através do membro de referência de *link* armazenado no nodo anterior. Por convenção, a referência de *link* no último nodo de uma lista é configurada como `null` para marcar o final da lista. Os dados são armazenados em uma lista encadeada dinamicamente – a lista cria cada nodo conforme necessário. O nodo pode conter dados de qualquer tipo, incluindo objetos de outras classes. As pilhas e filas também são estruturas de dados lineares e, como veremos, são versões limitadas de listas encadeadas. As árvores são estruturas de dados não-lineares.

As listas de dados podem ser armazenadas em *arrays*, mas as listas encadeadas fornecem várias vantagens. A lista encadeada é apropriada quando o número de elementos de dados a ser representado na estrutura de dados é imprevisível. As listas encadeadas são dinâmicas, de modo que o comprimento de uma lista pode aumentar ou diminuir conforme necessário. O tamanho de um *array* Java “convencional”, entretanto, não pode ser alterado, porque o tamanho de *array* é fixado no momento em que o programa cria o *array*. Os *arrays* “convencionais” podem ficar cheios. As listas encadeadas ficam cheias apenas quando o sistema tem memória insuficiente para satisfazer solicitações de alocação de dinâmica de memória. O pacote `java.util` contém a classe `LinkedList` para implementar e manipular listas encadeadas que crescem e encolhem durante a execução do programa. Discutiremos a classe `LinkedList` no Capítulo 21.

Dica de desempenho 19.1



Pode-se declarar um array para conter mais elementos que o número de itens esperado, mas isso pode desperdiçar memória. As listas encadeadas podem fornecer melhor utilização de memória nessas situações. As listas encadeadas permitem que o programa se adapte durante a execução.

Dica de desempenho 19.2



A inserção em uma lista encadeada é rápida – apenas duas referências têm de ser modificadas (depois que você encontrou o lugar para fazer a inserção). Todos os nodos existentes permanecem em suas posições atuais na memória.

As listas encadeadas podem ser mantidas em ordem de classificação simplesmente inserindo-se cada novo elemento no ponto adequado na lista (naturalmente, leva tempo para localizar o ponto de inserção adequado). Os elementos existentes na lista não precisam ser movidos.

Os nodos de listas encadeadas normalmente não são armazenados contiguamente na memória. Em vez disso, são logicamente contíguos. A Fig. 19.2 ilustra uma lista encadeada com vários nodos. Este diagrama apresenta uma *lista simplesmente encadeada* – cada nodo contém uma referência para o próximo nodo na lista. Freqüentemente as listas encadeadas são implementadas como listas duplamente encadeadas – cada nodo contém uma referência para o próximo nodo na lista e uma referência para o nodo anterior na lista. A classe `LinkedList` de Java (ver Capítulo 21) é uma implementação de lista duplamente encadeada.

Dica de desempenho 19.3



A inserção e a exclusão em um array ordenado podem consumir muito tempo – todos os elementos que se seguem ao elemento inserido ou excluído devem ser deslocados adequadamente.

Dica de desempenho 19.4



Os elementos de um array são armazenados contiguamente na memória. Isso permite o acesso imediato a qualquer elemento do array porque o endereço de qualquer elemento pode ser calculado diretamente como seu deslocamento desde o início do array. As listas encadeadas não possibilitam acesso tão imediato a seus elementos – um elemento só pode ser acessado percorrendo-se a lista desde seu início.

Dica de desempenho 19.5



Utilizar alocação dinâmica de memória (em vez de arrays) para estruturas de dados que crescem e encolhem durante a execução pode poupar memória. Lembre-se, entretanto, de que as referências ocupam espaço e essa alocação dinâmica de memória incorre na sobrecarga de chamadas de método.

O programa das Figs. 19.3 a 19.5 utiliza uma classe `List` para manipular uma lista de objetos de vários tipos. O método principal da classe `ListTest` (Fig. 19.5) cria uma lista de objetos, insere objetos no início da lista com o método `insertAtFront`, insere objetos no final da lista com o método `insertAtBack`, exclui objetos do início da lista com o método `removeFromFront` e exclui objetos do final da lista com o método `removeFromBack`. Depois de cada operação de inserção e exclusão, `ListTest` chama o método `print` de `List` para exibir o conteúdo atual da lista. Uma discussão detalhada do programa vem a seguir. Se for feita uma tentativa de remover um item de uma lista vazia, ocorre uma `EmptyListException` (Fig. 19.4).

O programa da Fig. 19.3 consiste em quatro classes – `ListNode` (Fig. 19.3), `List` (Fig. 19.3), `EmptyListException` (Fig. 19.4) e `ListTest` (Fig. 19.5). As classes `List` e `ListNode` são colocadas no pacote `com.deitel.jhttp4.ch19`, de modo que possam ser reutilizadas ao longo deste capítulo. Encapsulada em cada objeto `List` está uma lista encadeada de objetos `ListNode`. A classe `ListNode` (linhas 6 a 38 da Fig. 19.3) consiste nos membros com acesso de pacote `data` e `nextNode`. O membro `data` de `ListNode` pode fazer referência a qualquer `Object`. O membro `nextNode` de `ListNode` armazena uma referência para o próximo objeto `ListNode` na lista encadeada.

[Nota: muitas das classes neste capítulo estão definidas no pacote `com.deitel.jhttp4.ch19`. Cada uma destas classes deve ser compilada com a opção de linha de comando `-d` para `javac`. Ao compilar as classes que não estão neste pacote e executar os programas, assegure-se de usar a opção `-classpath` para `javac` e `java`, respectivamente.]

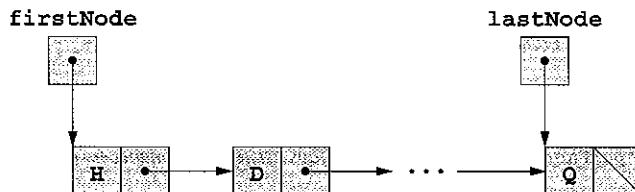


Fig. 19.2 Uma representação gráfica de uma lista encadeada.

```

1 // Fig. 19.3: List.java
2 // Definições da classe ListNode e da classe List
3 package com.deitel.jhttp4.ch19;
4
5 // classe para representar um nodo em uma lista
6 class ListNode {
7
8     // membros com acesso de pacote; List pode acessá-los diretamente
9     Object data;
10    ListNode nextNode;
11
12    // construtor para criar um ListNode que faz referência a um objeto
13    ListNode( Object object )
14    {
15        this( object, null );
16    }
17
18    // construtor para criar um ListNode que faz referência a um objeto
19    // e para o próximo ListNode em List
20    ListNode( Object object, ListNode node )
21    {
  
```

Fig. 19.3 Definições da classe `ListNode` e da classe `List` (parte 1 de 4).

```

22     data = object;
23     nextNode = node;
24 }
25
26 // devolve Object neste nodo
27 Object getObject()
28 {
29     return data;
30 }
31
32 // obtém próximo nodo
33 ListNode getNext()
34 {
35     return nextNode;
36 }
37
38 } // fim da classe ListNode
39
40 // definição da classe List
41 public class List {
42     private ListNode firstNode;
43     private ListNode lastNode;
44     private String name; // String como "list", usado na impressão
45
46     // constrói uma List vazia com um nome
47     public List( String string )
48     {
49         name = string;
50         firstNode = lastNode = null;
51     }
52
53     // constrói uma List vazia, com "list" como o nome
54     public List()
55     {
56         this( "list" );
57     }
58
59     // Insere Object no início de List. Se List estiver vazia,
60     // firstNode e lastNode farão referência ao mesmo objeto.
61     // Caso contrário, firstNode faz referência ao novo nodo.
62     public synchronized void insertAtFront( Object insertItem )
63     {
64         if ( isEmpty() )
65             firstNode = lastNode = new ListNode( insertItem );
66
67         else
68             firstNode = new ListNode( insertItem, firstNode );
69     }
70
71     // Insere Object no fim da List. Se a List estiver vazia,
72     // firstNode e lastNode farão referência ao mesmo Object.
73     // Caso contrário, o nextNode de lastNode faz referência ao novo nodo.
74     public synchronized void insertAtBack( Object insertItem )
75     {
76         if ( isEmpty() )
77             firstNode = lastNode = new ListNode( insertItem );
78
79         else
80             lastNode = lastNode.nextNode =
81                 new ListNode( insertItem );

```

Fig. 19.3 Definições da classe ListNode e da classe List (parte 2 de 4).

```

82     }
83
84     // remove o primeiro nodo de List
85     public synchronized Object removeFromFront()
86         throws EmptyListException
87     {
88         Object removeItem = null;
89
90         // dispara exceção se a List estiver vazia
91         if ( isEmpty() )
92             throw new EmptyListException( name );
93
94         // recupera os dados que estão sendo movidos
95         removeItem = firstNode.data;
96
97         // reinicializa as referências firstNode e lastNode
98         if ( firstNode == lastNode )
99             firstNode = lastNode = null;
100
101        else
102            firstNode = firstNode.nextNode;
103
104        // devolve os dados do nodo removido
105        return removeItem;
106    }
107
108    // remove o último nodo de List
109    public synchronized Object removeFromBack()
110        throws EmptyListException
111    {
112        Object removeItem = null;
113
114        // dispara uma exceção se a List estiver vazia
115        if ( isEmpty() )
116            throw new EmptyListException( name );
117
118        // recupera os dados que estão sendo removidos
119        removeItem = lastNode.data;
120
121        // reinicializa as referências firstNode e lastNode
122        if ( firstNode == lastNode )
123            firstNode = lastNode = null;
124
125        else {
126
127            // localiza o novo último nodo
128            ListNode current = firstNode;
129
130            // repete o laço enquanto o nodo atual não faz referência ao lastNode
131            while ( current.nextNode != lastNode )
132                current = current.nextNode;
133
134            // current é o novo lastNode
135            lastNode = current;
136            current.nextNode = null;
137        }
138
139        // devolve os dados do nodo removido
140        return removeItem;
141    }

```

Fig. 19.3 Definições da classe `ListNode` e da classe `List` (parte 3 de 4).

```

142
143     // devolve true se a List estiver vazia
144     public synchronized boolean isEmpty()
145     {
146         return firstNode == null;
147     }
148
149     // envia para a saída o conteúdo da List
150     public synchronized void print()
151     {
152         if ( isEmpty() ) {
153             System.out.println( "Empty " + name );
154             return;
155         }
156
157         System.out.print( "The " + name + " is: " );
158
159         ListNode current = firstNode;
160
161         // enquanto não é o fim da lista, envia os dados do nodo atual para a saída
162         while ( current != null ) {
163             System.out.print( current.data.toString() + " " );
164             current = current.nextNode;
165         }
166
167         System.out.println( "\n" );
168     }
169
170 } // fim da classe List

```

Fig. 19.3 Definições da classe `ListNode` e da classe `List` (parte 4 de 4).

A classe `List` contém os membros `private firstNode` (uma referência ao primeiro `ListNode` em uma `List`) e `lastNode` (uma referência ao último `ListNode` em uma `List`). Os construtores (linhas 47 a 51 e 54 a 57) inicializam ambas as referências como `null`. Os métodos mais importantes da classe `List` são os métodos `synchronized insertAtFront` (linhas 62 a 69), `insertAtBack` (linhas 74 a 82), `removeFromFront` (linhas 85 a 106) e `removeFromBack` (linhas 109 a 141). Esses métodos são declarados `synchronized` para que os objetos `List` possam ser *seguros para múltiplas threads* quando são utilizados em um programa com múltiplas `threads`. Se uma `thread` estiver modificando o conteúdo de uma `List`, nenhuma outra `thread` pode modificar o mesmo objeto `List` ao mesmo tempo. O método `isEmpty` (linhas 144 a 147) é um *método de predicado* que determina se a lista está vazia (isto é, a referência ao primeiro nodo da lista é `null`). Os métodos de predicado normalmente testam uma condição e não modificam o objeto sobre o qual eles são chamados. Se a lista estiver vazia, `isEmpty` devolve `true`; caso contrário, `false`. O método `print` (linhas 150 a 168) exibe o conteúdo da lista. Tanto `isEmpty` quanto `print` são também métodos `synchronized`.

```

1 // Fig. 19.4: EmptyListException.java
2 // Definição da classe EmptyListException
3 package com.deitel.jhtp4.ch19;
4
5 public class EmptyListException extends RuntimeException {
6
7     // inicializa uma EmptyListException
8     public EmptyListException( String name )
9     {
10        super( "The " + name + " is empty" );

```

Fig. 19.4 Definição da classe `EmptyListException` (parte 1 de 2).

```

11    }
12
13 } // fim da classe EmptyListException

```

Fig. 19.4 Definição da classe `EmptyListException` (parte 2 de 2).

```

1 // Fig. 19.5: ListTest.java
2 // Classe ListTest
3
4 // Pacotes Deitel
5 import com.deitel.jhtp4.ch19.List;
6 import com.deitel.jhtp4.ch19.EmptyListException;
7
8 public class ListTest {
9
10    // testa a classe List
11    public static void main( String args[] )
12    {
13        List list = new List(); // cria o contêiner List
14
15        // cria objetos para armazenar em List
16        Boolean bool = Boolean.TRUE;
17        Character character = new Character( '$' );
18        Integer integer = new Integer( 34567 );
19        String string = "hello";
20
21        // usa os métodos de inserção de List
22        list.insertAtFront( bool );
23        list.print();
24        list.insertAtFront( character );
25        list.print();
26        list.insertAtBack( integer );
27        list.print();
28        list.insertAtBack( string );
29        list.print();
30
31        // usa os métodos de remoção de List
32        Object removedObject;
33
34        // remove objetos da lista; imprime após cada remoção
35        try {
36            removedObject = list.removeFromFront();
37            System.out.println(
38                removedObject.toString() + " removed" );
39            list.print();
40
41            removedObject = list.removeFromFront();
42            System.out.println(
43                removedObject.toString() + " removed" );
44            list.print();
45
46            removedObject = list.removeFromBack();
47            System.out.println(
48                removedObject.toString() + " removed" );
49            list.print();
50
51            removedObject = list.removeFromBack();

```

Fig. 19.5 Manipulando uma lista encadeada (parte 1 de 2).

```

52         System.out.println(
53             removedObject.toString() + " removed");
54         list.print();
55     }
56
57     // processa exceção se a List está vazia quando
58     // se faz tentativa de remover um item
59     catch ( EmptyListException emptyListException ) {
60         emptyListException.printStackTrace();
61     }
62
63 } // fim do método main
64
65 } // fim da classe ListTest

```

```

The list is: true
The list is: $ true
The list is: $ true 34567
The list is: $ true 34567 hello
$ removed
The list is: true 34567 hello
true removed
The list is: 34567 hello
hello removed
The list is: 34567
34567 removed
Empty list

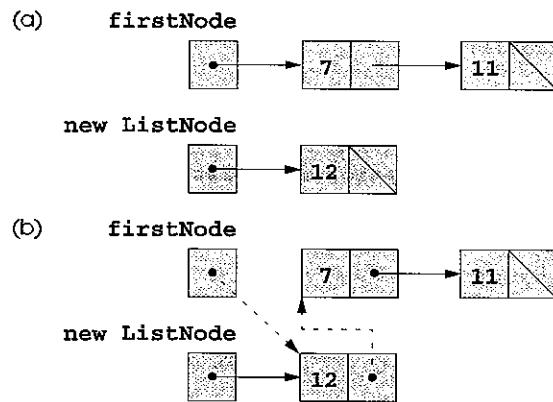
```

Fig. 19.5 Manipulando uma lista encadeada (parte 2 de 2).

Nas próximas páginas, discutimos cada um dos métodos da classe **List** (Fig. 19.3) em detalhes. O método **insertAtFront** (linhas 62 a 69 da Fig. 19.3) coloca um novo nodo no início da lista. O método consiste em vários passos (a Fig. 19.6 ilustra a operação):

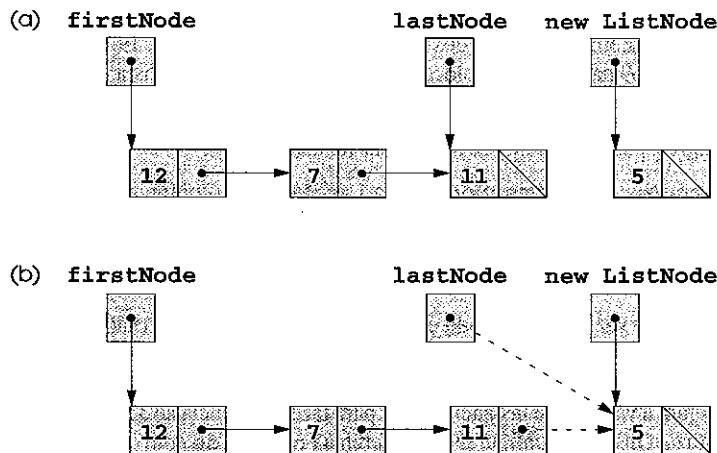
1. Chamar **isEmpty** para determinar se a lista está vazia (linha 64).
2. Se a lista estiver vazia, tanto **firstNode** como **lastNode** são configurados como o **ListNode** alocado com **new** e inicializados com **insertItem** (linha 65). O construtor **ListNode** nas linhas 13 a 16 chama o construtor **ListNode** nas linhas 20 a 24 para configurar a variável de instância **data** para fazer referência ao **insertItem** passado como argumento e configurar a referência **nextNode** como **null**.
3. Se a lista não estiver vazia, o novo nodo é “encadeado” ou “threaded” (não o confunda com *multithreading*) na lista, configurando **firstNode** como novo objeto **ListNode** e inicializando aquele objeto com **insertItem** e **firstNode** (linha 68). Quando o construtor **ListNode** (linhas 20 a 24) é executado, ele configura a variável de instância **data** para fazer referência ao **insertItem** recebido como argumento e realiza a inserção configurando a referência **nextNode** com o **ListNode** recebido como argumento.

Na Fig. 19.6, a parte (a) da figura mostra a lista e o novo nodo durante a operação **insertAtFront** e antes de o programa encadear o novo nodo na lista. As setas pontilhadas na parte (b) ilustram o passo 3 da operação **insertAtFront** que permite ao nodo que contém 12 tornar-se o novo início da lista.

**Fig. 19.6** A operação `insertAtFront`.

O método `insertAtBack` (linhas 74 a 82 da Fig. 19.3) coloca um novo nodo no final da lista. O método consiste em várias etapas (a Fig. 19.7 ilustra a operação):

1. Chamar `isEmpty` para determinar se a lista está vazia (linha 76).
2. Se a lista estiver vazia, tanto `firstNode` como `lastNode` são configurados como o `ListNode` alocado com `new` e inicializados com `insertItem` (linha 77). O construtor `ListNode` nas linhas 13 a 16 chama o construtor `ListNode` nas linhas 20 a 24 para configurar a variável de instância `data` para fazer referência ao `insertItem` passado como argumento e configura a referência `nextNode` como `null`.

**Fig. 19.7** Uma representação gráfica da operação `insertAtBack`.

3. Se a lista não estiver vazia, o novo nodo é encadeado na lista configurando `lastNode` e `lastNode.nextNode` como o `ListNode` que foi alocado com `new` e inicializado com `insertItem` (linhas 80 a 81). Quando o construtor `ListNode` (linhas 13 a 16) é executado, ele configura a variável de instância `data` para fazer referência ao `insertItem` passado como argumento e configura a referência `nextNode` como `null`.

Na Fig. 19.7, a parte (a) mostra a lista e o novo nodo durante a operação `insertAtBack` e antes que o programa encadeie o novo nodo na lista. As setas pontilhadas na parte (b) ilustram as etapas do método `insertAtBack` que permitem que um novo nodo seja adicionado ao fim de uma lista que não está vazia.

O método `removeFromFront` (linhas 85 a 106 da Fig. 19.3) remove o nodo do início da lista e devolve uma referência para os dados removidos. O método dispara uma `EmptyListException` (linhas 91 e 92) se a lista está vazia quando o programa chama este método. Caso contrário, o método devolve uma referência para os dados removidos. O método consiste em várias etapas (ilustradas na Fig. 19.8):

1. Atribuir `firstNode.data` (os dados que estão sendo removidos da lista) à referência `removeItem` (linha 95).
2. Se os objetos aos quais `firstNode` e `lastNode` faz referência forem iguais (linha 98), a lista tem apenas um elemento antes da tentativa de remoção. Neste caso, o método configura `firstNode` e `lastNode` como `null` (linha 99) para “desencadear” (remover) o nodo da lista (deixando a lista vazia).
3. Se a lista tiver mais de um nodo antes da remoção, então o método deixa a referência `lastNode` como está e simplesmente atribui `firstNode.next` à referência `firstNode` (linha 102). Assim, `firstNode` faz referência ao nodo que era o segundo nodo antes da chamada para `removeFromFront`.
4. Devolve a referência `removeItem` (linha 105).

Na Fig. 19.8, a parte (a) ilustra a lista antes da operação de exclusão. A parte (b) mostra as manipulações reais de referências.

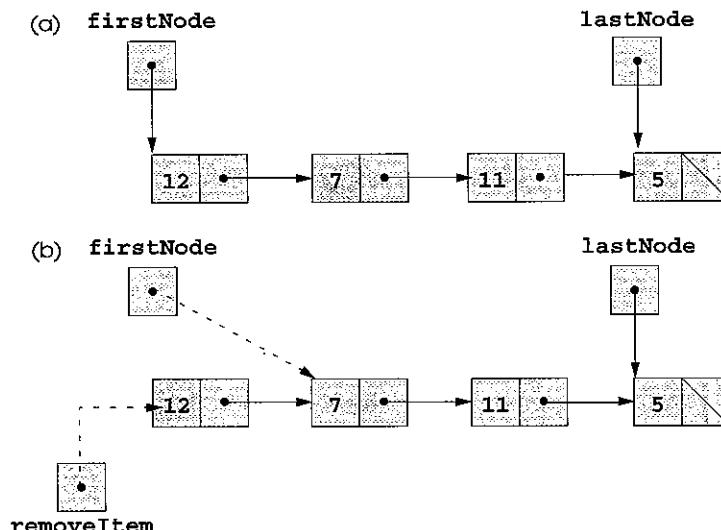


Fig. 19.8 Uma representação gráfica da operação `removeFromFront`.

O método `removeFromBack` (linhas 109 a 141 da Fig. 19.3) remove o último nodo de uma lista e devolve uma referência para os dados removidos. O método dispara uma `EmptyListException` (linhas 115 e 116) se a

lista está vazia quando o programa chama este método. O método consiste em várias etapas (a Fig. 19.9 ilustra a operação):

1. Atribuir `lastNode.data` (os dados que estão sendo removidos da lista) para a referência `removeItem`.
2. Se os objetos aos quais `firstNode` e `lastNode` faz referência forem iguais (linha 122), a lista tem apenas um elemento antes da tentativa de remoção. Nesta caso, o método configura `firstNode` e `lastNode` como `null` (linha 123) para “desencadear” (remover) esse nodo da lista (deixando a lista vazia).
3. Se a lista tiver mais de um nodo antes da remoção, cria a referência `ListNode current` e a inicializa como `firstNode`.
4. Agora “percorre a lista” com `current` até ela fazer referência ao nodo antes do último nodo. O laço `while` (linhas 131 e 132) atribui `current.nextNode` a `current` enquanto `current.nextNode` não for `lastNode`.
5. Depois de localizar o antepenúltimo nodo, atribuir `current` a `lastNode` (linha 135) para desvincular o último nodo da lista.
6. Configurar o `current.nextNode` como `null` (linha 136) para terminar a lista no nodo atual.
7. Devolver a referência `removeItem` (linha 140).

Na Fig. 19.9, a parte (a) mostra a lista antes da operação de remoção. A parte (b) mostra as manipulações reais de referências.

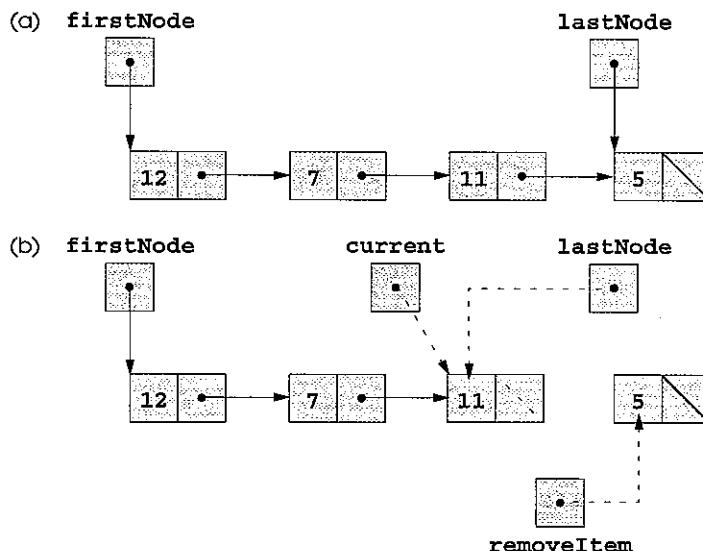


Fig. 19.9 Uma representação gráfica da operação `removeFromBack`.

O método `print` (linhas 150 a 168) primeiro determina se a lista está vazia (linhas 152 a 155). Se estiver, `print` exibe uma mensagem indicando que a lista está vazia e devolve o controle ao método que fez a chamada. Caso contrário, `print` imprime os dados da lista. A linha 159 cria a referência `current` de `ListNode` e a inicializa com `firstNode`. Enquanto `current` não é `null`, existem mais itens na lista e, portanto, a linha 163 envia para a saída a representação como `String` de `current.data`. A linha 164 move o próximo nodo na lista,

atribuindo o valor de `current.nextNode` a `current`. O algoritmo de impressão é idêntico às listas encadeadas, pilhas e filas.

19.5 Pilhas

A *pilha* é uma versão limitada de uma lista encadeada – novos nodos podem ser adicionados a uma pilha e removidos de uma pilha apenas na parte superior (topo). Por essa razão, a pilha é conhecida como uma estrutura de dados *primeiro a entrar, primeiro a sair (last-in, first-out – LIFO)*. O membro de *link* no nodo inferior (isto é, o último) da pilha é configurado como nulo para indicar a base da pilha.



Erro comum de programação 19.2

Não configurar o link no nodo inferior de uma pilha como null é um erro comum de lógica.

Os principais métodos utilizados para manipular uma pilha são `push` e `pop`. O método `push` adiciona um novo nodo ao topo da pilha. O método `pop` remove um nodo do topo da pilha e devolve o objeto `data` do nodo removido da pilha.

As pilhas têm muitas aplicações interessantes. Por exemplo, quando se faz uma chamada de método, o método chamado deve saber retornar para seu chamador, assim o endereço de retorno é adicionado à *pilha de execução do programa*. Se ocorre uma série de chamadas de métodos, os valores sucessivos de retorno são adicionados à pilha na ordem último a entrar, primeiro a sair, de modo que cada método possa retornar para seu chamador. As pilhas suportam chamadas de método recursivas, da mesma maneira que as chamadas de método não-recursivas convencionais.

A pilha de execução do programa contém o espaço criado para variáveis locais a cada invocação de um método durante a execução de um programa. Quando o método retorna para seu chamador, o espaço para variáveis locais desse método é removido da pilha e essas variáveis não são mais conhecidas para o programa.

As pilhas também são utilizadas por compiladores no processo de avaliação de expressões aritméticas e geração de código de linguagem de máquina para processar expressões. Os exercícios neste capítulo exploram várias aplicações de pilhas, incluindo seu uso para desenvolver um compilador completo. O pacote `java.util` da API Java contém a classe `Stack` para implementar e manipular pilhas que podem crescer e encolher durante a execução do programa. Discutiremos a classe `Stack` no Capítulo 20.

Tiramos proveito do íntimo relacionamento entre listas e pilhas para implementar uma classe de pilha principalmente através da reutilização de uma classe de lista. Utilizamos duas formas diferentes de reutilização. Primeiro, implementamos a classe de pilha por herança da classe `List` da Fig. 19.3. A seguir, implementamos uma classe de pilha de idêntico comportamento, através de composição, pela inclusão de um objeto `List` como membro `private` de uma classe de pilha. As estruturas de dados de lista, pilha e fila neste capítulo são implementadas para armazenar referências `Object` a fim de incentivar a reutilização futura. Portanto, qualquer tipo de objeto pode ser armazenado em uma lista, uma pilha ou uma fila.

O aplicativo das Figs. 19.10 e 19.11 cria uma classe de pilha por herança da classe `List` da Fig. 19.3. Queremos que a pilha tenha os métodos `push`, `pop`, `isEmpty` e `print`. Esses são essencialmente os métodos `insertAtFront`, `removeFromFront`, `isEmpty` e `print` da classe `List`. Naturalmente, a classe `List` contém outros métodos (como `insertAtBack` e `removeFromBack`) que preferiríamos não tornar acessíveis pela interface pública à classe de pilha. É importante lembrar que todos os métodos na interface pública da classe `List` também são métodos `public` da classe derivada `StackInheritance` (Fig. 19.10). Quando implementamos os métodos da pilha, fazemos cada método de `StackInheritance` chamar o método `List` adequado – o método `push` chama `insertAtFront`, o método `pop` chama `removeFromFront`. A classe `StackInheritance` é definida como parte do pacote `com.deitel.jhtp4.ch19` para fins de reutilização. Note que `StackInheritance` não importa `List`, porque ambas as classes estão no mesmo pacote.

```

1 // Fig. 19.10: StackInheritance.java
2 // Derivada da classe List
3 package com.deitel.jhtp4.ch19;

```

Fig. 19.10 Classe `StackInheritance` estende a classe `List` (parte 1 de 2).

```

4
5  public class StackInheritance extends List {
6
7      // constrói pilha
8      public StackInheritance()
9      {
10         super( "stack" );
11     }
12
13     // adiciona objeto à pilha
14     public synchronized void push( Object object )
15     {
16         insertAtFront( object );
17     }
18
19     // remove objeto da pilha
20     public synchronized Object pop() throws EmptyListException
21     {
22         return removeFromFront();
23     }
24
25 } // fim da classe StackInheritance

```

Fig. 19.10 Classe `StackInheritance` estende a classe `List` (parte 2 de 2).

O método `main` da classe `StackInheritanceTest` (Fig. 19.11) utiliza a classe `StackInheritance` para instanciar uma pilha de `Objects` denominada `stack`. O programa coloca na pilha (linhas 22, 24, 26 e 28) um objeto `Boolean` contendo `true`, um objeto `Character` que contém `$`, um objeto `Integer` que contém `34567` e um objeto `String` que contém `hello` e depois os remove de `Stack`. As linhas 37 a 42 removem os objetos da pilha com um laço `while` infinito. Quando não restam mais objetos a remover, o método `pop` dispara uma `EmptyListException` e o programa exibe o monitoramento da pilha que mostra os métodos na pilha de execução do programa no momento em que a exceção ocorreu. Note que o programa usa o método `print` (herdado de `List`) para exibir o conteúdo da pilha.

```

1 // Fig. 19.11: StackInheritanceTest.java
2 // Classe StackInheritanceTest
3
4 // Pacotes Deitel
5 import com.deitel.jhtp4.ch19.StackInheritance;
6 import com.deitel.jhtp4.ch19.EmptyListException;
7
8 public class StackInheritanceTest {
9
10    // testa a classe StackInheritance
11    public static void main( String args[] )
12    {
13        StackInheritance stack = new StackInheritance();
14
15        // cria objetos para armazenar na pilha
16        Boolean bool = Boolean.TRUE;
17        Character character = new Character( '$' );
18        Integer integer = new Integer( 34567 );
19        String string = "hello";
20
21        // usa o método push
22        stack.push( bool );
23        stack.print();

```

Fig. 19.11 Um programa simples com pilha (parte 1 de 2).

```

24     stack.push( character );
25     stack.print();
26     stack.push( integer );
27     stack.print();
28     stack.push( string );
29     stack.print();
30
31     // remove itens da pilha
32     try {
33
34         // usa o método pop
35         Object removedObject = null;
36
37         while ( true ) {
38             removedObject = stack.pop();
39             System.out.println( removedObject.toString() +
40                     " popped" );
41             stack.print();
42         }
43     }
44
45     // captura exceção se a pilha estiver vazia quando retira item
46     catch ( EmptyListException emptyListException ) {
47         emptyListException.printStackTrace();
48     }
49
50 } // fim do método main
51
52 } // fim da classe StackInheritanceTest

```

```

The stack is: true
The stack is: $ true
The stack is: 34567 $ true
The stack is: hello 34567 $ true
hello popped
The stack is: 34567 $ true
34567 popped
The stack is: $ true
$ popped
The stack is: true
true popped
Empty stack
com.deitel.jhttp4.ch19.EmptyListException: The stack is empty
    at com.deitel.jhttp4.ch19.List.removeFromFront(List.java:92)
    at com.deitel.jhttp4.ch19.StackInheritance.pop(
        StackInheritance.java:22)
    at StackInheritanceTest.main(StackInheritanceTest.java:38)

```

Fig. 19.11 Um programa simples com pilha (parte 2 de 2).

Outra maneira de implementar uma classe de pilha é reutilizando uma classe de lista por composição. A classe na Fig. 19.12 utiliza um objeto **private** da classe **List** (linha 6) na definição da classe **StackComposition**.

A composição permite ocultar os métodos da classe `List` que não devem estar na interface `public` de nossa pilha, fornecendo métodos públicos de interface apenas para os métodos `List` necessários. Essa técnica de implementar cada método de pilha como uma chamada para um método de `List` é chamada de *delegação* – o método de pilha invocado *delega* a chamada ao método de `List` adequado. Em particular, `StackCompositionTest` delega chamadas aos métodos `insertAtFront`, `removeFromFront`, `isEmpty` e `print` de `List`. Neste exemplo, não mostramos a classe `StackCompositionTest` porque a única diferença neste exemplo é que trocamos o tipo da pilha de `StackInheritance` para `StackComposition`. Se você executar o aplicativo a partir do código no CD que acompanha este livro, você verá que a saída é idêntica.

```

1 // Fig. 19.12: StackComposition.java
2 // Definição da classe StackComposition com o objeto List composto
3 package com.deitel.jhtp4.ch19;
4
5 public class StackComposition {
6     private List stackList;
7
8     // constrói pilha
9     public StackComposition()
10    {
11        stackList = new List( "stack" );
12    }
13
14     // adiciona objeto à pilha
15     public synchronized void push( Object object )
16    {
17        stackList.insertAtFront( object );
18    }
19
20     // remove objeto da pilha
21     public synchronized Object pop() throws EmptyListException
22    {
23        return stackList.removeFromFront();
24    }
25
26     // determina se a pilha está vazia
27     public synchronized boolean isEmpty()
28    {
29        return stackList.isEmpty();
30    }
31
32     // envia conteúdo da pilha para a saída
33     public synchronized void print()
34    {
35        stackList.print();
36    }
37
38 } // fim da classe StackComposition

```

Fig. 19.12 Uma classe de pilha comum usando composição.

19.6 Filas

Outra estrutura de dados comum é a *fila*. A fila é semelhante a uma fila de caixa em um supermercado – a primeira pessoa na fila é atendida primeiro e os outros clientes entram na fila apenas no final e esperam ser atendidos. Os nós da fila são removidos apenas do *início* (ou *cabeça*) da fila e são inseridos somente no *final* (ou *cauda*) da fila. Por essa razão, trata-se uma fila como uma estrutura de dados *primeiro a entrar, primeiro a sair* (*first-in, first-out – FI-FO*). As operações de inserção e remoção são conhecidas como `enqueue` (enfileirar) e `dequeue` (desenfileirar).

As filas têm muitas aplicações em sistemas de computador. A maioria dos computadores tem apenas um único processador, assim somente um aplicativo pode ser atendido de cada vez. Os pedidos dos outros aplicativos são colocados em uma fila. O pedido no início da fila é o próximo a ser atendido. Cada pedido avança gradualmente para o início da fila à medida que os aplicativos vão sendo atendidos.

As filas também são utilizadas para suportar *spooling* de impressão. Um ambiente multiusuário pode ter somente uma única impressora. Muitos usuários podem estar gerando saídas para serem impressas. Se a impressora estiver ocupada, outras pessoas podem continuar a mandar serviços para impressão. Estas são colocadas no “*spool*” em disco (de maneira muito parecida a como um fio é enrolado em um carretel) no qual esperam em uma fila até a impressora ficar disponível.

Os pacotes de informações também esperam em filas em redes de computadores. Toda vez que um pacote chega em um nodo da rede, ele deve ser roteado para o próximo nodo na rede ao longo do caminho até o destino final do pacote. O nodo de roteamento roteia um pacote por vez, assim pacotes adicionais são enfileirados até o roteador conseguir roteá-los.

O servidor de arquivos em uma rede de computadores trata as solicitações de acesso a arquivos de muitos clientes por toda a rede. Os servidores têm uma capacidade limitada para atender às solicitações de clientes. Quando essa capacidade é excedida, as solicitações dos clientes esperam em filas.

O aplicativo das Figs. 19.13 e 19.14 cria uma fila por herança da classe **List** (Fig. 19.3). Queremos que a classe **QueueInheritance** (Fig. 19.3) tenha os métodos **enqueue** e **dequeue**, **isEmpty** e **print**. Observe que esses são essencialmente os métodos **insertAtBack** e **removeFromFront** de **List**. Naturalmente, a classe **List** contém outros métodos (isto é, **insertAtFront** e **removeFromBack**), que não gostaríamos de tornar acessíveis através da interface **public** da fila. Lembre-se de que todos os métodos da interface pública da classe **List** também são métodos **public** da subclasse **QueueInheritance**. Na implementação da fila, cada método da classe **QueueInheritance** chama o método de **List** apropriado – o método **enqueue** chama **insertAtBack** e o método **dequeue** chama **removeFromFront**. A classe **QueueInheritance** é definida no pacote **com.deitel.jhttp4.ch19** para fins de reutilização.



Erro comum de programação 19.3

Não configurar o link no último nodo de uma fila para null é um erro comum de lógica.

```

1 // Fig. 19.13: QueueInheritance.java
2 // Classe QueueInheritance estende a classe List
3
4 // Pacotes Deitel
5 package com.deitel.jhttp4.ch19;
6
7 public class QueueInheritance extends List {
8
9     // constrói a fila
10    public QueueInheritance()
11    {
12        super( "queue" );
13    }
14
15    // adiciona objeto à fila
16    public synchronized void enqueue( Object object )
17    {
18        insertAtBack( object );
19    }
20
21    // remove objeto da fila
22    public synchronized Object dequeue() throws EmptyListException
23    {
24        return removeFromFront();
25    }
26
27 } // fim da classe QueueInheritance

```

Fig. 19.13 Classe **QueueInheritance** que estende a classe **List**.

O método `main` (Fig. 19.14) da classe `QueueInheritanceTest` utiliza a classe `QueueInheritance` para instanciar uma fila de `Objects` chamada `queue`. As linhas 22, 24, 26 e 28 colocam na fila um objeto `Boolean` que contém `true`, um objeto `Character` que contém `$`, um objeto `Integer` que contém `34567` e um objeto `String` que contém `hello`. As linhas 37 a 42 utilizam um laço `while` infinito para retirar da fila os objetos, na ordem primeiro a entrar, primeiro a sair. Quando não há mais objetos para desenfileirar, o método `dequeue` dispara uma `EmptyListException` e o programa exibe o monitoramento da pilha de exceções.

```

1 // Fig. 19.14: QueueInheritanceTest.java
2 // Classe QueueInheritanceTest
3
4 // Pacotes Deitel
5 import com.deitel.jhtp4.ch19.QueueInheritance;
6 import com.deitel.jhtp4.ch19.EmptyListException;
7
8 public class QueueInheritanceTest {
9
10    // testa a classe QueueInheritance
11    public static void main( String args[] )
12    {
13        QueueInheritance queue = new QueueInheritance();
14
15        // cria objetos para armazenar na fila
16        Boolean bool = Boolean.TRUE;
17        Character character = new Character( '$' );
18        Integer integer = new Integer( 34567 );
19        String string = "hello";
20
21        // usa o método enqueue
22        queue.enqueue( bool );
23        queue.print();
24        queue.enqueue( character );
25        queue.print();
26        queue.enqueue( integer );
27        queue.print();
28        queue.enqueue( string );
29        queue.print();
30
31        // remove objetos da fila
32        try {
33
34            // usa o método dequeue
35            Object removedObject = null;
36
37            while ( true ) {
38                removedObject = queue.dequeue();
39                System.out.println( removedObject.toString() +
40                    " dequeued" );
41                queue.print();
42            }
43        }
44
45        // processa exceção se a fila está vazia quando o item é removido
46        catch ( EmptyListException emptyListException ) {
47            emptyListException.printStackTrace();
48        }
49
50    } // fim do método main
51
52 } // fim da classe QueueInheritanceTest

```

Fig. 19.14 Processando uma fila (parte 1 de 2).

```

The queue is: true
The queue is: true $
The queue is: true $ 34567
The queue is: true $ 34567 hello
true dequeued
The queue is: $ 34567 hello
$ dequeued
The queue is: 34567 hello
34567 dequeued
The queue is: hello
hello dequeued
Empty queue
com.deitel.jhttp4.ch19.EmptyListException: The queue is empty
    at com.deitel.jhttp4.ch19.List.removeFromFront(List.java:92)
    at com.deitel.jhttp4.ch19.QueueInheritance.dequeue(
        QueueInheritance.java:24)
    at QueueInheritanceTest.main(QueueInheritanceTest.java:38)

```

Fig. 19.14 Processando uma fila (parte 2 de 2).

19.7 Árvores

Listas encadeadas, pilhas e filas são *estruturas de dados lineares* (isto é, *seqüências*). A árvore é uma estrutura de dados bidimensional, não-linear, com propriedades especiais. Os nodos da árvore contêm dois ou mais *links*. Esta seção discute as *árvores binárias* (Fig. 19.15) – as árvores cujos nodos contêm dois *links* (dos quais nenhum, um ou ambos podem ser `null`). O *nodo-raiz* é o primeiro nodo em uma árvore. Cada *link* no nodo-raiz faz referência a um *filho*. O *filho esquerdo* é o primeiro nodo na *subárvore esquerda* e o *filho direito* é o primeiro nodo na *subárvore direita*. Os filhos de um nodo específico são chamados de *irmãos*. Um nodo sem filhos é chamado de *nodo-folha*. Os cientistas da computação normalmente desenham árvores que vão do nodo-raiz para baixo – exatamente o oposto da maneira como a maioria das árvores cresce na natureza.



Erro comum de programação 19.4

Não configurar como null os links em nodos-folha de uma árvore é um erro comum de lógica.

Em nosso exemplo de árvore binária, criamos uma árvore binária especial chamada de *árvore de pesquisa binária*. Uma árvore de pesquisa binária (sem valores duplicados de nodo) tem a característica de que os valores em qualquer subárvore esquerda são menores que o valor em seu nodo pai e os valores em qualquer subárvore direita são maiores que o valor em seu nodo pai. A Fig. 19.16 ilustra uma árvore de pesquisa binária com 12 valores inteiros. Observe que a forma da árvore de pesquisa binária que corresponde a um conjunto de dados pode variar, dependendo da ordem em que os valores são inseridos na árvore.

O aplicativo das Figs. 19.17 e 19.18 cria uma árvore de pesquisa binária de inteiros e a percorre (isto é, passa por todos os seus nodos) de três maneiras – utilizando *travessias recursivas na ordem*, na *pré-ordem* e na *pós-ordem*. O programa gera 10 números aleatórios e insere cada um na árvore. A classe `Tree` é definida no pacote `com.deitel.jhttp4.ch19` para fins de reutilização.

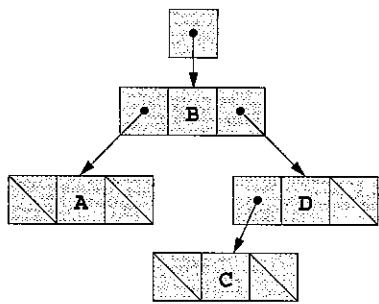


Fig. 19.15 Uma representação gráfica de uma árvore binária.

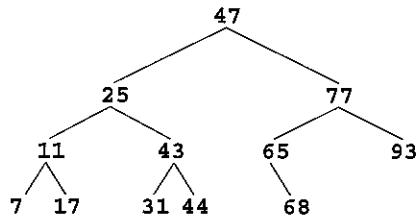


Fig. 19.16 Uma árvore de pesquisa binária que contém 12 valores.

```

1 // Fig. 19.17: Tree.java
2 // Definição da classe TreeNode e da classe Tree.
3
4 // Pacotes Deitel
5 package com.deitel.jhtp4.ch19;
6
7 // definição da classe TreeNode
8 class TreeNode {
9
10    // membros com acesso de pacote
11    TreeNode leftNode;
12    int data;
13    TreeNode rightNode;
14
15    // inicializa dados e os torna um nodo-folha
16    public TreeNode( int nodeData )
17    {
18        data = nodeData;
19        leftNode = rightNode = null; // nodo não tem filhos
20    }
21
22    // insere TreeNode em Tree que contém nodos;
  
```

Fig. 19.17 Definições de `TreeNode` e `Tree` para uma árvore de pesquisa binária (parte 1 de 3).

```

23     // ignora valores duplicados
24     public synchronized void insert( int insertValue )
25     {
26         // insere na subárvore esquerda
27         if ( insertValue < data ) {
28
29             // insere novo TreeNode
30             if ( leftNode == null )
31                 leftNode = new TreeNode( insertValue );
32
33             // continua percorrendo a subárvore esquerda
34             else
35                 leftNode.insert( insertValue );
36         }
37
38         // insere na subárvore direita
39         else if ( insertValue > data ) {
40
41             // insere novo TreeNode
42             if ( rightNode == null )
43                 rightNode = new TreeNode( insertValue );
44
45             // continua percorrendo a subárvore direita
46             else
47                 rightNode.insert( insertValue );
48         }
49
50     } // fim do método insert
51
52 } // fim da classe TreeNode
53
54 // definição da classe Tree
55 public class Tree {
56     private TreeNode root;
57
58     // constrói uma Tree de inteiros vazia
59     public Tree()
60     {
61         root = null;
62     }
63
64     // Insere um novo nodo na árvore de pesquisa binária.
65     // Se o nodo raiz for null, cria o nodo-raiz aqui.
66     // Caso contrário, chama o método insert da classe TreeNode.
67     public synchronized void insertNode( int insertValue )
68     {
69         if ( root == null )
70             root = new TreeNode( insertValue );
71
72         else
73             root.insert( insertValue );
74     }
75
76     // começa percurso na pré-ordem
77     public synchronized void preorderTraversal()
78     {
79         preorderHelper( root );
80     }
81
82     // método recursivo para fazer percurso na pré-ordem

```

Fig. 19.17 Definições de `TreeNode` e `Tree` para uma árvore de pesquisa binária (parte 2 de 3).

```

83     private void preorderHelper( TreeNode node )
84     {
85         if ( node == null )
86             return;
87
88         // envia dados do nodo para a saída
89         System.out.print( node.data + " " );
90
91         // percorre a subárvore esquerda
92         preorderHelper( node.leftNode );
93
94         // percorre a subárvore direita
95         preorderHelper( node.rightNode );
96     }
97
98     // começa o percurso na ordem
99     public synchronized void inorderTraversal()
100    {
101        inorderHelper( root );
102    }
103
104     // método recursivo para fazer o percurso na ordem
105     private void inorderHelper( TreeNode node )
106    {
107        if ( node == null )
108            return;
109
110        // percorre a subárvore da esquerda
111        inorderHelper( node.leftNode );
112
113        // envia dados do nodo para a saída
114        System.out.print( node.data + " " );
115
116        // percorre a subárvore direita
117        inorderHelper( node.rightNode );
118    }
119
120     // começa percurso na pós-ordem
121     public synchronized void postorderTraversal()
122    {
123        postorderHelper( root );
124    }
125
126     // método recursivo para fazer o percurso na pós-ordem
127     private void postorderHelper( TreeNode node )
128    {
129        if ( node == null )
130            return;
131
132        // percorre a subárvore esquerda
133        postorderHelper( node.leftNode );
134
135        // percorre a subárvore direita
136        postorderHelper( node.rightNode );
137
138        // envia dados do nodo para a saída
139        System.out.print( node.data + " " );
140    }
141
142 } // fim da classe Tree

```

Fig. 19.17 Definições de `TreeNode` e `Tree` para uma árvore de pesquisa binária (parte 3 de 3).

Vamos percorrer o programa de árvore binária. O método `main` da classe `TreeTest` (Fig. 19.18) começa instanciando um objeto `Tree` vazio e armazenando-o na referência `tree` (linha 11). O programa gera aleatoriamente 10 inteiros, e cada um deles é inserido na árvore binária por uma chamada ao método `synchronized insertNode` (linha 21). O programa, então, realiza travessias de `tree` na ordem, na pré-ordem e na pós-ordem (estas serão explicadas em breve).

```

1 // Fig. 19.18: TreeTest.java
2 // Este programa testa a classe Tree.
3 import com.deitel.jhtp4.ch19.Tree;
4
5 // Definição da class TreeTest
6 public class TreeTest {
7
8     // testa a classe Tree
9     public static void main( String args[] )
10    {
11        Tree tree = new Tree();
12        int value;
13
14        System.out.println( "Inserting the following values: " );
15
16        // insere 10 inteiros aleatórios de 0 a 99 na árvore
17        for ( int i = 1; i <= 10; i++ ) {
18            value = ( int )( Math.random() * 100 );
19            System.out.print( value + " " );
20
21            tree.insertNode( value );
22        }
23
24        // percorre a árvore na pré-ordem
25        System.out.println( "\n\nPreorder traversal" );
26        tree.preorderTraversal();
27
28        // percorre a árvore na ordem
29        System.out.println( "\n\nInorder traversal" );
30        tree.inorderTraversal();
31
32        // percorre a árvore na pós-ordem
33        System.out.println( "\n\nPostorder traversal" );
34        tree.postorderTraversal();
35        System.out.println();
36    }
37
38 } // fim da classe TreeTest

```

```

Inserting the following values:
39 69 94 47 50 72 55 41 97 73

Preorder traversal
39 69 47 41 50 55 94 72 73 97

Inorder traversal
39 41 47 50 55 69 72 73 94 97

Postorder traversal
41 55 50 47 73 72 97 94 69 39

```

Fig. 19.18 Criando e percorrendo uma árvore de pesquisa binária.

A classe **Tree** (linhas 55 a 142 da Fig. 19.17) tem um dado **private root** – uma referência ao nodo-raiz da árvore. A classe tem métodos públicos **insertNode** (linhas 67 a 74) para inserir um novo nodo na árvore e os métodos **public preorderTraversal** (linhas 77 a 80), **inorderTraversal** (linhas 99 a 102) e **postorderTraversal** (linhas 121 a 124), para iniciar percursos da árvore. Cada um desses métodos chama seu próprio método utilitário recursivo separado para realizar as operações de percurso na representação interna da árvore. O construtor **Tree** (linhas 59 a 62) inicializa **root** como **null** para indicar que a árvore está vazia.

O método **synchronized insertNode** da classe **Tree** (linhas 67 a 74) primeiro determina se a árvore está vazia. Se estiver, ele aloca um novo **TreeNode**, inicializa o nodo com o inteiro que está sendo inserido na árvore e atribui o novo nodo à referência **root** (linha 70). **insertNode** chama o método **insert** de **TreeNode** (linhas 24 a 52). Este método usa recursão para determinar a posição do nodo na árvore e insere o nodo nesta posição. *O nodo só pode ser inserido como um nodo-folha em uma árvore de pesquisa binária.*

O método **insert** de **TreeNode** compara o valor a ser inserido com o valor **data** no nodo-raiz. Se o valor de inserção é menor que o dado do nodo-raiz, o programa determina se a subárvore esquerda está vazia (linha 30). Se estiver, a linha 31 aloca um novo **TreeNode** e o inicializa com o inteiro que está sendo inserido, e atribui o novo nodo à referência **leftNode** (linha 24). Caso contrário, a linha 35 chama **insert** recursivamente para a subárvore esquerda, para inserir o valor na subárvore esquerda. Se o valor de inserção for maior que o dado do nodo-raiz, o programa determina se a subárvore direita está vazia (linha 42). Se estiver, a linha 43 aloca um novo **TreeNode** e o inicializa com o inteiro que está sendo inserido, e atribui o novo nodo à referência **rightNode**. Caso contrário, a linha 47 chama **insert** recursivamente para a subárvore direita, para inserir o valor na subárvore direita.

Os métodos **inorderTraversal**, **preorderTraversal** e **postorderTraversal** chamam os métodos auxiliares **inorderHelper** (linhas 105 a 118), **preorderHelper** (linhas 83 a 96) e **postorderHelper** (linhas 127 a 140), respectivamente, para percorrer a árvore (Fig. 19.19) e imprimir os valores dos nodos. O propósito dos métodos auxiliares na classe **Tree** é permitir ao programador iniciar um percurso sem a necessidade de primeiro obter uma referência para o nodo **root** e, depois, chamar o método recursivo com aquela referência. Os métodos **inorderTraversal**, **preorderTraversal** e **postorderTraversal** pegam simplesmente a referência **private root** e a passam ao método auxiliar adequado para iniciar um percurso da árvore.

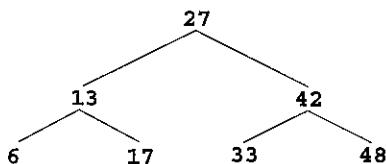


Fig. 19.19 Uma árvore de pesquisa binária.

O método **inorderHelper** (linhas 105 a 118) define os passos para um percurso em ordem. Estes passos são os seguintes:

1. Percorrer a subárvore esquerda com uma chamada a **inorderHelper** (linha 111).
2. Processar o valor no nodo (linha 114).
3. Percorrer a subárvore direita com uma chamada a **inorderHelper** (linha 117).

O percurso em ordem não processa o valor em um nodo até os valores dos nodos naquela subárvore esquerda serem processados. O percurso em ordem da árvore na Fig. 19.19 é

6 13 17 27 33 42 48

Observe que o percurso em ordem de uma árvore de pesquisa binária imprime os valores dos nodos na ordem crescente. Na verdade, o processo de criar uma árvore de pesquisa binária ordena os dados – e, assim, esse processo se chama *classificação de árvore binária*.

O método `preorderHelper` (linhas 83 a 96) define os passos para um percurso na pré-ordem. São os seguintes:

1. Processar o valor no nodo (linhas 89).
2. Percorrer a subárvore esquerda com uma chamada para `preorderHelper` (linha 92).
3. Percorrer a subárvore direita com uma chamada para `preorderHelper` (linha 95).

O percurso em pré-ordem processa o valor de cada nodo quando o nodo é visitado. Depois que o valor em um nodo dado é processado, o percurso em pré-ordem processa os valores na subárvore esquerda, depois os valores na subárvore direita. O percurso em pré-ordem da árvore na Fig. 19.19 é

27 13 6 17 42 33 48

O método `postorderHelper` (linhas 127 a 140) define os passos para um percurso em pós-ordem. São os seguintes:

1. Percorrer a subárvore esquerda com um `postorderHelper` (linha 133).
2. Percorrer a subárvore direita com um `postorderHelper` (linha 136).
3. Processar o valor no nodo (linha 139).

O percurso em pós-ordem processa o valor em cada nodo depois dos valores de todos os nodos-filhos serem processados. O `postorderTraversal` da árvore na Fig. 19.19 é

6 17 13 33 48 42 27

A árvore de pesquisa binária facilita a *eliminação de duplicatas*. Quando a árvore é criada, as tentativas de inserir um valor duplicado são reconhecidas porque uma duplicata segue as mesmas decisões “siga para a esquerda” ou “siga para a direita”, em cada comparação, que o valor original seguiu. Portanto, a duplicata acaba sendo comparada com um nodo que contém o mesmo valor. O valor duplicado simplesmente pode ser descartado nesse ponto.

Pesquisar em uma árvore binária um valor que corresponde a um valor-chave também é rápido, especialmente em árvores *compactadas*. Em uma árvore compactada, cada nível contém aproximadamente duas vezes o número de elementos que o nível anterior. A Fig. 19.19 é uma árvore binária compactada. Assim, uma árvore de pesquisa binária com n elementos tem um mínimo de $\log_2 n$ níveis. Portanto, no máximo $\log_2 n$ comparações teriam de ser feitas para localizar uma correspondência ou para determinar que nenhuma correspondência existe. Pesquisar em uma árvore de pesquisa binária (compactada) de 1.000 elementos exige pelo menos 10 comparações, porque $2^{10} > 1.000$. Pesquisar uma árvore de pesquisa binária (compactada) de 1.000.000 elementos exige no máximo 20 comparações, porque $2^{20} > 1.000.000$.

Nos exercícios, são apresentados algoritmos para várias outras operações sobre árvores binárias, como excluir um item de uma árvore binária, imprimir uma árvore binária em um formato de árvore bidimensional e realizar um *percurso na ordem de nível de uma árvore binária*. O percurso na ordem de nível de uma árvore binária percorre os nodos da árvore linha por linha, iniciando no nível do nodo-raiz. Em cada nível da árvore, os nodos são percorridos da esquerda para a direita. Outros exercícios de árvore binária incluem permitir que uma árvore de pesquisa binária contenha valores duplicados, inserir valores de *strings* em uma árvore binária e determinar quantos níveis estão contidos em uma árvore binária.

Resumo

- As estruturas de dados dinâmicas podem crescer e encolher durante a execução.
- As listas encadeadas são coleções de itens de dados “colocados em fila” – inserções e exclusões podem ser feitas em qualquer lugar em uma lista encadeada.
- As pilhas são importantes em compiladores e sistemas operacionais – as inserções e exclusões são feitas apenas em uma extremidade de uma pilha, seu topo.
- As filas representam filas de espera; as inserções são feitas na parte posterior (também conhecida como cauda) de uma fila e as exclusões são feitas na parte da frente (também conhecida como cabeça) de uma fila.
- As árvores binárias facilitam a pesquisa e a classificação de dados em alta velocidade, eliminação eficiente de itens de dados duplicados, representação de diretórios de sistemas de arquivos e compilação de expressões em linguagem de máquina.

- A classe auto-referencial contém uma referência para outro objeto do mesmo tipo de classe. Os objetos auto-referenciais podem ser encadeados entre si para formar estruturas de dados úteis, como listas, filas, pilhas e árvores.
- Criar e manter estruturas de dados dinâmicas exige alocação dinâmica de memória – a capacidade de um programa obter mais espaço de memória durante a execução para armazenar novos nodos e liberar espaço não mais necessário.
- O limite para alocação dinâmica de memória pode ser tão grande quanto a memória física disponível no computador ou a quantidade de espaço em disco disponível em um sistema de memória virtual. Frequentemente, os limites são muito menores, porque a memória disponível do computador deve ser compartilhada entre muitos usuários.
- O operador `new` recebe como operando o tipo do objeto que está sendo dinamicamente alocado e devolve uma referência a um objeto desse tipo recém-criado. Se não houver memória disponível, `new` dispara um `OutOfMemoryError`.
- A lista encadeada é uma coleção linear (isto é, uma seqüência) de objetos de classe auto-referencial, denominados nodos, conectados por *links* de referência.
- A lista encadeada é acessada através de uma referência ao primeiro nodo da lista. Cada nodo subsequente é acessado através do membro de referência de *link* armazenado no nodo anterior.
- Por convenção, a referência de *link* no último nodo de uma lista é configurada como `null` para marcar o final da lista.
- O nodo pode conter dados de qualquer tipo, incluindo objetos de outras classes.
- As árvores são estruturas de dados não-lineares.
- A lista encadeada é apropriada quando o número de elementos de dados a representar na estrutura de dados é imprevisível. As listas encadeadas são dinâmicas, portanto o comprimento de uma lista pode aumentar ou diminuir conforme necessário.
- O tamanho de um *array* Java “convencional” não pode ser alterado – o tamanho do *array* é fixado no momento da criação.
- As listas encadeadas podem ser mantidas em ordem de classificação simplesmente inserindo-se cada novo elemento no ponto adequado na lista.
- Os nodos de lista, em geral, não são armazenados contiguamente na memória. Em vez disso, eles são logicamente contíguos.
- Os métodos que manipulam o conteúdo de uma lista devem ser declarados `synchronized` para que os objetos `List` possam ser seguros para múltiplas *threads* quando utilizados em um programa com múltiplas *threads*. Se uma *thread* está modificando o conteúdo de uma lista, nenhuma outra *thread* tem permissão para modificar a mesma lista ao mesmo tempo.
- A pilha é uma versão limitada de uma lista encadeada – novos nodos podem ser adicionados e removidos apenas da parte superior de uma pilha. A pilha é conhecida como uma estrutura de dados “último a entrar, primeiro a sair” (*last-in, first-out* – LIFO).
- Os métodos primários utilizados para manipular uma pilha são `push` e `pop`. O método `push` adiciona um novo nodo ao topo da pilha. O método `pop` remove um nodo do topo da pilha e devolve o objeto `data` do nodo retirado.
- As pilhas têm muitas aplicações interessantes. Quando é feita uma chamada de método, o método chamado deve saber retornar para seu chamador, assim o endereço de retorno é adicionado à pilha de execução do programa. Se ocorre uma série de chamadas de métodos, os sucessivos valores de retorno são adicionados à pilha, na ordem último a entrar, primeiro a sair, de modo que cada método possa retornar para seu chamador.
- A pilha de execução do programa contém o espaço criado para variáveis locais a cada invocação de um método. Quando o método retorna para seu chamador, o espaço para as variáveis locais desse método é removido da pilha e essas variáveis não são mais conhecidas para o programa.
- As pilhas também são utilizadas por compiladores no processo de avaliação de expressões aritméticas e de geração de código de linguagem de máquina para processar as expressões.
- A técnica de implementar cada método de pilha como uma chamada para um método `List` é chamada de delegação – o método de pilha invocado delega a chamada para o método de `List` adequado.
- A fila é uma versão limitada de uma lista.
- A fila é semelhante a uma fila de caixa em um supermercado – a primeira pessoa na fila é atendida primeiro e os outros clientes entram na fila apenas no fim e esperam ser atendidos.
- Os nodos da fila são removidos apenas do início da fila e são inseridos somente no final da fila. Por essa razão, a fila é conhecida como uma estrutura de dados primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO).
- As operações de inserção e de remoção para uma fila são conhecidas como `enqueue` (enfileirar) e `dequeue` (desenfileirar).
- As filas têm muitas aplicações nos sistemas de computador. A maioria dos computadores tem apenas um único processador, portanto apenas um usuário por vez pode ser atendido. Os pedidos para os outros usuários são colocados em uma fila. O pedido no início da fila é o próximo a ser atendido. Cada entrada avança gradualmente para o início da fila à medida que os usuários são atendidos.

- As filas também são utilizadas para suportar *spooling* de impressão. Um ambiente multiusuário pode ter só uma impressora. Muitos usuários podem estar gerando saídas para impressão. Se a impressora estiver ocupada, outras saídas ainda podem ser geradas. Estas são colocadas no “*spool*” em disco (de maneira muito semelhante à maneira como um fio é enrolado em um carretel) onde esperam na fila até a impressora ficar disponível.
- Os pacotes de informações também esperam em filas em redes de computadores. Toda vez que um pacote chega em um nodo da rede, ele deve ser roteado para o próximo nodo na rede ao longo do caminho até o destino final do pacote. O nodo de roteamento roteia um pacote por vez, portanto os pacotes adicionais são enfileirados até o roteador conseguir roteá-los.
- Um servidor de arquivos em uma rede de computadores trata as solicitações de acesso a arquivos de muitos clientes por toda a rede. Os servidores têm uma capacidade limitada de atender às solicitações de clientes. Quando essa capacidade é excedida, as solicitações dos clientes esperam em filas.
- A árvore é uma estrutura bidimensional e não-linear de dados.
- Os nodos de árvores contêm dois ou mais *links*.
- A árvore binária é uma árvore cujos nodos contêm dois *links*. O nodo-raiz é o primeiro nodo em uma árvore.
- Cada *link* no nodo-raiz faz referência a um filho. O filho esquerdo é o primeiro nodo na subárvore esquerda e o filho direito é o primeiro nodo na subárvore direita.
- Os filhos de um nodo são chamados de irmãos. O nodo sem filhos é chamado de nodo-folha.
- Os cientistas de computação normalmente desenham as árvores indo do nodo-raiz para baixo.
- A árvore de pesquisa binária (sem valores de nodo duplicados) apresenta a característica de que os valores em qualquer subárvore esquerda são menores que o valor em seu nodo-pai e os valores em qualquer subárvore direita são maiores que o valor em seu nodo-pai.
- O nodo só pode ser inserido como um nodo-folha em uma árvore de pesquisa binária.
- O percurso na ordem de uma árvore de pesquisa binária processa os valores dos nodos em ordem crescente.
- O processo de criar uma árvore de pesquisa binária, na verdade, ordena os dados – e portanto esse processo é chamado de classificação de árvore binária.
- Em um percurso na pré-ordem, o valor em cada nodo é processado quando o nodo é visitado. Depois que o valor em um nodo dado é processado, os valores na subárvore esquerda são processados, e então os valores na subárvore direita são processados.
- Em um percurso na pós-ordem, o valor em cada nodo é processado depois dos valores de seus filhos.
- A árvore de pesquisa binária facilita a eliminação de duplicatas. Quando a árvore é criada, as tentativas de inserir um valor duplicado são reconhecidas porque uma duplicata segue as mesmas decisões “siga para a esquerda” ou “siga para a direita” em cada comparação que o valor original seguiu. Portanto, a duplicata acaba sendo comparada com um nodo que contém o mesmo valor. O valor duplicado simplesmente pode ser descartado nesse ponto.
- Pesquisar em uma árvore binária um valor que corresponde a um valor-chave também é rápido, especialmente em árvores *compactadas*. Em uma árvore compactada, cada nível contém aproximadamente duas vezes o número de elementos que o nível anterior. Assim, uma árvore de pesquisa binária com n elementos tem um mínimo de $\log_2 n$ níveis e, portanto, no máximo $\log_2 n$ comparações teriam de ser feitas para localizar uma correspondência ou para determinar que nenhuma correspondência existe. Pesquisar em uma árvore binária (compactada) de 1.000 elementos exige no máximo 10 comparações porque $2^{10} > 1.000$. Pesquisar uma árvore de pesquisa binária (compactada) de 1.000.000 elementos exige no máximo 20 comparações porque $2^{20} > 1.000.000$.

Terminologia

<i>algoritmos de travessia de árvore recursiva</i>	<i>excluindo um nodo</i>
<i>árvore</i>	<i>FIFO (primeiro a entrar, primeiro a sair)</i>
<i>árvore binária</i>	<i>fila</i>
<i>árvore de pesquisa binária</i>	<i>filho direito</i>
<i>cabeça de uma fila</i>	<i>filho esquerdo</i>
<i>classe auto-referencial</i>	<i>filhos</i>
<i>classificação de árvore binária</i>	<i>final de uma fila</i>
<i>delegando</i>	<i>inserindo um nodo</i>
<i>desenfileirar</i>	<i>LIFO (último a entrar, primeiro a sair)</i>
<i>eliminação de duplicata</i>	<i>lista encadeada</i>
<i>enfileirar</i>	<i>método predicado</i>
<i>estrutura linear de dados</i>	<i>nodo</i>
<i>estrutura não-linear de dados</i>	<i>nodo-filho</i>
<i>estruturas dinâmicas de dados</i>	<i>nodo-folha</i>

<i>nodo-pai</i>	<i>subárvore</i>
<i>nodo-raiz</i>	<i>subárvore direita</i>
<i>OutOfMemoryError</i>	<i>subárvore esquerda</i>
<i>parte superior de uma pilha (topo)</i>	<i>travessia</i>
<i>pilha</i>	<i>travessia na ordem de nível de uma árvore binária</i>
<i>pilha de execução do programa</i>	<i>travessia na ordem de uma árvore binária</i>
<i>pop</i>	<i>travessia na pós-ordem de uma árvore binária</i>
<i>push</i>	<i>travessia na pré-ordem de uma árvore binária</i>
<i>referência null</i>	<i>visita a um nodo</i>

Exercícios de auto revisão

19.1 Preencha as lacunas em cada uma das frases seguintes:

- a) A classe auto-_____ é utilizada para formar estruturas dinâmicas de dados que podem crescer e encolher durante a execução.
- b) O operador _____ aloca memória dinamicamente; esse operador devolve uma referência para a memória alocada.
- c) A _____ é uma versão limitada de uma lista encadeada em que os nodos podem ser inseridos e ser excluídos apenas do início da lista; essa estrutura de dados devolve valores dos nodos na ordem primeiro a entrar, primeiro a sair.
- d) O método que não altera uma lista encadeada mas simplesmente a examina para determinar se ela está vazia é conhecido como método _____.
- e) A fila é conhecida como uma estrutura de dados _____ porque os primeiros nodos inseridos são os primeiros nodos removidos.
- f) A referência ao próximo nodo em uma lista encadeada é conhecida como uma _____.
- g) Recuperar automaticamente memória alocada dinamicamente em Java se chama _____.
- h) A _____ é uma versão limitada de uma lista encadeada em que os nodos podem ser inseridos apenas no final da lista e excluídos apenas do início da lista.
- i) A _____ é uma estrutura de dados bidimensional não-linear que contém nodos com dois ou mais *links*.
- j) A pilha é uma estrutura de dados _____, porque o último nodo inserido é o primeiro nodo removido.
- k) Os nodos de uma árvore contém dois membros de *link*.
- l) O primeiro nodo de uma árvore é o nodo-_____.
- m) Cada *link* em um nodo de árvore faz referência a um _____ ou _____ desse nodo.
- n) O nodo de árvore que não tem filhos é chamado de nodo _____.
- o) Os quatro algoritmos de percorrer que mencionamos no texto para árvores de pesquisa binária são _____, _____, _____ e _____.

19.2 Quais são as diferenças entre uma lista encadeada e uma pilha?

19.3 Quais são as diferenças entre uma pilha e uma fila?

19.4 Talvez um título mais apropriado para este capítulo fosse “Estruturas reutilizáveis de dados”. Comente como cada uma das seguintes entidades ou conceitos contribuem para a capacidade de reutilização das estruturas de dados:

- a) classes
- b) herança
- c) composição

Forneça manualmente os percursos na ordem, pré-ordem e pós-ordem da árvore de pesquisa binária da Fig. 19.20.

Respostas dos exercícios de auto revisão

19.1 a) referencial. b) *new*. c) pilha. d) predicado. e) primeiro a entrar, primeiro a sair (FIFO) f) *link*. g) coleta de lixo. h) fila i) árvore. j) último a entrar, primeiro a sair (LIFO). k) binária. l) raiz. m) filho ou subárvore. n) folha. o) ordem, pré-ordem, pós-ordem, ordem de nível.

19.2 É possível inserir um nodo em qualquer lugar em uma lista encadeada e remover um nodo de qualquer lugar dela. Os nodos em uma pilha só podem ser inseridos no topo da pilha e só podem ser removidos do topo da pilha.

19.3 A fila tem referências a seu início e seu final de tal modo que os nodos podem ser inseridos no final e excluídos do início. Ela tem uma única referência ao topo da pilha, em que tanto a inserção como a exclusão de nodos são realizadas.

19.4 a) As classes permitem instanciar quantos objetos de estrutura de dados de um certo tipo (isto é, classe) quisermos.

b) A herança permite reutilizar código de uma superclasse em uma subclasse, de modo que a estrutura de dados da classe derivada também é uma estrutura de dados da classe básica.

c) A composição permite reutilizar código tornando uma estrutura de dados de um objeto de classe um membro de uma classe composta; se tornarmos o objeto de classe um membro **private** da classe composta, os métodos públicos do objeto de classe não ficam disponíveis por meio da interface do objeto composto.

19.5 A travessia na ordem é

11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

A travessia na pré-ordem é

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

A travessia na pós-ordem é

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

Exercícios

19.6 Escreva um programa que concatena dois objetos de lista encadeada de caracteres. A classe **ListaConcatenada** deve incluir o método **concatenar** que aceita referências para os dois objetos de lista como argumentos e concatena a segunda lista com a primeira lista.

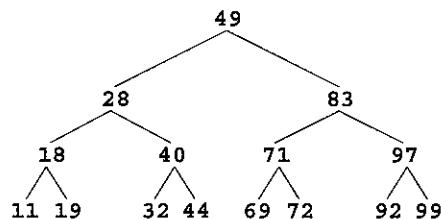


Fig. 19.20 Uma árvore de pesquisa binária de 15 nós.

19.7 Escreva um programa que mescla dois objetos lista ordenada de inteiros em um único objeto lista ordenada de inteiros. O método **merge** da classe **ListMerge** deve receber referências para cada um dos objetos de lista a ser mesclado e deve devolver uma referência ao objeto lista mesclada.

19.8 Escreva um programa que insere 25 inteiros aleatórios de 0 a 100 em ordem em um objeto lista encadeada. O programa deve calcular a soma dos elementos e a média em ponto flutuante dos elementos.

19.9 Escreva um programa que cria um objeto de lista encadeada de 10 caracteres e depois cria um segundo objeto de lista que contém uma cópia da primeira lista, mas na ordem inversa.

19.10 Escreva um programa que lê uma linha de texto e utiliza um objeto pilha para imprimir as palavras da linha na ordem inversa.

19.11 Escreva um programa que utiliza uma pilha para determinar se um *string* é um palíndromo (isto é, o *string* é sólido identicamente do início para o fim e do fim para o início). O programa deve ignorar espaços e pontuação.

19.12 As pilhas são utilizadas pelos compiladores para ajudar no processo de avaliação de expressões e geração de código de linguagem de máquina. Nesse e no próximo exercício, investigamos como os compiladores avaliam expressões aritméticas que consistem apenas em constantes, operadores e parênteses.

As pessoas geralmente escrevem expressões como $3 + 4 \cdot 7 / 9$ em que o operador (+ ou /, neste caso) é escrito entre seus operandos – isso se chama *notação infix*. Os computadores “preferem” a *notação pós-fixa* em que o operador é escrito à direita de seus dois operandos. As expressões infixas precedentes apareceriam na notação pós-fixa como $3\ 4 + \cdot 7\ 9 /$, respectivamente.

Para avaliar uma expressão infix complexa, o compilador primeiro converteria a expressão em notação pós-fixa e avaliaria a versão pós-fixa da expressão. Cada um desses algoritmos exige apenas uma única passagem da esquerda para a direita pela expressão. Cada algoritmo utiliza um objeto pilha como suporte à sua operação e em cada algoritmo a pilha é utilizada para um propósito diferente.

Nesse exercício, você escreverá uma versão em Java do algoritmo de conversão de infix para pós-fixa. No próximo exercício, você escreverá uma versão Java do algoritmo de avaliação de expressão pós-fixa. Em um exercício posterior, você descobrirá que o código que você escrever nesse exercício pode ajudá-lo a implementar um compilador completo que funciona.

Escreva a classe `InfixToPostfixConverter` para converter uma expressão aritmética infix comum (suponha que uma expressão válida foi digitada) com inteiros de um único dígito como

`(6 + 2) * 5 - 8 / 4`

para uma expressão pós-fixa. A versão pós-fixa da expressão infix precedente é (note que nenhum parêntese foi necessário)

`6 2 + 5 * 8 4 / -`

O programa deve ler a expressão para o `StringBuffer infix` e utilizar uma das classes de pilha implementadas neste capítulo para ajudar a criar a expressão pós-fixa no `StringBuffer postfix`. O algoritmo para criar uma expressão pós-fixa é o seguinte:

- Inserir um parêntese esquerdo '(' na pilha.
- Acrescentar um parêntese direito ')' ao final de `infix`.
- Enquanto a pilha não estiver vazia, ler `infix` da esquerda para a direita e fazer o seguinte:

Se o caractere atual em `infix` for um dígito, acrescentá-lo a `postfix`.

Se o caractere atual em `infix` for um parêntese esquerdo, adicioná-lo à pilha.

Se o caractere atual em `infix` for um operador:

Retirar os operadores (se houver algum) do topo da pilha enquanto eles tiverem precedência igual ou mais alta que a do operador atual e acrescentar os operadores removidos a `postfix`.

Inserir na pilha o caractere atual de `infix`.

Se o caractere atual em `infix` for um parêntese direito:

Retirar os operadores do topo da pilha e acrescentá-los a `postfix` até que um parêntese esquerdo esteja na parte superior da pilha.

Retirar (e descartar) o parêntese esquerdo da pilha.

As seguintes operações aritméticas são permitidas em uma expressão:

- + adição
- subtração
- * multiplicação
- / divisão
- ^ exponenciação
- % módulo

A pilha deve ser mantida com nodos de pilha que contenham uma variável de instância e uma referência ao próximo nodo da pilha. Eis alguns métodos que você pode querer fornecer:

- O método `convertToPostfix`, que converte a expressão infix para a notação pós-fixa.
- O método `isOperator`, que determina se `c` é um operador.
- O método `precedence`, que determina se a precedência do `operator1` (da expressão infix) é menor que, igual ou maior que a precedência do `operator2` (da pilha). O método devolve `true` se `operator1` tiver precedência mais baixa que `operator2`. Caso contrário, `false` é devolvido.
- O método `stackTop` (que deve ser adicionado à classe de pilha), que devolve o valor do topo da pilha sem removê-lo da pilha.

19.13 Escreva a classe `PostfixEvaluator`, que avalia uma expressão pós-fixa (suponha que é válida), como

6 2 + 5 * 8 2 / -

O programa deve ler uma expressão pós-fixa que consiste em dígitos e operadores em um `StringBuffer`. Utilizando versões modificadas dos métodos de pilha implementados anteriormente neste capítulo, o programa deve percorrer a expressão e avaliá-la. O algoritmo é o seguinte:

- Acrescentar um parêntese direito (' ') ao final da expressão pós-fixa. Quando o caractere parêntese direito for encontrado, mais nenhum processamento é necessário.
- Enquanto o caractere parêntese direito não for encontrado, ler a expressão da esquerda para a direita.

Se o caractere atual for um dígito, faça o seguinte:

Inserir seu valor de inteiro na pilha (o valor inteiro de um caractere de dígito é seu valor no conjunto de caracteres do computador menos o valor de '0' em Unicode).

Caso contrário, se o caractere atual for um *operador*:

Remover os dois elementos do topo da pilha para as variáveis **x** e **y**.

Calcular o valor de **y operator x**.

Inserir o resultado do cálculo na pilha.

- Quando o parêntese direito for encontrado na expressão, remover o valor do topo da pilha. Esse é o resultado da expressão pós-fixa.

[Nota: em (b) (com base no exemplo de expressão no início deste exercício), se o operador for '/ ', o topo da pilha é 2 e o próximo elemento na pilha é 8; assim, remova 2 para **x**, remova 8 para **y**, avalie **8 / 2** e insira o resultado, 4, de volta na pilha. Essa nota também se aplica ao operador '- ']. As operações aritméticas permitidas em uma expressão são:

- + adição
- subtração
- * multiplicação
- / divisão
- ^ exponenciação
- % módulo

A pilha deve ser mantida com uma das classes de pilha apresentadas neste capítulo. Você pode querer fornecer os seguintes métodos:

- O método `evaluatePostfixExpression`, que avalia a expressão pós-fixa.
- O método `calculate`, que avalia a expressão `op1 operator op2`.
- O método `push`, que insere um valor na pilha.
- O método `pop`, que remove um valor da pilha.
- O método `isEmpty`, que determina se a pilha está vazia.
- O método `printStack`, que imprime a pilha.

19.14 Modifique o programa avaliador de expressão pós-fixa do Exercício 19.13 de modo que ele possa processar operandos inteiros maiores que 9.

19.15 (*Simulação de supermercado*) Escreva um programa que simula uma fila de caixa em um supermercado. A fila é um objeto fila. Os clientes (isto é, os objetos cliente) chegam em intervalos aleatórios inteiros de 1 a 4 minutos. Além disso, cada cliente é atendido em intervalos aleatórios inteiros de 1 a 4 minutos. Obviamente, as taxas precisam ser equilibradas. Se a taxa média de chegada for maior que a taxa média de atendimento, a fila crescerá infinitamente. Mesmo com taxas "equilibradas", a aleatoriedade ainda pode provocar filas longas. Execute a simulação de supermercado para um dia de 12 horas (720 minutos), utilizando o seguinte algoritmo:

- Escolher um inteiro aleatório entre 1 e 4 para determinar o minuto em que o primeiro cliente chega.
- Na hora de chegada do primeiro cliente:
 - Determinar o tempo de atendimento do cliente (inteiro aleatório de 1 a 4).
 - Começar a atender o cliente.
 - Agendar a hora de chegada do próximo cliente (inteiro aleatório 1 a 4 adicionado à hora atual).
- Para cada minuto do dia considere o seguinte:
 - Se o próximo cliente chegar, fazer o seguinte:
 - Informar isso.
 - Colocar o cliente na fila.
 - Agendar a hora de chegada do próximo cliente.
 - Se o atendimento foi completado para o último cliente, fazer o seguinte:
 - Informar isso.

- Tirar da fila o próximo cliente a ser atendido.
 Determinar o tempo de conclusão do atendimento ao cliente (inteiro aleatório de 1 a 4 adicionado à hora atual).

Agora execute sua simulação para 720 minutos e responda a cada uma das perguntas seguintes:

- Qual é o número máximo de clientes na fila a qualquer momento?
- Qual é a espera mais longa que qualquer cliente experimenta?
- O que acontece se o intervalo de chegada é alterado de 1 a 4 minutos para 1 a 3 minutos?

- 19.16** Modifique o programa das Figs. 19.17 e 19.18 para permitir que a árvore binária contenha duplicatas.
- 19.17** Escreva um programa baseado no programa das Figs. 19.17 e 19.18 que leia uma linha de texto, “quebre” a frase em palavras separadas (você pode querer utilizar a classe `StreamTokenizer` do pacote `java.io`), insira as palavras em uma árvore de pesquisa binária e imprima percursos da árvore na ordem, na pré-ordem e na pós-ordem.
- 19.18** Neste capítulo, vimos que a eliminação de duplicatas é simples e direta quando se cria uma árvore de pesquisa binária. Descreva como você realizaria a eliminação de duplicatas ao utilizar apenas um *array* unidimensional. Compare o desempenho da eliminação de duplicatas baseada em *array* com o desempenho da eliminação de duplicatas baseada em árvore de pesquisa binária.
- 19.19** Escreva um método `depth` que recebe uma árvore binária e determina quantos níveis ela tem.
- 19.20** (*Imprimir recursivamente uma lista de trás para frente*) Escreva um método `printListBackwards` que dá saída recursivamente aos itens em um objeto de lista encadeada na ordem inversa. Escreva um programa de teste que cria uma lista ordenada de inteiros e imprime a lista na ordem inversa.

- 19.21** (*Pesquisar recursivamente uma lista*) Escreva um método `searchList` que procura recursivamente em um objeto lista encadeada um valor especificado. O método `searchList` deve devolver uma referência para o valor se ele for localizado; caso contrário, deve devolver nulo. Utilize seu método em um programa de teste que cria uma lista de inteiros. O programa deve solicitar ao usuário um valor para localizar na lista.

- 19.22** (*Exclusão de árvore binária*) Neste exercício, discutimos a exclusão de itens de árvores de pesquisa binária. O algoritmo de exclusão não é tão simples e direto quanto o algoritmo de inserção. Há três casos que são encontrados ao se excluir um item – o item está contido em um nodo-folha (isto é, não tem filhos), o item está contido em um nodo que tem um filho ou o item está contido em um nodo que tem dois filhos.

Se o item a ser excluído estiver contido em um nodo-folha, o nodo é excluído e a referência no nodo-pai é configurada como nulo.

Se o item a ser excluído estiver contido em um nodo com um filho, a referência no nodo-pai é configurada para fazer referência ao nodo-filho e o nodo que contém o item de dados é excluído. Isso faz com que o nodo-filho tome o lugar do nodo excluído na árvore.

O último caso é o mais difícil. Quando um nodo com dois filhos é excluído, outro nodo na árvore deve tomar seu lugar. Entretanto, a referência no nodo pai não pode simplesmente ser alterada para fazer referência a um dos filhos do nodo a ser excluído. Na maioria dos casos, a árvore de pesquisa binária resultante não obedeceria à seguinte característica das árvores de pesquisa binária (sem valores duplicados): *os valores em qualquer subárvore esquerda são menores que o valor no nodo-pai e os valores em qualquer subárvore direita são maiores que o valor no nodo-pai*.

Qual é o nodo utilizado como *nodo substituto* para manter essa característica? Será o nodo que contém o maior valor na árvore menor que o valor no nodo que está sendo excluído, ou o nodo que contém o menor valor na árvore maior que o valor no nodo que está sendo excluído. Vamos considerar o nodo com o menor valor. Em uma árvore de pesquisa binária, o maior valor menor que um valor do pai encontra-se na subárvore esquerda do nodo-pai e seguramente estará contido no nodo mais à direita da subárvore. Esse nodo é encontrado percorrendo-se para baixo a subárvore esquerda pela direita até que a referência ao filho direito do nodo atual seja nula. Agora estamos fazendo referência ao nodo substituto que é ou um nodo-folha ou um nodo com um filho à sua esquerda. Se o nodo substituto for um nodo-folha, os passos para realizar a exclusão são os seguintes:

- Armazenar a referência ao nodo a ser excluído em uma variável de referência temporária.
- Configurar a referência no pai do nodo que está sendo excluído para fazer referência ao nodo substituto.
- Configurar a referência no pai do nodo substituto como nulo.
- Configurar a referência para a subárvore direita no nodo substituto para fazer referência à subárvore direita do nodo a ser excluído.
- Configurar a referência para a subárvore esquerda no nodo substituto para fazer referência à subárvore esquerda do nodo a ser excluído.

Os passos de exclusão para um nodo substituto com um filho à esquerda são semelhantes àqueles para um nodo substituto sem filhos, mas o algoritmo também deve mover o filho para a posição do nodo substituto na árvore. Se o nodo substituto for um nodo com um filho à esquerda, os passos para realizar a exclusão são os seguintes:

- Armazenar a referência ao nodo a ser excluído em uma variável de referência temporária.
- Configurar a referência no pai do nodo que está sendo excluído para fazer referência ao nodo substituto.
- Configurar a referência no pai do nodo substituto para fazer referência ao filho esquerdo do nodo substituto.
- Configurar a referência à subárvore direita do nodo substituto para fazer referência à subárvore direita do nodo a ser excluído.
- Configurar a referência à subárvore esquerda no nodo substituto para fazer referência à subárvore esquerda do nodo a ser excluído.

Escreva o método `deleteNode`, que recebe como argumento o valor a ser excluído. O método `deleteNode` deve localizar na árvore o nodo contendo o valor a ser excluído e utilizar os algoritmos discutidos aqui para excluir o nodo. Se o valor não for localizado na árvore, o método deve imprimir uma mensagem que indica se o valor foi excluído ou não. Modifique o programa das Figs. 19.17 e 19.18 para utilizar esse método. Depois de excluir um item, chame os métodos `inorderTraversal`, `preorderTraversal` e `postorderTraversal` para confirmar que a operação de exclusão foi realizada corretamente.

19.23 (Pesquisa de árvore binária) Escreva o método `binaryTreeSearch`, que tenta localizar um valor especificado em um objeto árvore de pesquisa binária. O método deve aceitar como argumento uma chave de pesquisa a ser localizada. Se o nodo que contém a chave de pesquisa for localizado, o método deve devolver uma referência a esse nodo; caso contrário, o método deve devolver uma referência nula.

19.24 (Travessia de árvore binária na ordem de nível) O programa das Figs. 19.17 e 19.18 ilustrou os três métodos recursivos de percorrer uma árvore binária – os percursos na ordem, pré-ordem e pós-ordem. Esse exercício apresenta a *travessia na ordem de nível* de uma árvore binária, em que os valores dos nodos são impressos nível por nível, iniciando no nível do nodo-raiz. Os nodos em cada nível são impressos da esquerda para a direita. O percurso na ordem de nível não é um algoritmo recursivo. Ela utiliza um objeto fila para controlar a saída dos nodos. O algoritmo é o seguinte:

- Inserir o nodo-raiz na fila.
- Enquanto houver nodos na fila, fazer o seguinte:
 - Obter o próximo nodo na fila.
 - Imprimir o valor do nodo.
 - Se a referência ao filho esquerdo do nodo não for nula:
 - Inserir o nodo-filho esquerdo na fila.
 - Se a referência ao filho direito do nodo não é nula:
 - Inserir o nodo-filho direito na fila.

Escreva o método `levelOrder` para realizar um percurso na ordem de nível de um objeto de árvore binária. Modifique o programa das Figs. 19.17 e 19.18 para utilizar esse método. [Nota: você também precisará utilizar métodos de processamento de fila da Fig. 19.13 nesse programa.]

19.25 (Imprimindo árvores) Escreva um método recursivo `outputTree` para exibir um objeto árvore binária na tela. O método deve dar saída à árvore linha por linha com o topo da árvore na parte esquerda da tela e a parte inferior em direção à parte direita da tela. Cada linha é enviada para a saída verticalmente. Por exemplo, a árvore binária ilustrada na Fig. 19.20 é enviada para a saída como mostrado na Fig. 19.21.

Observe que o nodo mais à direita da folha aparece no topo da saída na coluna mais à direita e o nodo-raiz aparece à esquerda da saída. Cada coluna de saída inicia cinco espaços à direita da coluna precedente. O método `outputTree` deve receber um argumento `totalSpaces` para representar o número de espaços que precedem o valor a ser enviado para a saída (essa variável deve iniciar como zero de modo que o nodo-raiz seja enviado para a saída na parte esquerda da tela). O método utiliza um percurso na ordem modificado para dar a saída à árvore – ele inicia no nodo mais à direita na árvore e segue para a esquerda. O algoritmo é o seguinte:

Enquanto a referência ao nodo atual não for nula:

Chamar recursivamente `outputTree` com a subárvore direita do nodo atual e `totalSpaces + 5`.

Utilizar uma estrutura `for` para contar de 1 a `totalSpaces` e enviar os espaços para saída. Envie para a saída o valor no nodo atual.

Configurar a referência ao nodo atual para fazer referência à subárvore esquerda do nodo atual.

Incrementar `totalSpaces` por 5.

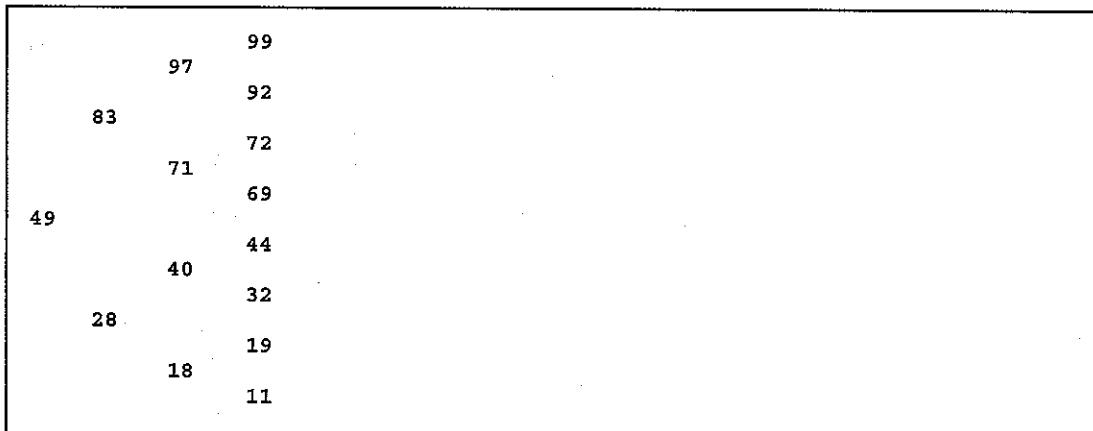


Fig. 19.21 Exemplo de saída do método recursivo `outputTree`.

Seção especial: construindo seu próprio compilador

Nos Exercícios 7.42 e 7.43, apresentamos a Simpletron Machine Language (SML) e implementamos um simulador de computador Simpletron para executar programas escritos em SML. Nesta seção, construímos um compilador que converte programas escritos em uma linguagem de programação de alto nível para SML. Esta seção “amarra” o processo de programação inteiro. Você escreverá os programas nessa nova linguagem de alto nível, compilará esses programas no compilador que construir e executará os programas no simulador que construiu no Exercício 7.43. Você deve se esforçar o máximo possível para implementar seu compilador de uma maneira orientada a objetos.

19.26 (*A linguagem Simple*) Antes de iniciarmos a construção do compilador, discutimos uma linguagem simples, mas ainda poderosa e de alto nível, semelhante às versões iniciais da conhecida linguagem Basic. Chamamos essa linguagem de *Simple*. Cada *instrução Simple* consiste em um *número de linha* e uma *instrução Simple*. Os números de linha devem aparecer em ordem crescente. Cada instrução inicia com um dos seguintes *comandos Simple*: `rem`, `input`, `let`, `print`, `goto`, `if/goto` ou `end` (veja a Fig. 19.22). Todos os comandos, exceto `end`, podem ser utilizados repetidamente. O Simple avalia apenas expressões inteiras que utilizam os operadores `+, -, *, /`. Esses operadores têm a mesma precedência que em Java. Podem usar parênteses para alterar a ordem de avaliação de uma expressão.

Comando	Instrução de exemplo	Descrição
<code>rem</code>	<code>50 rem isto é comentário</code>	Qualquer texto depois do comando <code>rem</code> é apenas para fins de documentação e é ignorado pelo compilador.
<code>input</code>	<code>30 input x</code>	Exibe um ponto de interrogação que pede ao usuário para digitar um inteiro. Lê esse inteiro do teclado e armazena o inteiro em <code>x</code> .
<code>let</code>	<code>80 let u = 4 * (j - 56)</code>	Atribui <code>u</code> o valor de <code>4 * (j - 56)</code> . Observe que uma expressão arbitrariamente complexa pode aparecer à direita do sinal de igual.
<code>print</code>	<code>10 print w</code>	Exibe o valor de <code>w</code> .
<code>goto</code>	<code>70 goto 45</code>	Transfere o controle do programa para a linha <code>45</code> .
<code>if/goto</code>	<code>35 if i == z goto 80</code>	Verifica se <code>i</code> e <code>z</code> são iguais e transfere o controle do programa para a linha <code>80</code> se a condição for verdadeira; caso contrário, continua a execução com a próxima instrução.
<code>end</code>	<code>99 end</code>	Termina a execução do programa.

Fig. 19.22 Comandos de Simple.

Nosso compilador Simple reconhece apenas letras minúsculas. Todos os caracteres em um arquivo Simple devem estar em letras minúsculas (letras maiúsculas resultam em um erro de sintaxe, a menos que apareçam em uma instrução `rem`, caso em que são ignoradas). O *nome de variável* tem uma única letra. Simple não permite nomes de variáveis descritivos, portanto as variáveis devem ser explicadas em comentários para indicar sua finalidade em um programa. Simple utiliza apenas variáveis inteiras. Não tem declarações de variáveis – a mera menção de um nome de variável em um programa faz com que a variável seja declarada e inicializada com zero. A sintaxe de Simple não permite manipulação de *strings* (ler um *string*, gravar um *string*, comparar *strings*, etc.) Se um *string* for encontrado em um programa Simple (após qualquer outro comando que não `rem`), o compilador gera um erro de sintaxe. A primeira versão de nosso compilador supõe que os programas Simple são digitados corretamente. O Exercício 19.29 pede para o leitor modificar o compilador para realizar verificação de erros de sintaxe.

Simple utiliza a instrução condicional `if/goto` e a instrução incondicional `goto` para alterar o fluxo de controle durante a execução do programa. Se a condição na instrução `if/goto` for verdadeira, o controle é transferido para uma linha específica do programa. Os seguintes operadores relacionais e de igualdade são válidos em uma instrução `if/goto`: `<`, `>`, `<=`, `=` ou `!=`. A precedência desses operadores é a mesma que em Java.

Vamos agora considerar diversos programas que demonstram recursos de Simple. O primeiro programa (Fig. 19.23) lê dois inteiros do teclado, armazena os valores nas variáveis `a` e `b` e calcula e imprime sua soma (armazenada na variável `c`).

O programa da Fig. 19.24 determina e imprime o maior de dois inteiros. Os inteiros são lidos do teclado e armazenados em `s` e `t`. A instrução `if/goto` testa a condição `s >= t`. Se a condição for verdadeira, o controle é transferido para a linha 90 e `s` é enviado para a saída; caso contrário, `t` é enviado para a saída e o controle é transferido para a instrução `end` na linha 99, em que o programa termina.

```

1 10 rem  determina e imprime a soma de dois inteiros
2 15 rem
3 20 rem  lê os dois inteiros
4 30 input a
5 40 input b
6 45 rem
7 50 rem  soma os inteiros e armazena o resultado em c
8 60 let c = a + b
9 65 rem
10 70 rem  imprime o resultado
11 80 print c
12 90 rem  termina a execução do programa
13 99 end

```

Fig. 19.23 Programa em Simple que determina a soma de dois inteiros.

Simple não fornece uma estrutura de repetição (como `for`, `while` ou `do/while` de Java). Entretanto, a linguagem pode simular cada uma das estruturas de repetição de Java utilizando as instruções `if/goto` e `goto`. A Fig. 19.25 utiliza um laço controlado por sentinelas para calcular os quadrados de vários inteiros. Cada inteiro é lido do teclado e armazenado na variável `j`. Se o valor lido for o valor de sentinelas `-9999`, o controle é transferido para a linha 99, na qual o programa termina. Caso contrário, o quadrado de `j` é atribuído a `k`, é enviado para a saída na tela e o controle é passado para a linha 20, em que o próximo inteiro é lido.

Utilizando os programas de exemplo das Figs. 19.23 a 19.25 como guia, escreva um programa Simple para realizar cada uma das seguintes tarefas:

- Ler três inteiros, determinar sua média e imprimir o resultado.
- Utilizar um laço controlado por sentinelas para ler 10 inteiros e calcular e imprimir sua soma.
- Utilizar um laço controlado por contador para ler 7 inteiros, alguns positivos e alguns negativos e calcular e imprimir a média.
- Ler uma série de inteiros e determinar e imprimir o maior. O primeiro inteiro lido indica quantos números devem ser processados (após ele).
- Ler 10 inteiros e imprimir o menor.
- Calcular e imprimir a soma dos inteiros pares de 2 a 30.
- Calcular e imprimir o produto dos inteiros ímpares de 1 a 9.

```

1 10 rem determina e imprime o maior de dois inteiros
2 20 input s
3 30 input t
4 32 rem
5 35 rem testa se s >= t
6 40 if s >= t goto 90
7 45 rem
8 50 rem t é maior que s, então imprime t
9 60 print t
10 70 goto 99
11 75 rem
12 80 rem s é maior que ou igual a t, então imprime s
13 90 print s
14 99 end

```

Fig. 19.24 Programa em Simple que determina o maior de dois inteiros.

```

1 10 rem calcula os quadrados de vários inteiros
2 20 input j
3 23 rem
4 25 rem testa o valor da sentinel
5 30 if j == -9999 goto 99
6 33 rem
7 35 rem calcula o quadrado de j e atribui o resultado a k
8 40 let k = j * j
9 50 print k
10 53 rem
11 55 rem repete o laço para obter o próximo j
12 60 goto 20
13 99 end

```

Fig. 19.25 Calcula os quadrados de vários inteiros.

19.27 (*Construindo um compilador; pré-requisito: completar os Exercícios 7.42, 7.43, 19.12, 19.13 e 19.26*) Agora que a linguagem Simple foi apresentada (Exercício 19.26), discutimos como construir um compilador de Simple. Primeiro, analisaremos o processo pelo qual um programa Simple é convertido para SML e executado pelo simulador Simpletron (veja a Fig. 19.26). O arquivo que contém um programa Simple é lido pelo compilador e convertido para código SML. O código SML é enviado para um arquivo de saída em disco, no qual as instruções de SML aparecem uma em cada linha. O arquivo SML é então carregado no simulador Simpletron e os resultados são enviados para um arquivo em disco e para a tela. Observe que o programa Simpletron desenvolvido no Exercício 7.43 pegou sua entrada a partir do teclado. Ele deve ser modificado para ler de um arquivo em disco de modo que possa executar os programas produzidos pelo nosso compilador.

O compilador de Simple realiza duas *passagens* do programa Simple para convertê-lo em SML. A primeira passagem constrói uma *tabela de símbolos* (objeto) em que cada *número de linha* (objeto), *nome de variável* (objeto) e *constante* (objeto) do programa Simple é armazenado com seu tipo e posição correspondente no código SML final (a tabela de símbolos é discutida em detalhes a seguir). A primeira passagem também produz o(s) objeto(s) instrução (*instruction*) de SML correspondente(s) para cada uma das instruções (*statements*) de Simple (objeto, etc.). Se o programa Simple contiver instruções (*statements*) que transferem o controle para uma linha mais adiante no programa, a primeira passagem resulta em um programa SML que contém algumas instruções “inacabadas”. A segunda passagem do compilador localiza e completa as instruções inacabadas e envia o programa SML para um arquivo de saída.

Primeira passagem

O compilador começa lendo uma instrução (*statement*) do programa Simple para a memória. A linha deve ser separada em suas *tokens* individuais (isto é, “fragmentos” de uma instrução) para processamento e compilação (a classe **StreamTokenizer** do pacote **java.io** pode ser utilizada). Lembre-se de que cada instrução (*statement*) inicia com um número de linha seguido por um comando. Quando o compilador divide uma instrução (*statement*) em *tokens*, se a *token* for um número de linha, uma variável ou uma constante, ela é colocada na tabela de símbolos. Coloca-se um número de linha na

tabela de símbolos apenas se for a primeira *token* em uma instrução. O objeto `symbolTable` é um *array* de objetos `tableEntry` que representa cada símbolo no programa. Não há restrição quanto ao número de símbolos que podem aparecer no programa. Portanto, a `symbolTable` para um programa particular pode ser grande. Por enquanto, crie a `symbolTable` como um *array* de 100 elementos. Você pode aumentar ou diminuir seu tamanho depois que o programa estiver funcionando.

Cada objeto `tableEntry` contém três membros. O membro `symbol` é um inteiro que contém a representação em Unicode do nome de uma variável (lembre-se de que os nomes de variáveis são caracteres isolados), um número de linha ou uma constante. O membro `type` é um dos seguintes caracteres que indicam o tipo do símbolo: 'C' para constante, 'L' para número de linha ou 'V' para variável. O membro `location` contém a posição da memória do Simpletron (00 a 99) a que o símbolo faz referência. A memória do Simpletron é um *array* de 100 inteiros em que as instruções de SML e os dados são armazenados. Para um número de linha, a `location` indica o elemento no *array* de memória do Simpletron em que as instruções de SML para a instrução Simple iniciam. Para uma variável ou constante, a posição é o elemento no *array* de memória de Simpletron em que a variável ou constante é armazenada. Variáveis e constantes são alocadas a partir do fim da memória do Simpletron para trás. A primeira variável ou constante é armazenada na posição 99, a próxima na posição 98, etc.

A tabela de símbolos desempenha um papel importante na conversão de programas Simple em SML. Aprendemos no Capítulo 7 que uma instrução SML é um inteiro de quatro dígitos composto por duas partes – o *código de operação* e o *operando*. O código de operação é determinado por comandos em Simple. Por exemplo, o comando `input` de Simple corresponde ao código de operação 10 (ler) do SML e o comando `print` corresponde ao código de operação 11 (escrever) de SML. O operando é uma posição da memória que contém os dados com que o código de operação realiza sua tarefa (por exemplo, o código de operação 10 lê um valor do teclado e o armazena na posição da memória especificada pelo operando). O compilador pesquisa a `symbolTable` para determinar a posição da memória do Simpletron para cada símbolo, assim a posição correspondente pode ser utilizada para completar as instruções de SML.

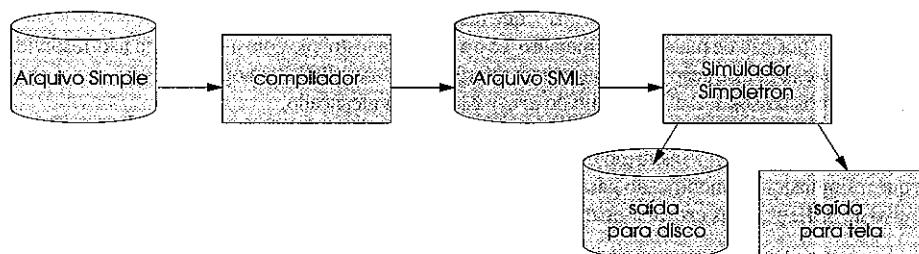


Fig. 19.26 Gravando, compilando e executando um programa da linguagem Simple.

A compilação de cada instrução de Simple baseia-se em seu comando. Por exemplo, depois que o número de linha de uma instrução `rem` é inserido na tabela de símbolos, o restante da instrução é ignorada pelo compilador porque um comentário (`remark`) serve apenas para fins de documentação. As instruções `input`, `print`, `goto` e `end` correspondem às instruções do SML `read`, `write`, `branch` (para uma posição específica) e `halt`. As instruções que contêm esses comandos do Simple são convertidas diretamente em SML (*nota*: uma instrução `goto` pode conter uma referência não-resolvida se o número de linha especificado se referir a uma instrução mais adiante no arquivo do programa Simple; esse processo se chama referência antecipada).

Quando uma instrução `goto` é compilada com uma referência não-resolvida, a instrução de SML deve ser *marcada* para indicar que a segunda passagem do compilador deve completar a instrução. As “marcas” são armazenadas em um *array* `flags` de 100 elementos do tipo `int` em que cada elemento é inicializado como -1. Se a posição da memória à qual um número de linha no programa Simple se refere ainda não é conhecida (isto é, não está na tabela de símbolos), o número de linha é armazenado no *array* `flags` no elemento com o mesmo subscrito que a instrução incompleta. O operando da instrução incompleta é configurado provisoriamente como 00. Por exemplo, uma instrução de *unconditional branch* (fazendo uma referência antecipada) é deixada como +4000 até a segunda passagem do compilador. A segunda passagem do compilador será descrita brevemente.

A compilação das instruções `if/goto` e `let` é mais complicada que a de outras instruções – elas são as únicas instruções que produzem mais de uma instrução SML. Para uma instrução `if/goto`, o compilador produz código para tes-

tar a condição e desviar para outra linha se necessário. O resultado do desvio pode ser uma referência não-resolvida. Cada um dos operadores relacionais e de igualdade pode ser simulado utilizando-se as instruções *branch zero* e *branch negative* do SML (ou possivelmente uma combinação das duas).

Para uma instrução `let`, o compilador produz código para avaliar uma expressão aritmética arbitrariamente complexa que consiste em de variáveis e/ou constantes inteiros. As expressões devem separar cada operando e operador com espaços. Os Exercícios 19.12 e 19.13 apresentaram o algoritmo de conversão de notação infixa para pós-fixa e o algoritmo de avaliação de pós-fixa utilizado por compiladores para avaliar expressões. Antes de prosseguir com seu compilador, você deve completar cada um desses exercícios. Quando um compilador encontra uma expressão, ele converte a expressão da notação infixa para a notação pós-fixa, e depois avalia a expressão pós-fixa.

Como é que o compilador produz a linguagem de máquina para avaliar uma expressão que contém variáveis? O algoritmo de avaliação pós-fixa contém um “gancho” no qual o compilador pode gerar instruções de SML em vez de realmente avaliar a expressão. Para ativar esse “gancho” no compilador, o algoritmo de avaliação pós-fixa deve ser modificado para pesquisar a tabela de símbolos para cada símbolo que ele encontra (e possivelmente inseri-lo), determinar a posição de memória correspondente ao símbolo e *inserir a posição de memória na pilha (em vez do símbolo)*. Quando um operador é encontrado na expressão pós-fixa, as duas posições de memória no topo da pilha são removidas e a linguagem de máquina para efetuar a operação é produzida com as posições de memória como operandos. O resultado de cada subexpressão é armazenado em uma posição temporária na memória e inserido de volta na pilha de modo que a avaliação da expressão pós-fixa possa continuar. Quando a avaliação pós-fixa está completa, a posição de memória que contém o resultado é a única posição deixada na pilha. Ela é removida e são geradas as instruções de SML para atribuir o resultado à variável à esquerda da instrução `let`.

Segunda passagem

A segunda passagem do compilador realiza duas tarefas: resolver quaisquer referências não-resolvidas e enviar o código de SML para um arquivo de saída. A solução de referências ocorre da seguinte forma:

- Procurar no array `flags` uma referência não-resolvida (isto é, um elemento com um valor diferente de `-1`).
- Localizar o objeto no array `symbolTable` que contém o símbolo armazenado no array `flags` (certifique-se de que o tipo do símbolo é '`L`', que indica número de linha).
- Inserir a posição de memória do membro `location` na instrução com a referência não-resolvida (lembre-se de que uma instrução que contém uma referência não-resolvida tem operando `00`).
- Repetir os passos 1, 2 e 3 até o fim do array `flags` ser alcançado.

Depois que o processo de solução estiver completo, o array inteiro que contém o código de SML é enviado para um arquivo de saída em disco com uma instrução de SML por linha. Esse arquivo pode ser lido pelo Simpletron para execução (depois que o simulador for modificado para ler sua entrada de um arquivo). Compilar seu primeiro programa Simple para um arquivo de SML e depois executar esse arquivo deve lhe dar uma verdadeira sensação de realização pessoal.

Um exemplo completo

O seguinte exemplo ilustra a conversão completa de um programa Simple para SML da forma que ela será realizada pelo compilador Simple. Considere um programa Simple que lê um inteiro e calcula a soma dos valores de 1 até esse inteiro. O programa e as instruções de SML produzidas pela primeira passagem do compilador Simple são ilustrados na Fig. 19.27. A tabela de símbolos construída pela primeira passagem é mostrada na Fig. 19.28.

Programa Simple	Posição e instrução na SML	Descrição
5 rem soma 1 a x	nenhuma	rem ignorado
10 input x	00 +1099	lê x para a posição 99
15 rem testa se y == x	nenhuma	rem ignorado
20 if y == x goto 60	01 +2098	carrega y (98) no acumulador
	02 +3199	subtrai x (99) do acumulador
	03 +4200	desvia se zero para posição não-resolvida
25 rem incrementa y	nenhuma	rem ignorado

Fig. 19.27 Instruções de SML produzidas depois da primeira passagem do compilador (parte 1 de 2).

Programa Simple	Posição e instrução na SML	Descrição
30 let y = y + 1	04 +2098 05 +3097 06 +2196 07 +2096 08 +2198	carrega y no acumulador soma 1 (97) ao acumulador armazena na posição temporária 96 carrega da posição temporária 96 armazena o acumulador em y
35 rem soma y a total	nenhuma	rem ignorado
40 let t = t + y	09 +2095 10 +3098 11 +2194 12 +2094 13 +2195	carrega t (95) no acumulador adiciona y ao acumulador armazena na posição temporária 94 carrega da posição temporária 94 armazena acumulador em t
45 rem repete y	nenhuma	rem ignorado
50 goto 20	14 +4001	desvia para a posição 01
55 rem escreve resultado	nenhuma	rem ignorado
60 print t	15 +1195	escreve t na tela
99 end	16 +4300	termina a execução

Fig. 19.27 Instruções de SML produzidas depois da primeira passagem do compilador (parte 2 de 2).

Símbolo	Tipo	Posição
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Fig. 19.28 Tabela de símbolos para o programa da Fig. 19.27.

A maioria das instruções (*statements*) de Simple é convertida diretamente em uma única instrução (*instruction*) de SML. As exceções nesse programa são os comentários, a instrução **if/goto** na linha 20 e as instruções **let**. Os comentários não são traduzidos para a linguagem de máquina. Entretanto, o número de linha de um comentário (**rem**) é colocado na tabela de símbolos para o caso de se fazer referência ao número da linha em uma instrução **goto** ou em uma instru-

ção **if/goto**. A linha 20 do programa especifica que, se a condição **y == x** for verdadeira, o controle do programa é transferido para a linha 60. Como a linha 60 aparece mais adiante no programa, a primeira passagem do compilador ainda não colocou 60 na tabela de símbolos (os números de linha das instruções são colocados na tabela de símbolos apenas quando aparecem como a primeira *token* em uma instrução). Portanto, não é possível, nesse momento, determinar o operando da instrução *branch zero* de SML na posição 03 no *array* de instruções de SML. O compilador coloca 60 na posição 03 do *array flags* para indicar que a segunda passagem completará essa instrução.

Devemos monitorar a próxima posição da instrução no *array* de SML porque não há uma correspondência um para um entre as instruções de Simple e as instruções de SML. Por exemplo, a instrução **if/goto** da linha 20 é compilada para três instruções de SML. Toda vez que uma instrução é produzida, devemos incrementar o *contador de instruções* para a próxima posição no *array* de SML. Observe que o tamanho da memória do Simpletron pode ser um problema para programas Simple com muitas instruções, variáveis e constantes. É concebível que o compilador fique sem memória. Para testar esse caso, o programa deve conter um *contador de dados* para monitorar a posição em que a próxima variável ou constante será armazenada no *array* de SML. Se o valor do contador de instruções for maior que o valor do contador de dados, o *array* de SML está cheio. Nesse caso, o processo de compilação deve terminar e o compilador deve imprimir uma mensagem de erro que indica que ficou sem memória durante a compilação. Isso serve para enfatizar que, embora o programador seja liberado pelo compilador do ônus de gerenciar a memória, o próprio compilador deve cuidadosamente determinar a colocação de instruções e dados na memória e deve detectar erros como o esgotamento da memória durante o processo de compilação.

Uma visualização do processo de compilação, passo a passo

Vamos agora percorrer o processo de compilação para o programa Simple na Fig. 19.27. O compilador lê a primeira linha do programa

```
5 rem soma 1 a x
```

para a memória. A primeira *token* na instrução (o número da linha) é determinada com a classe **StreamTokenizer** (veja o Capítulo 10 para ler uma discussão sobre os métodos de manipulação de *strings* de Java). A *token* devolvida pelo **StreamTokenizer** é convertida em um inteiro com o método **static Integer.parseInt()**, de modo que o símbolo 5 possa ser localizado na tabela de símbolos. Se o símbolo não for localizado, ele é inserido na tabela de símbolos.

Estamos no começo do programa e essa é a primeira linha, e ainda não existe nenhum símbolo na tabela. Portanto, 5 é inserido na tabela de símbolos com o tipo **L** (número de linha) e atribuído à primeira posição no *array* de SML (00). Embora essa linha seja um comentário, um espaço na tabela de símbolos ainda é alocado para o número da linha (no caso de se fazer referência a ela em um **goto** ou em um **if/goto**). Nenhuma instrução de SML é gerada para uma instrução **rem**, de modo que o contador de instruções não é incrementado.

```
10 input x
```

é separada em *tokens* a seguir. O número de linha 10 é colocado na tabela de símbolos como tipo **L** e atribuído à primeira posição no *array* de SML (00 como o programa iniciou com um comentário, assim o contador de instruções atualmente é 00). O comando **input** indica que a próxima *token* é uma variável (apenas uma variável pode aparecer em uma instrução **input**). **input** corresponde diretamente a um código de operação de SML; portanto, o compilador simplesmente tem de determinar a posição de **x** no *array* de SML. O símbolo **x** não é encontrado na tabela de símbolos. Assim, ele é inserido na tabela de símbolos com a representação de **x** em Unicode, o tipo de dado **V** e alocado à posição 99 no *array* de SML (o armazenamento de dados inicia em 99 e é alocado de trás para diante). Agora pode ser gerado o código de SML para essa instrução. O código de operação 10 (o código de operação de leitura do SML) é multiplicado por 100 e a posição de **x** (como consta na tabela de símbolos) é adicionada para completar a instrução. A instrução é armazenada no *array* de SML na posição 00. O contador de instruções é incrementado por um, porque uma única instrução de SML foi produzida.

A instrução

```
15 rem testa se y == x
```

é separada em *tokens* a seguir. A tabela de símbolos é pesquisada em busca do número de linha 15 (que não é encontrado). O número da linha é inserido com o tipo **L** e atribuído à próxima posição no *array*, 01 (lembre-se de que as instruções **rem** não produzem código, assim o contador de instruções não é incrementado).

A instrução

```
20 if y == x goto 60
```

é separada em *tokens* a seguir. O número de linha 20 é inserido na tabela de símbolos e recebe o tipo **L** e a próxima posição no *array* de SML, 01. O comando **if** indica que uma condição deve ser avaliada. A variável **y** não é encontrada na

tabela de símbolos, assim é inserida e lhe são atribuídos o tipo **V** e a posição de SML 98. Em seguida, são geradas instruções de SML para avaliar a condição. Já que não há equivalente direto em SML a **if/goto**, ela deve ser simulada com um cálculo que usa **x** e **y** e desvia com base no resultado. Se **y** for igual a **x**, o resultado de subtrair **x** de **y** é zero; assim, a instrução **branch zero** pode ser utilizada com o resultado do cálculo para simular a instrução **if/goto**. O primeiro passo requer que **y** seja carregado (da posição 98 do SML) no acumulador. Isso produz a instrução **01 +2098**. Em seguida, **x** é subtraído do acumulador. Isso produz a instrução **02 +3199**. O valor no acumulador pode ser zero, positivo ou negativo. O operador **==**, de modo que queremos **branch zero** (desviar se o acumulador for igual a zero). Primeiro, a tabela de símbolos é pesquisada quanto à posição do desvio (60 nesse caso), que não é encontrada. Assim, 60 é colocado na posição 03 do array **flags** e a instrução **03 +4200** é gerada (não podemos adicionar a posição de desvio porque ainda não atribuímos uma posição à linha 60 no array de SML). O contador de instruções é incrementado para 04.

O compilador prossegue para instrução

25 rem incrementa y

O número de linha 25 é inserido na tabela de símbolos com o tipo **L** e atribuído à posição 04 da SML. O contador de instruções não é incrementado.

A instrução

30 let y = y + 1

é separada em *tokens*, o número de linha 30 é inserido na tabela de símbolos com o tipo **L** e é atribuído à posição 04 da SML. O comando **let** indica que a linha é uma instrução de atribuição. Primeiro, todos os símbolos na linha são inseridos na tabela de símbolos (se eles ainda não estiverem lá). O inteiro 1 é adicionado à tabela de símbolos com o tipo **C** e é atribuído à posição 97 da SML. Em seguida, o lado direito da atribuição é convertido de notação infixa para pós-fixa. Então, a expressão pós-fixa (**y 1 +**) é avaliada. O símbolo **y** é localizado na tabela de símbolos e sua posição de memória correspondente é inserida na pilha. O símbolo 1 também é localizado na tabela de símbolos e sua posição correspondente na memória é inserida na pilha. Quando o operador **+** é encontrado, o avaliador de pós-fixa remove da pilha o operando direito do operador, depois remove da pilha o operando esquerdo do operador **e**, então, produz as instruções de SML

04 +2098 (carrega y)
05 +3097 (soma 1)

O resultado da expressão é armazenado em uma posição temporária na memória (96) com a instrução

06 +2196 (armazena em temporária)

e a posição temporária é inserida na pilha. Agora que a expressão foi avaliada, o resultado deve ser armazenado em **y** (isto é, a variável no lado esquerdo de **=**). Assim, a posição temporária é carregada no acumulador e o acumulador é armazenado em **y** com as instruções

07 +2096 (carrega temporária)
08 +2198 (armazena em y)

O leitor imediatamente notará que as instruções de SML parecem ser redundantes. Discutiremos essa questão em breve.

A instrução

35 rem soma y a total

é separada em *tokens*, o número da linha 35 é inserido na tabela de símbolos com o tipo **L** e atribuído à posição 09.

A instrução

40 let t = t + y

é semelhante à linha 30. A variável **t** é inserida na tabela de símbolos com o tipo **V** e é atribuída à posição 95 da SML. As instruções seguem a mesma lógica e o mesmo formato que a linha 30, e as instruções **09 +2095**, **10 +3098**, **11 +2194**, **12 +2094** e **13 +2195** são geradas. Observe que o resultado de **t + y** é atribuído à posição temporária 94 antes de ser atribuído a **t** (95). Mais uma vez, o leitor notará que as instruções nas posições de memória 11 e 12 parecem ser redundantes. Novamente, discutiremos isso em breve.

A instrução

45 rem repete y

é um comentário, assim a linha 45 é adicionada à tabela de símbolos com o tipo **L** e é atribuída à posição 14 da SML.

A instrução

50 goto 20

transfere o controle para a linha 20. O número de linha 50 é inserido na tabela de símbolos com o tipo **L** e é atribuído à posição 14 da SML. O equivalente de **goto** em SML é a instrução **unconditional branch** (40), que transfere o controle

para uma posição específica da SML. O compilador pesquisa a tabela de símbolos procurando a linha **20** e descobre que corresponde à posição **01** da SML. O código de operação (**40**) é multiplicado por 100 e a posição **01** é adicionada para produzir a instrução **14 +4001**.

A instrução

55 rem escreve resultado

é um comentário, assim a linha **55** é inserida na tabela de símbolos com o tipo **L** e é atribuída à posição **15** da SML.

A instrução

60 print t

é uma instrução de saída. O número de linha **60** é inserido na tabela de símbolos com o tipo **L** e é atribuído à posição **15** da SML. O equivalente de **print** na SML é o código de operação **11** (*write*). A posição de **t** é determinada a partir da tabela de símbolos e adicionada ao resultado do código de operação multiplicado por 100.

A instrução

99 end

é a linha final do programa. O número de linha **99** é armazenado na tabela de símbolos com o tipo **L** e atribuído à posição **16** da SML. O comando **end** produz a instrução de SML **+4300** (**43** é *halt* na SML), que é escrita como a instrução final no *array* de memória de SML.

Isso completa a primeira passagem do compilador. Agora analisamos a segunda passagem. Pesquisa-se o *array flags* em busca de valores diferentes de **-1**. A posição **03** contém **60**, assim o compilador sabe que a instrução **03** está incompleta. O compilador completa a instrução procurando por **60** na tabela de símbolos, determinando sua posição e adicionando a posição à instrução incompleta. Nesse caso, a pesquisa determina que a linha **60** corresponde à posição **15** da SML, assim a instrução completada **03 +4215** é produzida, substituindo **03 +4200**. O programa Simple agora foi compilado com sucesso.

Para construir o compilador, você terá de realizar cada uma das seguintes tarefas:

- Modifique o programa do simulador Simpletron que você escreveu no Exercício 7.43 para pegar sua entrada de um arquivo especificado pelo usuário (veja o Capítulo 16). O simulador deve enviar seus resultados para um arquivo de saída em disco no mesmo formato que a saída em tela. Converta o simulador para ser um programa orientado a objetos. Em particular, torne cada parte do *hardware* um objeto. Organize os tipos de instrução em uma hierarquia de classes com herança. Depois execute o programa polimorficamente simplesmente dizendo para cada instrução executar a si própria com uma mensagem **executeInstruction**.
- Modifique o algoritmo de avaliação de notação infixa para pós-fixa do Exercício 19.12 para processar operandos inteiros de múltiplos dígitos e operandos nome de variável de uma única letra. (Dica: a classe **StreamTokenizer** pode ser utilizada para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de *strings* para inteiros com o método **parseInt** da classe **Integer**.) [Nota: a representação de dados da expressão pós-fixa deve ser alterada para suportar nomes de variáveis e constantes inteiras.]
- Modifique o algoritmo de avaliação pós-fixa para processar operandos inteiros de múltiplos dígitos e operandos nome de variável. Além disso, o algoritmo agora deve implementar o “gancho” já discutido, de modo que sejam produzidas instruções de SML, em vez de se avaliar diretamente a expressão. (Dica: pode-se usar a classe **StreamTokenizer** para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de *strings* para inteiros com o método **parseInt** da classe **Integer**.) [Nota: deve-se alterar a representação de dados da expressão pós-fixa para suportar nomes de variáveis e constantes de inteiras.]
- Construa o compilador. Incorpore as partes (b) e (c) para avaliar as expressões em instruções **let**. O programa deve conter um método que realiza a primeira passagem do compilador e um método que realiza a segunda passagem do compilador. Ambos os métodos podem chamar outros métodos para realizar essas tarefas. Torne seu compilador o mais orientado a objetos possível.

19.28 (*Otimizando o compilador de Simple*) Quando um programa é compilado e convertido para SML, gera-se um conjunto de instruções. Certas combinações de instruções freqüentemente se repetem, normalmente em triplas denominadas *produções*. Uma produção normalmente consiste em três instruções, como *carregar*, *adicionar* e *armazenar*. Por exemplo, a Fig. 19.29 ilustra cinco das instruções de SML que foram produzidas na compilação do programa da Fig. 19.27. As primeiras três instruções são a produção que adiciona **1** a **y**. Observe que as instruções **06** e **07** armazem o valor do acumulador na posição temporária **96**, depois carregam o valor de volta no acumulador de modo que a instrução **08** possa armazenar o valor na posição **98**. Geralmente após uma produção há uma instrução de carga para a mesma posição que acabou de ser armazenada. Pode-se *otimizar* esse código eliminando-se a instrução de armazenar e a instrução

subsequente de carregar que opera sobre a mesma posição da memória, permitindo, assim, que o Simpletron rode o programa mais rápido. A Fig. 19.30 ilustra o SML otimizado para o programa da Fig. 19.27. Observe que há quatro instruções a menos no código otimizado – uma economia de 25% de espaço de memória.

1	04	+2098	(carrega)
2	05	+3097	(soma)
3	06	+2196	(armazena)
4	07	+2096	(carrega)
5	08	+2198	(armazena)

Fig. 19.29 Código não-otimizado do programa da Fig. 19.25.

Programa Simple	Posição e instrução na SML	Descrição
5 rem soma 1 a x	nenhuma	rem ignorado
10 input x	00 +1099	lê x para a posição 99
15 rem testa se y == x	nenhuma	rem ignorado
20 if y == x goto 60	01 +2098 02 +3199 03 +4211	carrega y (98) no acumulador subtrai x (99) do acumulador desvia para a posição 11 se zero
25 rem incrementa y	nenhuma	rem ignorado
30 let y = y + 1	04 +2098 05 +3097 06 +2198	carrega y no acumulador soma 1 (97) ao acumulador armazena o acumulador em y (98)
35 rem soma y a total	nenhuma	rem ignorado
40 let t = t + y	07 +2096 08 +3098 09 +2196	carrega t da posição (96) adiciona y (98) ao acumulador armazena acumulador em t (96)
45 rem repete y	nenhuma	rem ignorado
50 goto 20	10 +4001	desvia para a posição 01
55 rem escreve resultado	nenhuma	rem ignorado
60 print t	11 +1196	escreve t (96) na tela
99 end	12 +4300	termina a execução

Fig. 19.30 Código otimizado para o programa da Fig. 19.27.

19.29 (Modificações do compilador de Simple) Realize as seguintes modificações no compilador Simple. Algumas dessas modificações também podem exigir modificações no programa simulador Simpletron escrito no Exercício 5.43.

- Permita que o operador de módulo (%) seja utilizado em instruções **let**. A Simpletron Machine Language deve ser modificada para incluir uma instrução de módulo.
- Permita exponenciação em uma instrução **let** com o símbolo ^ como operador de exponenciação. A Simpletron Machine Language deve ser modificada para incluir uma instrução de exponenciação.
- Permita que o compilador reconheça letras minúsculas e maiúsculas em instruções Simple (por exemplo, 'A' é equivalente a 'a'). Não é necessária nenhuma modificação no simulador Simpletron.
- Permita que as instruções **input** leiam os valores para múltiplas variáveis, como **input x, y**. Não é necessária nenhuma modificação no simulador Simpletron para realizar esse melhoramento no compilador Simple.

- e) Permita que o compilador dê saída de múltiplos valores com uma única instrução **print**, como **print a, b, c**. Não é necessária nenhuma modificação no simulador Simpletron para realizar esse melhoramento.
- f) Adicione capacidades de verificação de sintaxe ao compilador de modo que as mensagens de erro sejam enviadas para a saída quando erros de sintaxe forem encontrados em um programa Simple. Não é necessária nenhuma modificação no simulador Simpletron.
- g) Permita *arrays* de inteiros. Não é necessária nenhuma modificação no simulador Simpletron para realizar esse melhoramento.
- h) Permita sub-rotinas especificadas pelos comandos **gosub** e **return** de Simple. O comando **gosub** passa o controle do programa para uma sub-rotina e o comando **return** passa o controle de volta à instrução depois do **gosub**. Isso é semelhante a uma chamada de método em Java. A mesma sub-rotina pode ser chamada de muitos comandos **gosub** distribuídos por todo um programa. Não é necessária nenhuma modificação no simulador Simpletron.
- i) Permita estruturas de repetição na forma

```
for x = 2 to 10 step 2
    Instruções Simple
next
```

Essa instrução **for** faz um laço de 2 a 10 com um incremento de 2. A linha **next** marca o fim do corpo da linha **for**. Não é necessária nenhuma modificação no simulador Simpletron.

- j) Permita estruturas de repetição da forma

```
for x = 2 to 10
    Instruções Simple
next
```

Essa instrução **for** faz um laço de 2 a 10 com um incremento padrão de 1. Não é necessária nenhuma modificação no simulador Simpletron.

- k) Permita que o compilador processe entrada e saída de *strings*. Isso exige que o simulador Simpletron seja modificado para processar e armazenar valores de *strings*. [Dica: cada palavra do Simpletron pode ser dividida em duas partes, cada uma armazenando um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o equivalente decimal de um caractere Unicode. Adicione uma instrução de linguagem de máquina para imprimir um *string* que inicia em uma certa posição da memória do Simpletron. A primeira metade da palavra nessa posição é uma contagem do número de caracteres no *string* (isto é, o comprimento do *string*). Cada meia palavra sucessiva contém um caractere Unicode expresso como dois dígitos decimais. A instrução de linguagem de máquina verifica o comprimento e imprime o *string*, traduzindo cada número de dois dígitos para seu caractere equivalente.]
- l) Permita que o compilador processe valores em ponto flutuante além de inteiros. O Simulador Simpletron também deve ser modificado para processar valores em ponto flutuante.

19.30 (*Um interpretador de Simple*) O interpretador é um programa que lê uma instrução de programa de uma linguagem de alto nível, determina a operação a ser realizada pela instrução e executa a operação imediatamente. O programa em linguagem de alto nível não é convertido em linguagem de máquina primeiro. Os interpretadores executam mais lentamente que os compiladores porque cada instrução encontrada no programa que está sendo interpretado deve primeiro ser decifrada durante a execução. Se as instruções são contidas em um laço, as instruções são decifradas toda vez que são encontradas no laço. As primeiras versões da linguagem de programação Basic eram implementadas como interpretadores. A maioria dos programas Java é executada de forma interpretada.

Escreva um interpretador para a linguagem Simple discutida no Exercício 19.26. O programa deve utilizar o conversor de notação infixa para pós-fixa desenvolvido no Exercício 19.12 e o avaliador de pós-fixa desenvolvido no Exercício 19.13 para avaliar expressões em uma instrução **let**. As mesmas restrições impostas na linguagem Simple do Exercício 19.26 devem ser obedecidas nesse programa. Teste o interpretador com os programas Simple escritos no Exercício 19.26. Compare os resultados da execução desses programas no interpretador com os resultados da compilação dos programas Simple e de sua execução no simulador Simpletron construído no Exercício 7.43.

19.31 (*Inserção/exclusão em qualquer lugar em uma lista encadeada*) Nossa classe de lista encadeada permitia inserções e exclusões só no início e no fim da lista encadeada. Essas capacidades foram-nos convenientes quando utilizamos herança ou composição para produzir uma classe de pilha e uma classe de fila com uma quantidade mínima de código simplesmente reutilizando a classe de lista. As listas encadeadas são normalmente mais gerais que aquelas que fornecemos. Modifique a classe de lista encadeada que desenvolvemos neste capítulo para tratar inserções e exclusões em qualquer lugar na lista.

19.32 (*Listas e filas sem referências ao último nodo*) Nossa implementação de uma lista encadeada (Fig. 19.3) utilizou tanto um `firstNode` como um `lastNode`. O `lastNode` foi útil para os métodos `insertAtBack` e `removeFromBack` da classe `List`. O método `insertAtBack` corresponde ao método `enqueue` da classe `Queue`.

Rescreva a classe `List` de modo que ela não utilize um `lastNode`. Portanto, quaisquer operações sobre o fim de uma lista devem começar pesquisando a lista desde o início. Isso afeta nossa implementação da classe `Queue` (Fig. 19.13)?

19.33 (*Desempenho da classificação e da pesquisa de árvore binária*) Um problema com a classificação de árvore binária é que a ordem em que os dados são inseridos afeta a forma da árvore – para a mesma coleção de dados, diferentes ordens podem produzir árvores binárias de formas significativamente diferentes. O desempenho dos algoritmos de classificação e de pesquisa de árvore binária é sensível à forma da árvore binária. Que forma teria uma árvore binária se seus dados fossem inseridos na ordem crescente? E na ordem decrescente? Que forma a árvore deveria ter para obter desempenho máximo na pesquisa?

19.34 (*Listas indexadas*) Como foi apresentado no texto, as listas encadeadas devem ser pesquisadas seqüencialmente. Para listas grandes, isso pode resultar em desempenho pobre. Uma técnica comum para melhorar o desempenho da pesquisa em listas é criar e manter um índice para a lista. O índice é um conjunto de referências para lugares-chave na lista. Por exemplo, o aplicativo que pesquisa uma lista grande de nomes pode melhorar o desempenho criando um índice com 26 entradas – uma para cada letra do alfabeto. Assim, a operação de pesquisa para um sobrenome que inicia com ‘Y’ pesquisaria primeiro o índice para determinar onde as entradas ‘Y’ iniciam e, então, “saltaria” para a lista nesse ponto e pesquisaria linearmente até que o nome desejado fosse localizado. Isso seria muito mais rápido que pesquisar a lista encadeada desde o início. Utilize a classe `List` da Fig. 19.3 como base para uma classe `IndexedList`.

Escreva um programa que demonstra a operação de listas indexadas. Certifique-se de incluir os métodos `insertInIndexedList`, `searchIndexedList` e `deleteFromIndexedList`.

20

Pacote de utilitários Java e manipulação de *bits*

Objetivos

- Entender contêineres como as classes `Vector` e `Stack` e a interface `Enumeration`.
- Ser capaz de criar objetos `Hashtable` e tabelas de `hash` persistentes chamadas de objetos `Properties`.
- Entender geração de números aleatórios com instâncias da classe `Random`.
- Utilizar manipulação de *bits* e objetos `BitSet`.

Nada pode ter valor sem ser um objeto de utilidade.
Karl Marx

Já entrei no Quem é Quem, e sei o que é o quê, mas é a primeira vez que entro em um dicionário.

Mae West

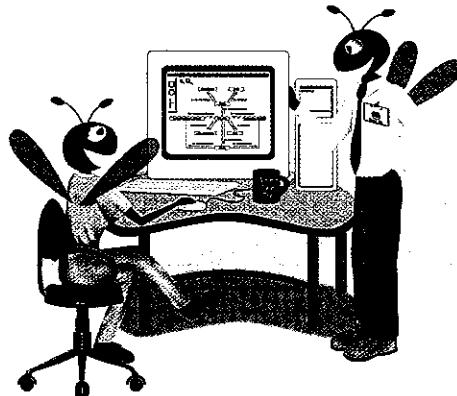
*O! many a shaft at random sent
Finds mark the archer little meant!*
Sir Walter Scott

*There was the Door to which I found no Key;
There was the Veil through which I might not see.*
Edward FitzGerald, *The Rubáiyát of Omar Khayyám*, st. 32

“É um tipo pobre de memória que só funciona de trás para frente”, observou a Rainha.

Lewis Carroll [Charles Lutwidge Dodgson]

Não é pela idade mas pela capacidade que se adquire a sabedoria.
Tito Márcio Plauto, *Trinummus*, ato II, cena ii, l. 88



Sumário do capítulo

- 20.1 Introdução
- 20.2 A classe `Vector` e a interface `Enumeration`
- 20.3 A classe `Stack`
- 20.4 A classe `Dictionary`
- 20.5 A classe `Hashtable`
- 20.6 A classe `Properties`
- 20.7 A classe `Random`
- 20.8 Manipulação de *bits* e os operadores sobre *bits*
- 20.9 A classe `BitSet`

Resumo • *Terminologia* • *Exercícios de auto-revisão* • *Respostas aos exercícios de auto-revisão* • *Exercícios*

20.1 Introdução

Este capítulo discute diversas classes utilitárias e interfaces do pacote `java.util`, inclusive a classe `Vector`, a interface `Enumeration`, a classe `Stack`, a classe `Dictionary`, a classe `Hashtable`, a classe `Properties`, a classe `Random` e a classe `BitSet`.

Os programas usam a classe `Vector` para criar objetos semelhantes a *arrays* que podem crescer e encolher dinamicamente conforme mudam os requisitos de armazenamento de dados do programa. Analisamos a interface `Enumeration`, que permite a um programa iterar percorrendo os elementos de um contêiner como um `Vector`.

A classe `Stack` oferece operações convencionais de pilhas, `push` e `pop`, além de várias outras que não analisamos no Capítulo 19.

A classe `Dictionary` é uma classe `abstract` que fornece uma estrutura para armazenar dados com campos de chave em tabelas e recuperar esses dados. Este capítulo examina a teoria de “*hashing*”, uma técnica para armazenar e recuperar rapidamente as informações de tabelas, e demonstra a construção e manipulação de tabelas de *hash* com a classe `Hashtable` de Java. Além disso, o capítulo analisa a classe `Properties`, que fornece suporte a tabelas de *hash* persistentes – que podem ser escritas em um arquivo com um fluxo de saída e lidas de um arquivo com um fluxo de entrada.

A classe `Random` fornece uma coleção mais rica de recursos para números aleatórios do que os disponíveis com o método estático `random` da classe `Math`.

O capítulo apresenta uma extensa discussão sobre operadores de manipulação de *bits*, seguida por uma discussão da classe `BitSet` que permite a criação de objetos semelhantes a *arrays* de *bits* para configurar e recuperar *bits* isolados.

O Capítulo 21 apresenta uma estrutura para manipular grupos de objetos denominados de *coleções*. Objetos dos tipo `Vector`, `Stack` e `Hashtable` são coleções.

20.2 A classe `Vector` e a interface `Enumeration`

Na maioria das linguagens de programação, incluindo Java, os *arrays* convencionais têm tamanho fixo – eles não podem crescer ou encolher em resposta a alterações nos requisitos de armazenamento de um aplicativo. A classe `Vector` de Java fornece os recursos de estruturas de dados semelhantes a *arrays* que podem redimensionar dinamicamente a si mesmas.

A qualquer momento, `Vector` contém um certo número de elementos menor que ou igual à sua *capacidade*. A capacidade é o espaço que foi reservado para os elementos do `Vector`. Se o `Vector` necessita de mais capacidade, ele cresce com um *incremento de capacidade* que você especifica ou com um *default* assumido pelo sistema. Se você não especificar um incremento de capacidade, o sistema dobrará o tamanho do `Vector` toda vez que for necessária capacidade adicional.



Dica de desempenho 20.1

Inserir elementos adicionais em um **Vector** cujo tamanho atual é menor que sua capacidade é uma operação relativamente rápida.



Dica de desempenho 20.2

Inserir um elemento em um **Vector** que precisa crescer mais para acomodar o novo elemento é uma operação relativamente lenta.



Dica de desempenho 20.3

O incremento de capacidade default dobra o tamanho do **Vector**. Isto pode parecer um desperdício de armazenamento, mas na realidade é uma maneira eficiente de muitos **Vectors** crescerem rapidamente para estar "próximos do tamanho certo". Isso é muito mais eficiente em termos de tempo do que fazer o **Vector** crescer a cada vez apenas por tanto espaço quanto necessário para armazenar um único elemento. A desvantagem é que o **Vector** pode ocupar mais espaço do que ele precisa.



Dica de desempenho 20.4

Se a memória for um recurso escasso, utilize o método **trimToSize** de **Vector** para aparar a capacidade de um **Vector** ao tamanho exato do **Vector**. Isso otimiza a utilização de memória por um **Vector**. Entretanto, adicionar outro elemento ao **Vector** fará o **Vector** a crescer dinamicamente – esse ajuste não deixa nenhum espaço para o crescimento.

Os **Vectors** armazenam referências para **Objects**. Portanto, um programa pode armazenar referências para quaisquer objetos em um **Vector**. Para armazenar valores de tipos primitivos de dados em **Vectors**, utilize as classes empacotadoras de tipos (por exemplo, **Integer**, **Long**, **Float** etc.) do pacote **java.lang** para criar objetos que contêm os valores de tipo primitivo de dados.

A Fig. 20.1 demonstra a classe **Vector** e diversos dos seus métodos. O programa fornece um **JButton** que permite testar cada um dos métodos. O usuário pode digitar um **String** no **JTextField** fornecido, e depois pressionar um botão para ver o que o método faz. Cada operação exibe uma mensagem em um **JLabel** para indicar os resultados de cada operação.

```

1 // Fig. 20.1: VectorTest.java
2 // Testando a classe Vector do pacote java.util
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6 import java.awt.*;
7 import java.awt.event.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class VectorTest extends JFrame {
13     private JLabel statusLabel;
14     private Vector vector;
15     private JTextField inputField;
16
17     // configura a GUI para testar os métodos de Vector
18     public VectorTest()
19     {
20         super( "Vector Example" );
21
22         Container container = getContentPane();
23         container.setLayout( new FlowLayout() );
24
25         statusLabel = new JLabel();

```

Fig. 20.1 Demonstrando a classe **Vector** do pacote **java.util** (parte 1 de 6).

```

26     vector = new Vector( 1 );
27
28     container.add( new JLabel( "Enter a string" ) );
29
30     inputField = new JTextField( 10 );
31     container.add( inputField );
32
33     // botão para adicionar elemento a vector
34     JButton addButton = new JButton( "Add" );
35
36     addButton.addActionListener(
37
38         new ActionListener() {
39
40             public void actionPerformed( ActionEvent event )
41             {
42                 // adiciona um elemento a vector
43                 vector.addElement( inputField.getText() );
44                 statusLabel.setText( "Added to end: " +
45                     inputField.getText() );
46                 inputField.setText( "" );
47             }
48         }
49     ); // fim da chamada para addActionListener
50
51     container.add( addButton );
52
53     // botão para remover um elemento de vector
54     JButton removeButton = new JButton( "Remove" );
55
56     removeButton.addActionListener(
57
58         new ActionListener() {
59
60             public void actionPerformed( ActionEvent event )
61             {
62                 // remove o elemento de vector
63                 if ( vector.removeElement( inputField.getText() ) )
64                     statusLabel.setText( "Removed: " +
65                         inputField.getText() );
66                 else
67                     statusLabel.setText( inputField.getText() +
68                         " not in vector" );
69             }
70         }
71     ); // fim da chamada para addActionListener
72
73     container.add( removeButton );
74
75     // botão para obter o primeiro elemento de vector
76     JButton firstButton = new JButton( "First" );
77
78     firstButton.addActionListener(
79
80         new ActionListener() {
81
82             public void actionPerformed( ActionEvent event )
83             {
84                 // devolve o primeiro elemento de vector
85                 try {

```

Fig. 20.1 Demonstrando a classe `Vector` do pacote `java.util` (parte 2 de 6).

```
86         statusLabel.setText(
87             "First element: " + vector.firstElement() );
88     }
89
90     // captura a exceção se Vector está vazio
91     catch ( NoSuchElementException exception ) {
92         statusLabel.setText( exception.toString() );
93     }
94 }
95
96 ); // fim da chamada para addActionListener
97
98 container.add( firstButton );
99
100 // botão para obter o último elemento de vector
101 JButton lastButton = new JButton( "Last" );
102
103 lastButton.addActionListener(
104
105     new ActionListener() {
106
107         public void actionPerformed( ActionEvent event )
108         {
109             // devolve o último elemento de vector
110             try {
111                 statusLabel.setText(
112                     "Last element: " + vector.lastElement() );
113             }
114
115             // captura a exceção se Vector estiver vazio
116             catch ( NoSuchElementException exception ) {
117                 statusLabel.setText( exception.toString() );
118             }
119         }
120     }
121 ); // fim da chamada para addActionListener
122
123 container.add( lastButton );
124
125 // botão para determinar se vector está vazio
126 JButton emptyButton = new JButton( "Is Empty?" );
127
128 emptyButton.addActionListener(
129
130     new ActionListener() {
131
132         public void actionPerformed( ActionEvent event )
133         {
134             // determina se Vector está vazio
135             statusLabel.setText( vector.isEmpty() ?
136                 "Vector is empty" : "Vector is not empty" );
137         }
138     }
139 ); // fim da chamada para addActionListener
140
141 container.add( emptyButton );
142
143 // botão para determinar se vector contém a chave de pesquisa
144 JButton containsButton = new JButton( "Contains" );
145
```

Fig. 20.1 Demonstrando a classe Vector do pacote java.util (parte 3 de 6).

```

146     containsButton.addActionListener(
147
148         new ActionListener() {
149
150             public void actionPerformed( ActionEvent event )
151             {
152                 String searchKey = inputField.getText();
153
154                 // determina se Vector contém searchKey
155                 if ( vector.contains( searchKey ) )
156                     statusLabel.setText(
157                         "Vector contains " + searchKey );
158                 else
159                     statusLabel.setText(
160                         "Vector does not contain " + searchKey );
161             }
162         }
163     ); // fim da chamada para addActionListener
164
165     container.add( containsButton );
166
167     // botão para determinar a posição de um valor em vector
168     JButton locationButton = new JButton( "Location" );
169
170     locationButton.addActionListener(
171
172         new ActionListener() {
173
174             public void actionPerformed( ActionEvent event )
175             {
176                 // obtém posição de um objeto em Vector
177                 statusLabel.setText( "Element is at location " +
178                     vector.indexOf( inputField.getText() ) );
179             }
180         }
181     ); // fim da chamada para addActionListener
182
183     container.add( locationButton );
184
185     // botão para aparar o tamanho de vector
186     JButton trimButton = new JButton( "Trim" );
187
188     trimButton.addActionListener(
189
190         new ActionListener() {
191
192             public void actionPerformed( ActionEvent event )
193             {
194                 // remove elementos não-ocupados para economizar memória
195                 vector.trimToSize();
196                 statusLabel.setText( "Vector trimmed to size" );
197             }
198         }
199     );
200
201     container.add( trimButton );
202
203     // botão para exibir o tamanho e a capacidade de vector

```

Fig. 20.1 Demonstrando a classe Vector do pacote java.util (parte 4 de 6).

```

204     JButton statsButton = new JButton( "Statistics" );
205
206     statsButton.addActionListener(
207
208         new ActionListener() {
209
210             public void actionPerformed( ActionEvent event )
211             {
212                 // obtém o tamanho e a capacidade de Vector
213                 statusLabel.setText( "Size = " + vector.size() +
214                     "; capacity = " + vector.capacity() );
215             }
216         }
217     ); // fim da chamada para addActionListener
218
219     container.add( statsButton );
220
221     // botão para exibir o conteúdo de vector
222     JButton displayButton = new JButton( "Display" );
223
224     displayButton.addActionListener(
225
226         new ActionListener() {
227
228             public void actionPerformed( ActionEvent event )
229             {
230                 // usa Enumeration para exibir o conteúdo de Vector
231                 Enumeration enum = vector.elements();
232                 StringBuffer buf = new StringBuffer();
233
234                 while ( enum.hasMoreElements() )
235                     buf.append( enum.nextElement() ).append( " " );
236
237                 JOptionPane.showMessageDialog( null,
238                     buf.toString(), "Display",
239                     JOptionPane.PLAIN_MESSAGE );
240             }
241         }
242     ); // fim da chamada para addActionListener
243
244     container.add( displayButton );
245     container.add( statusLabel );
246
247     setSize( 300, 200 );
248     setVisible( true );
249
250 } // fim do construtor VectorTest
251
252 // executa o aplicativo
253 public static void main( String args[] )
254 {
255     VectorTest application = new VectorTest();
256
257     application.setDefaultCloseOperation(
258         JFrame.EXIT_ON_CLOSE );
259 }
260
261 } // fim da classe VectorTest

```

Fig. 20.1 Demonstrando a classe Vector do pacote java.util (parte 5 de 6).

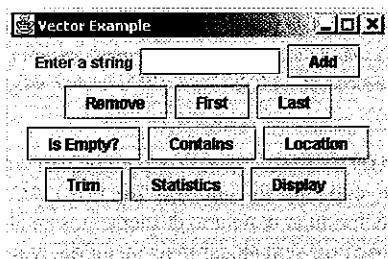


Fig. 20.1 Demonstrando a classe `Vector` do pacote `java.util` (parte 6 de 6).

O construtor do aplicativo cria um `Vector` (linha 26) com uma capacidade inicial de um elemento. Esse `Vector` dobrará de tamanho toda vez que precisar crescer para acomodar mais elementos. A classe `Vector` fornece três outros construtores. O construtor sem argumentos cria um `Vector` vazio com uma *capacidade inicial* de 10 elementos. O construtor que recebe dois argumentos cria um `Vector` com uma *capacidade inicial* especificada pelo primeiro argumento e um *incremento de capacidade* especificado pelo segundo argumento. Toda vez que o `Vector` precisar crescer, ele adicionará espaço para o número de elementos especificado no incremento de capacidade. O construtor que recebe uma `Collection` cria uma cópia dos elementos de uma coleção e os armazena em um `Vector`. No Capítulo 21, discutimos `Collections`.

A linha 43 chama o método `addElement` de `Vector` para adicionar o argumento ao final do `Vector`. Se necessário, o `Vector` aumenta a capacidade para acomodar o novo elemento. A classe `Vector` também fornece o método `insertElementAt` para inserir um elemento em uma posição especificada no `Vector` e o método `setElementAt` para configurar o elemento em uma posição específica no `Vector`. O método `insertElementAt` cria espaço para o novo elemento deslocando elementos. O método `setElementAt` substitui o elemento na posição especificada por seu argumento.

A linha 63 chama o método `removeElement` de `Vector` para remover do `Vector` a primeira ocorrência do argumento. O método devolve `true` se o elemento for localizado no `Vector`; caso contrário, `false`. Se o elemento for removido, todos os elementos depois desse elemento no `Vector` são deslocados uma posição em direção ao começo do `Vector` para preencher a posição do elemento removido. A classe `Vector` também fornece o método `removeAllElements` para remover todos os elementos do `Vector` e o método `removeElementAt` para remover o elemento no índice.

A linha 87 chama o método `firstElement` de `Vector` para devolver uma referência ao primeiro elemento no `Vector`. Esse método dispara uma `NoSuchElementException` se não houver nenhum elemento atualmente no `Vector`. A linha 112 chama o método `lastElement` de `Vector` para devolver uma referência ao último elemento no `Vector`. Esse método dispara uma `NoSuchElementException` se não houver nenhum elemento atualmente no `Vector`.

A linha 135 chama o método `isEmpty` de `Vector` para determinar se o `Vector` está vazio. O método devolve `true` se não houver nenhum elemento no `Vector`; caso contrário, o método devolve `false`.

A linha 155 chama o método `contains` de `Vector` para determinar se o `Vector` contém a `searchKey` especificada como argumento. O método `contains` devolve `true` se `searchKey` estiver em `Vector`; caso contrário, o método devolve `false`. O método `contains` utiliza o método `equals` de `Object` para determinar se a `searchKey` é igual a um dos elementos de `Vector`. Muitas classes sobrescrevem o método `equals` para fazer as comparações de uma maneira específica para aquelas classes. Por exemplo, a classe `String` define `equals` para comparar os caracteres individuais nos dois `Strings` que estão sendo comparados. Se o método `equals` não for sobreescrito, é usada a versão original da classe `Object`. Esta versão faz comparações com o operador `==` para determinar se duas referências se referem ao mesmo objeto na memória.

A linha 178 chama o método `indexOf` de `Vector` para determinar o índice da primeira posição no `Vector` que contém o argumento. O método devolve `-1` se o argumento não for encontrado no `Vector`. Uma versão sobrecarregada desse método recebe um segundo argumento que especifica o índice no `Vector` em que a pesquisa deve iniciar.

**Dica de desempenho 20.5**

Os métodos `contains` e `indexOf` de `Vector` fazem pesquisas lineares no conteúdo de um `Vector`, as quais são inefficientes para `Vectors` muito grandes. Se o programa procura elementos em uma coleção com muita frequência, pense em usar uma `Hashtable` (veja Seção 20.5) ou uma das implementações de `Map` da API Java Collections (veja Capítulo 21).

A linha 195 chama o método `trimToSize` de `Vector` para reduzir a capacidade do `Vector` para o número atual de elementos no `Vector` (isto é, o tamanho atual do `Vector`). As linhas 213 e 214 utilizam os métodos `size` e `capacity` de `Vector` para determinar o número de elementos atualmente no `Vector` e o número de elementos que pode ser armazenado no `Vector` sem alocar mais memória, respectivamente.

A linha 231 chama o método `elements` de `Vector` para devolver uma `Enumeration` que permite ao programa iterar através dos elementos do `Vector`. A `Enumeration` fornece dois métodos – `hasMoreElements` e `nextElement`. Na linha 234, o método `hasMoreElements` devolve `true` se existirem mais elementos no `Vector`. Na linha 235, o método `nextElement` devolve uma referência para o próximo elemento no `Vector`. Se não houver mais elementos, o método `nextElement` dispara uma `NoSuchElementException`.

Para obter informações completas sobre a classe `Vector` e seus outros métodos, veja a documentação *on-line* da Java API.

20.3 A classe Stack

No Capítulo 19, aprendemos como construir estruturas de dados fundamentais como listas encadeadas, pilhas, filas e árvores. Em um mundo de reutilização de *software*, em vez de construirmos estruturas de dados quando precisarmos delas, freqüentemente podemos tirar proveito de classes de estruturas de dados existentes. Nesta seção, investigamos a classe `Stack` no pacote de utilitários Java (`java.util`).

Na Seção 20.2, discutimos a classe `Vector`, que implementa um *array* dinamicamente redimensionável. A classe `Stack` estende a classe `Vector` para implementar uma estrutura de dados de pilha. Assim como `Vector`, a classe `Stack` armazena `Objects`. Para armazenar tipos primitivos de dados, deve-se utilizar a classe empacotadora de tipo apropriada para criar um objeto que contém o valor primitivo (`Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` ou `Double`). A Fig. 20.2 fornece uma GUI que permite testar cada um dos métodos de `Stack`.

```

1 // Fig. 20.2: StackTest.java
2 // Testando a classe Stack do pacote java.util
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.util.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class StackTest extends JFrame {
13     private JLabel statusLabel;
14     private JTextField inputField;
15     private Stack stack;
16
17     // cria GUI para manipular uma Stack
18     public StackTest()
19     {
20         super( "Stacks" );
21
22         Container container = getContentPane();
23
24         statusLabel = new JLabel();

```

Fig. 20.2 Demonstrando a classe `Stack` do pacote `java.util` (parte 1 de 4).

```
25     stack = new Stack();
26
27     container.setLayout( new FlowLayout() );
28     container.add( new JLabel( "Enter a string" ) );
29     inputField = new JTextField( 10 );
30     container.add( inputField );
31
32     // botão para colocar um objeto na pilha
33     JButton pushButton = new JButton( "Push" );
34
35     pushButton.addActionListener(
36
37         new ActionListener() {
38
39             public void actionPerformed( ActionEvent event )
40             {
41                 // coloca o objeto na pilha
42                 statusLabel.setText( "Pushed: " +
43                     stack.push( inputField.getText() ) );
44             }
45         }
46     );
47
48     container.add( pushButton );
49
50     // botão para remover o objeto do topo da pilha
51     JButton popButton = new JButton( "Pop" );
52
53     popButton.addActionListener(
54
55         new ActionListener() {
56
57             public void actionPerformed( ActionEvent event )
58             {
59                 // remove o elemento da pilha
60                 try {
61                     statusLabel.setText( "Popped: " + stack.pop() );
62                 }
63
64                 // processa a exceção se a pilha estiver vazia
65                 catch ( EmptyStackException exception ) {
66                     statusLabel.setText( exception.toString() );
67                 }
68             }
69         }
70     );
71
72     container.add( popButton );
73
74     // botão para examinar o elemento no topo da pilha
75     JButton peekButton = new JButton( "Peek" );
76
77     peekButton.addActionListener(
78
79         new ActionListener() {
80
81             public void actionPerformed( ActionEvent event )
82             {
83                 // examina o elemento no topo da pilha
84                 try {
```

Fig. 20.2 Demonstrando a classe Stack do pacote java.util (parte 2 de 4).

```

85         statusLabel.setText( "Top: " + stack.peek() );
86     }
87
88     // processa a exceção se a pilha estiver vazia
89     catch ( EmptyStackException exception ) {
90         statusLabel.setText( exception.toString() );
91     }
92 }
93 }
94 );
95
96 container.add( peekButton );
97
98 // botão para determinar se a pilha está vazia
99 JButton emptyButton = new JButton( "Is Empty?" );
100
101 emptyButton.addActionListener(
102
103     new ActionListener() {
104
105         public void actionPerformed( ActionEvent event )
106     {
107         // determina se a pilha está vazia
108         statusLabel.setText( stack.empty() ?
109             "Stack is empty" : "Stack is not empty" );
110     }
111 }
112 );
113
114 container.add( emptyButton );
115
116 // botão para determinar se a chave de pesquisa está na pilha
117 JButton searchButton = new JButton( "Search" );
118
119 searchButton.addActionListener(
120
121     new ActionListener() {
122
123         public void actionPerformed( ActionEvent event )
124     {
125         // procura o objeto especificado na pilha
126         String searchKey = inputField.getText();
127         int result = stack.search( searchKey );
128
129         if ( result == -1 )
130             statusLabel.setText( searchKey + " not found" );
131         else
132             statusLabel.setText( searchKey +
133                 " found at element " + result );
134     }
135 }
136 );
137
138 container.add( searchButton );
139
140 // botão para exibir o conteúdo da pilha
141 JButton displayButton = new JButton( "Display" );
142
143 displayButton.addActionListener(
144

```

Fig. 20.2 Demonstrando a classe Stack do pacote java.util (parte 3 de 4).

```

145     new ActionListener() {
146
147         public void actionPerformed( ActionEvent event )
148         {
149             // exibe o conteúdo da pilha
150             Enumeration enumeration = stack.elements();
151             StringBuffer buffer = new StringBuffer();
152
153             while ( enumeration.hasMoreElements() )
154                 buffer.append(
155                     enumeration.nextElement() ).append( " " );
156
157             JOptionPane.showMessageDialog( null,
158                 buffer.toString(), "Display",
159                 JOptionPane.PLAIN_MESSAGE );
160         }
161     }
162 );
163
164     container.add( displayButton );
165     container.add( statusLabel );
166
167     setSize( 675, 100 );
168     setVisible( true );
169 }
170
171 // executa o aplicativo
172 public static void main( String args[] )
173 {
174     StackTest application = new StackTest();
175
176     application.setDefaultCloseOperation(
177         JFrame.EXIT_ON_CLOSE );
178 }
179 } // fim da classe StackTest

```

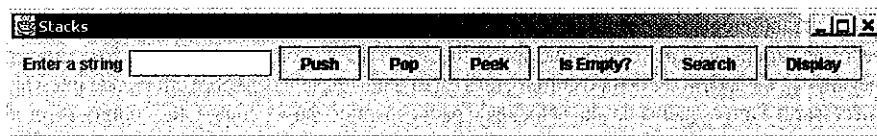


Fig. 20.2 Demonstrando a classe **Stack** do pacote **java.util** (parte 4 de 4).

A linha 43 cria uma **Stack** vazia. A linha 43 chama o método **push** de **Stack** para adicionar seu argumento ao topo da pilha. O método devolve uma referência **Object** para seu argumento.

A linha 61 chama o método **pop** de **Stack** para remover o elemento do topo da pilha. O método devolve uma referência **Object** para o elemento que foi removido. Se não houver nenhum elemento na **Stack**, o método **pop** dispara uma **EmptyStackException**.

A linha 85 chama o método **peek** de **Stack** para olhar o elemento do topo da pilha sem remover o elemento. O método **peek** devolve uma referência **Object** para o elemento.

A linha 108 chama o método **empty** de **Stack** para determinar se a pilha está vazia. Se ela estiver, o método devolve **true**; caso contrário, **false**.

A linha 127 chama o método **search** de **Stack** para determinar se seu argumento está na pilha. Se estiver, o método devolve a posição do elemento na pilha. Observe que o *elemento do topo é a posição 1*. Se o elemento não estiver na pilha, -1 é devolvido.

Toda a interface **public** da classe **Vector** faz, na realidade, parte da classe **Stack**, porque **Stack** herda de **Vector**. Para provar isso, nosso exemplo fornece um botão para exibir o conteúdo da pilha. Esse botão invoca o método **elements** para obter uma **Enumeration** da pilha. A **Enumeration** é então utilizada para percorrer os elementos da pilha.



Dica de teste e de depuração 20.1

Uma vez que `Stack` herda de `Vector`, o usuário pode realizar operações sobre objetos `Stack` que normalmente não são permitidas em estruturas convencionais de pilha de dados. Isso poderia corromper os elementos da `Stack` e destruir a integridade da `Stack`.

20.4 A classe Dictionary

A classe `Dictionary` mapeia *chaves* para *valores*. Ao pesquisar um valor em um `Dictionary`, o programa fornece uma chave e o `Dictionary` devolve o valor correspondente. `Dictionary` é uma classe `abstract`. Em particular, é a superclasse da classe `Hashtable` que discutiremos na Seção 20.5. A classe `Dictionary` fornece os métodos de interface `public` necessários para manter uma tabela de *pares chave/valor* em que as chaves representam as “linhas” na tabela e os valores são as “colunas” da tabela. Cada chave na tabela é única. A estrutura de dados é semelhante a um dicionário de palavras e definições – a palavra é a *chave* que é utilizada para pesquisar a definição (isto é, o *valor*).

O método `size` de `Dictionary` devolve o número de pares chave/valor no objeto `Dictionary`. O método `isEmpty` devolve `true` se o `Dictionary` estiver vazio; caso contrário, `false`. O método `keys` devolve uma `Enumeration` que o programa pode usar para iterar através das chaves de um `Dictionary`. O método `elements` devolve uma `Enumeration` que o programa pode usar para iterar através dos valores de um `Dictionary`. O método `get` devolve o objeto que corresponde a um dado valor de chave. O método `put` coloca um objeto associado a uma dada chave na tabela. O método `remove` remove um elemento correspondente a uma dada chave e devolve uma referência.

20.5 A classe Hashtable

As linguagens de programação orientada a objetos facilitam a criação de novos tipos de dados. Quando um programa cria objetos de tipos novos ou de tipos existentes, ele precisa gerenciar esses objetos eficientemente. Isso inclui armazenamento e recuperação de objetos. Armazenar e recuperar informações com *arrays* é eficiente se algum aspecto de seus dados corresponder diretamente ao valor da chave e se essas chaves forem únicas e compactadas. Se você tiver 100 empregados com CIC de nove algarismos e quiser armazenar e recuperar os dados dos empregados que usam o CIC como chave, isso exigiria nominalmente um *array* com 999.999.999 elementos, uma vez que há 999.999.999 números de nove algarismos distintos. Isso é complicado para praticamente todos os aplicativos cuja chave é baseada no CIC. Mas se você pudesse ter um *array* bem maior, você poderia obter maior desempenho armazenando e recuperando registros de empregados simplesmente utilizando o CIC como o índice do *array*.

Há uma enorme variedade de aplicativos que apresenta esse problema, ou seja, as chaves são do tipo errado (isto é, inteiros não-negativos), ou podem ser do tipo correto, mas estão muito espalhadas sobre um grande intervalo de valores.

O que é necessário é um esquema de alta velocidade para converter chaves como CIC, códigos de produtos em estoque e muitos outros em subscritos de *array* únicos. Então, quando um aplicativo precisa armazenar algo, o esquema poderia rapidamente converter a chave do aplicativo em um subscrito e o registro das informações poderia ser armazenado nessa posição no *array*. A recuperação é realizada da mesma maneira: uma vez que o aplicativo tenha uma chave através da qual quer recuperar o registro de dados, o aplicativo simplesmente aplica a conversão sobre a chave – isso produz o subscrito para acessar a posição do *array* em que os dados estão armazenados e em que os dados são recuperados.

O esquema que descrevemos aqui é a base de uma técnica chamada *hashing*. Por que esse nome? Porque quando convertemos uma chave em um subscrito de *array*, nós literalmente embaralhamos os *bits*, para formar um tipo de número “misturado”. O número realmente não tem nenhuma importância real além de sua utilidade para armazenar e recuperar o registro de dados com número particular.

Acontece uma *falla* no esquema quando ocorrem *colisões* (isto é, duas chaves diferentes “produzem hash para” a mesma célula (ou elemento) no *array*). Não podemos armazenar dois registros de dados diferentes no mesmo espaço, de modo que precisamos localizar uma posição alternativa para todos os registros depois do primeiro que produz *hash* para um subscrito de *array* particular. Há muitos esquemas para se fazer isso. Um é “fazer *hash* novamente” (isto é, replicar a transformação de *hashing* à chave para obter uma próxima célula candidata no *array*). O processo de *hashing* é projetado para ser aleatório; assim, a suposição é que com apenas alguns *hashes* uma célula disponível será localizada.

Outro esquema utiliza um *hash* para localizar a primeira célula candidata. Se essa célula estiver ocupada, pesquisam-se linearmente as células sucessivas até que uma célula disponível seja localizada. A recuperação funciona

da mesma maneira: a chave sofre *hash* uma vez, a célula apontada é verificada para ver se ela contém os dados desejados. Se ela os contiver, a pesquisa é concluída. Se não contiver, as células são pesquisadas linearmente até que os dados desejados sejam localizados.

A solução mais popular para colisões em tabela de *hash* é fazer cada célula da tabela ser um “*recipiente*” de *hash*, em geral uma lista encadeada de todos os pares chave/valor que produzem *hash* para essa célula. Essa é a solução que a classe **Hashtable** de Java (do pacote `java.util`) implementa.

Um fator que afeta o desempenho de esquemas de *hashing* é o chamado *fator de carga*. Trata-se da relação entre o número de células ocupadas na tabela de *hash* e o tamanho da tabela. Quanto mais a proporção se aproximar de 1, maior será a chance de colisões.

Dica de desempenho 20.6



O fator de carga em uma tabela de hash é um exemplo clássico de um compromisso entre espaço e tempo: aumentando o fator de carga, melhoramos a utilização da memória, mas o programa roda mais lentamente devido ao aumento das colisões de hashing. Diminuindo o fator de carga, melhoramos a velocidade do programa devido à redução das colisões de hashing, mas fazemos uma utilização pobre da memória porque uma parte maior da tabela de hash permanece vazia.

A complexidade de programar tabelas de *hash* adequadamente é grande demais para a maioria dos programadores casuais. Os alunos de ciência da computação estudam esquemas de *hashing* a fundo em cursos chamados “Estruturas de dados” e/ou “Algoritmos”. Reconhecendo o valor do *hashing* para a maioria dos programadores, Java fornece a classe **Hashtable** e alguns recursos relacionados para permitir que os programadores usem *hashing* sem ter de se preocupar com detalhes confusos.

Na verdade, a frase precedente é profundamente importante em nosso estudo de programação orientada a objetos. Como discutimos em capítulos anteriores, as classes encapsulam e ocultam a complexidade (isto é, detalhes da implementação) e oferecem interfaces amigáveis ao usuário. Construir classes para fazer isso adequadamente é uma das habilidades mais valorizadas no campo de programação orientada a objetos.

A Fig. 20.3 fornece uma GUI que permite testar muitos dos métodos de **Hashtable**. A linha 25 cria uma **Hashtable** vazia com uma capacidade *default* de 101 elementos e um fator de carga *default* de 0,75. Quando o número de posições ocupadas na **Hashtable** se tornar maior que a capacidade vezes o fator de carga, a tabela automaticamente crescerá. A classe **Hashtable** também fornece um construtor que recebe um argumento, especificando a capacidade, e um construtor que recebe dois argumentos, especificando a capacidade e o fator de carga, respectivamente.

```

1 // Fig. 20.3: HashtableTest.java
2 // Demonstra a classe Hashtable do pacote java.util.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.util.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class HashtableTest extends JFrame {
13     private JLabel statusLabel;
14     private Hashtable table;
15     private JTextArea displayArea;
16     private JTextField lastNameField;
17     private JTextField firstNameField;
18
19     // configura a GUI para demonstrar características de Hashtable
20     public HashtableTest()
21     {
22         super( "Hashtable Example" );
23

```

Fig. 20.3 Demonstrando a classe **Hashtable** (parte 1 de 6).

```

24     statusLabel = new JLabel();
25     table = new Hashtable();
26     displayArea = new JTextArea( 4, 20 );
27     displayArea.setEditable( false );
28
29     JPanel northSubPanel = new JPanel();
30
31     northSubPanel.add( new JLabel( "First name" ) );
32     firstNameField = new JTextField( 8 );
33     northSubPanel.add( firstNameField );
34
35     northSubPanel.add( new JLabel( "Last name (key)" ) );
36     lastNameField = new JTextField( 8 );
37     northSubPanel.add( lastNameField );
38
39     JPanel northPanel = new JPanel();
40     northPanel.setLayout( new BorderLayout() );
41     northPanel.add( northSubPanel, BorderLayout.NORTH );
42     northPanel.add( statusLabel, BorderLayout.SOUTH );
43
44     JPanel southPanel = new JPanel();
45     southPanel.setLayout( new GridLayout( 2, 5 ) );
46     JButton putButton = new JButton( "Put" );
47
48     putButton.addActionListener(
49
50         new ActionListener() {
51
52             // adiciona novo par chave/valor à tabela hash
53             public void actionPerformed( ActionEvent event )
54             {
55                 Employee employee = new Employee(
56                     firstNameField.getText(),
57                     lastNameField.getText() );
58
59                 Object value =
60                     table.put( lastNameField.getText(), employee );
61
62                 // primeira vez que esta chave foi adicionada
63                 if ( value == null )
64                     statusLabel.setText(
65                         "Put: " + employee.toString() );
66
67                 // substituiu valor anterior para esta chave
68                 else
69                     statusLabel.setText(
70                         "Put: " + employee.toString() +
71                         ", Replaced: " + value.toString() );
72             }
73         });
74
75     southPanel.add( putButton );
76
77     // botão para obter valor para uma chave específica
78     JButton getButton = new JButton( "Get" );
79
80     getButton.addActionListener(
81
82         new ActionListener() {

```

Fig. 20.3 Demonstrando a classe Hashtable (parte 2 de 6).

```
84          // obtém valor para chave específica
85          public void actionPerformed( ActionEvent event )
86          {
87              Object value = table.get( lastNameField.getText() );
88
89              // encontrando valor para a chave
90              if ( value != null )
91                  statusLabel.setText(
92                      "Get: " + value.toString() );
93
94              // não encontrou valor para a chave
95              else
96                  statusLabel.setText(
97                      "Get: " + lastNameField.getText() +
98                      " not in table" );
99
100         }
101     );
102
103     southPanel.add( getButton );
104
105     // botão para remover o par chave/valor da tabela
106     JButton removeButton = new JButton( "Remove" );
107
108     removeButton.addActionListener(
109
110         new ActionListener() {
111
112             // remove o par chave/valor
113             public void actionPerformed( ActionEvent event )
114             {
115                 Object value =
116                     table.remove( lastNameField.getText() );
117
118                 // encontrou a chave
119                 if ( value != null )
120                     statusLabel.setText( "Remove: " +
121                         value.toString() );
122
123                 // não encontrou a chave
124                 else
125                     statusLabel.setText( "Remove: " +
126                         lastNameField.getText() + " not in table" );
127
128         }
129     );
130
131     southPanel.add( removeButton );
132
133     // botão para determinar se a tabela de hash está vazia
134     JButton emptyButton = new JButton( "Empty" );
135
136     emptyButton.addActionListener(
137
138         new ActionListener() {
139
140             // determina se a tabela de hash está vazia
141             public void actionPerformed( ActionEvent event )
142             {
143
```

Fig. 20.3 Demonstrando a classe Hashtable (parte 3 de 6).

```

144         statusLabel.setText( "Empty: " + table.isEmpty() );
145     }
146   }
147 );
148
149 southPanel.add( emptyButton );
150
151 // botão para determinar se a tabela de hash contém a chave
152 JButton containsKeyButton = new JButton( "Contains key" );
153
154 containsKeyButton.addActionListener(
155
156     new ActionListener() {
157
158         // determina se a tabela de hash contém a chave
159         public void actionPerformed( ActionEvent event )
160     {
161         statusLabel.setText( "Contains key: " +
162             table.containsKey( lastNameField.getText() ) );
163     }
164   );
165
166 southPanel.add( containsKeyButton );
167
168 // botão para limpar todo o conteúdo da tabela de hash
169 JButton clearButton = new JButton( "Clear table" );
170
171 clearButton.addActionListener(
172
173     new ActionListener() {
174
175         // limpa o conteúdo da tabela de hash
176         public void actionPerformed( ActionEvent event )
177     {
178         table.clear();
179         statusLabel.setText( "Clear: Table is now empty" );
180     }
181   );
182
183 southPanel.add( clearButton );
184
185 // botão para exibir elementos da tabela de hash
186 JButton listElementsButton = new JButton( "List objects" );
187
188 listElementsButton.addActionListener(
189
190     new ActionListener() {
191
192         // exibe elementos da tabela de hash
193         public void actionPerformed( ActionEvent event )
194     {
195         StringBuffer buffer = new StringBuffer();
196
197         for ( Enumeration enumeration = table.elements();
198             enumeration.hasMoreElements(); )
199             buffer.append(
200                 enumeration.nextElement() ).append( '\n' );
201
202
203

```

Fig. 20.3 Demonstrando a classe Hashtable (parte 4 de 6).

```
204         displayArea.setText( buffer.toString() );
205     }
206   }
207 );
208
209 southPanel.add( listElementsButton );
210
211 // botão para exibir as chaves da tabela de hash
212 JButton listKeysButton = new JButton( "List keys" );
213
214 listKeysButton.addActionListener(
215
216     new ActionListener() {
217
218         // exibe as chaves da tabela de hash
219         public void actionPerformed( ActionEvent event )
220     {
221         StringBuffer buffer = new StringBuffer();
222
223         for ( Enumeration enumeration = table.keys();
224             enumeration.hasMoreElements(); )
225             buffer.append(
226                 enumeration.nextElement() ).append( '\n' );
227
228         JOptionPane.showMessageDialog( null,
229             buffer.toString(), "Display",
230             JOptionPane.PLAIN_MESSAGE );
231     }
232   }
233 );
234
235 southPanel.add( listKeysButton );
236
237 Container container = getContentPane();
238 container.add( northPanel, BorderLayout.NORTH );
239 container.add( new JScrollPane( displayArea ),
240     BorderLayout.CENTER );
241 container.add( southPanel, BorderLayout.SOUTH );
242
243 setSize( 540, 300 );
244 setVisible( true );
245 }
246
247 // executa o aplicativo
248 public static void main( String args[] )
249 {
250     HashtableTest application = new HashtableTest();
251
252     application.setDefaultCloseOperation(
253         JFrame.EXIT_ON_CLOSE );
254 }
255
256 } // fim da classe HashtableTest
257
258 // classe Employee para representar nome e sobrenome
259 class Employee {
260     private String first, last;
261
262     // inicializa como Employee
263     public Employee( String firstName, String lastName )
```

Fig. 20.3 Demonstrando a classe Hashtable (parte 5 de 6).

```

264     {
265         first = firstName;
266         last = lastName;
267     }
268
269     // converte Employee para representação como String
270     public String toString()
271     {
272         return first + " " + last;
273     }
274
275 } // fim da classe Employee

```

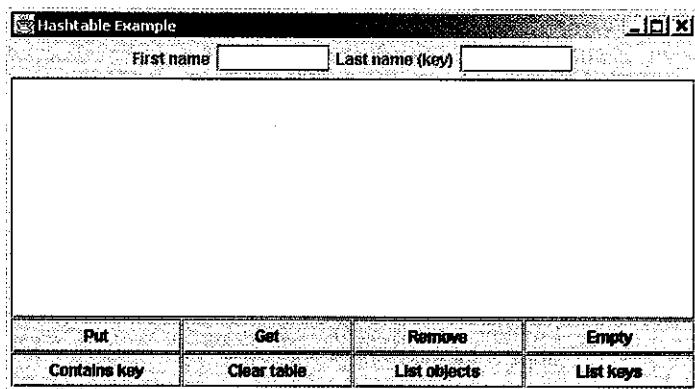


Fig. 20.3 Demonstrando a classe `Hashtable` (parte 6 de 6).

As linhas 59 e 60 chamam o método `put` de `Hashtable` para adicionar uma *chave* (o primeiro argumento) e um *valor* (o segundo argumento) à `Hashtable`. O método `put` devolve `null` se a chave não foi inserida na `Hashtable` anteriormente. Caso contrário, o método `put` devolve o valor original para aquela chave na `Hashtable`; isso ajuda o programa a gerenciar os casos em que se pretende substituir o valor armazenado para uma determinada chave. Se a chave ou o valor for `null`, ocorre uma `NullPointerException`.

A linha 88 chama o método `get` de `Hashtable` para localizar o valor associado à chave especificada como argumento. Se a chave estiver presente na tabela, `get` devolve uma referência `Object` para o valor correspondente; caso contrário, o método devolve `null`.

As linhas 116 e 117 chamam o método `remove` de `Hashtable` para remover um par chave/valor da tabela. O método devolve uma referência para o `Object` removido. Se não houver nenhum valor mapeado para a chave especificada, o método devolve `null`.

A linha 144 chama o método `isEmpty` de `Hashtable`, que devolve `true` se a `Hashtable` estiver vazia; caso contrário, ele devolve `false`.

A linha 162 chama o método `containsKey` de `Hashtable` para determinar se a chave especificada como argumento está na `Hashtable` (isto é, um valor é associado com essa chave). Caso esteja, o método devolve `true`; caso contrário, o método devolve `false`. A classe `Hashtable` também fornece o método `contains` para determinar se o `Object` especificado como argumento está na `Hashtable`.

A linha 179 chama o método `clear` de `Hashtable` para esvaziar o conteúdo da `Hashtable`. A linha 199 chama o método `elements` de `Hashtable` para obter uma `Enumeration` dos valores na `Hashtable`. A linha 223 chama o método `keys` de `Hashtable` para obter uma `Enumeration` das chaves na `Hashtable`.

Para obter mais informações sobre os métodos de `Hashtable`, veja a documentação da Java API.

20.6 A classe Properties

O objeto `Properties` é um objeto `Hashtable` persistente, que normalmente armazena pares chave/valor de `Strings` – partindo-se do princípio de que você usa os métodos `setProperty` e `getProperty` para manipu-

lar a tabela em vez dos métodos `put` e `get` de `Hashtable`. Por persistente, queremos dizer que o objeto `Hashtable` pode ser enviado para um fluxo de saída e dirigido para um arquivo, e depois ser lido de volta por um fluxo de entrada. Na verdade, a maioria dos objetos em Java pode agora ter saída e entrada com a serialização de objetos de Java (veja o Capítulo 16). A classe `Properties` estende a classe `Hashtable`, portanto os objetos `Properties` têm os métodos que discutimos na Fig. 20.3. As chaves e os valores em um objeto `Properties` devem ser do tipo `String`. A classe `Properties` fornece alguns métodos adicionais que são demonstrados na Fig. 20.4.

```
1 // Fig. 20.4: PropertiesTest.java
2 // Demonstra a classe Properties do pacote java.util.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.io.*;
8 import java.util.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 public class PropertiesTest extends JFrame {
14     private JLabel statusLabel;
15     private Properties table;
16     private JTextArea displayArea;
17     private JTextField valueField, nameField;
18
19     // configura a GUI para testar a tabela Properties
20     public PropertiesTest()
21     {
22         super( "Properties Test" );
23
24         // cria a tabela Properties
25         table = new Properties();
26
27         Container container = getContentPane();
28
29         // configura NORTH do BorderLayout da janela
30         JPanel northSubPanel = new JPanel();
31
32         northSubPanel.add( new JLabel( "Property value" ) );
33         valueField = new JTextField( 10 );
34         northSubPanel.add( valueField );
35
36         northSubPanel.add( new JLabel( "Property name (key)" ) );
37         nameField = new JTextField( 10 );
38         northSubPanel.add( nameField );
39
40         JPanel northPanel = new JPanel();
41         northPanel.setLayout( new BorderLayout() );
42         northPanel.add( northSubPanel, BorderLayout.NORTH );
43
44         statusLabel = new JLabel();
45         northPanel.add( statusLabel, BorderLayout.SOUTH );
46
47         container.add( northPanel, BorderLayout.NORTH );
48
49         // configura CENTER do BorderLayout da janela
50         displayArea = new JTextArea( 4, 35 );
51         container.add( new JScrollPane( displayArea ),
```

Fig. 20.4 Demonstrando a classe `Properties` (parte 1 de 5).

```

52         BorderLayout.CENTER );
53
54     // configura SOUTH do BorderLayout da janela
55     JPanel southPanel = new JPanel();
56     southPanel.setLayout( new GridLayout( 1, 5 ) );
57
58     // botão para colocar um par nome/valor em uma tabela Properties
59     JButton putButton = new JButton( "Put" );
60
61     putButton.addActionListener(
62
63         new ActionListener() {
64
65             // coloca um par nome/valor em uma tabela Properties
66             public void actionPerformed( ActionEvent event )
67             {
68                 Object value = table.setProperty(
69                     nameField.getText(), valueField.getText() );
70
71                 if ( value == null )
72                     showstatus( "Put: " + nameField.getText() +
73                         " " + valueField.getText() );
74
75                 else
76                     showstatus( "Put: " + nameField.getText() +
77                         " " + valueField.getText() +
78                         "; Replaced: " + value.toString() );
79
80                 listProperties();
81             }
82         }
83     ); // fim da chamada para addActionListener
84
85     southPanel.add( putButton );
86
87     // botão para esvaziar o conteúdo da tabela Properties
88     JButton clearButton = new JButton( "Clear" );
89
90     clearButton.addActionListener(
91
92         new ActionListener() {
93
94             // usa o método clear para esvaziar a tabela
95             public void actionPerformed( ActionEvent event )
96             {
97                 table.clear();
98                 showstatus( "Table in memory cleared" );
99                 listProperties();
100            }
101        }
102    ); // fim da chamada para addActionListener
103
104    southPanel.add( clearButton );
105
106    // botão para obter o valor de uma propriedade
107    JButton getPropertyButton = new JButton( "Get property" );
108
109    getPropertyButton.addActionListener(
110
111        new ActionListener() {

```

Fig. 20.4 Demonstrando a classe Properties (parte 2 de 5).

```
112          // usa o método getProperty para obter um valor de propriedade
113      public void actionPerformed( ActionEvent event )
114      {
115          Object value = table.getProperty(
116              nameField.getText() );
117
118          if ( value != null )
119              showstatus( "Get property: " +
120                  nameField.getText() + " " +
121                  value.toString() );
122
123          else
124              showstatus( "Get: " + nameField.getText() +
125                  " not in table" );
126
127          listProperties();
128      }
129  }
130 }
131 ); // fim da chamada para addActionListener
132
133 southPanel.add( getPropertyButton );
134
135 // botão para gravar conteúdo da tabela Properties em arquivo
136 JButton saveButton = new JButton( "Save" );
137
138 saveButton.addActionListener(
139
140     new ActionListener() {
141
142         // usa o método save para colocar conteúdo no arquivo
143         public void actionPerformed( ActionEvent event )
144         {
145             // salva o conteúdo da tabela
146             try {
147                 FileOutputStream output =
148                     new FileOutputStream( "props.dat" );
149
150                 table.store( output, "Sample Properties" );
151                 output.close();
152
153                 listProperties();
154             }
155
156             // processa problemas com a saída para o arquivo
157             catch( IOException ioException ) {
158                 ioException.printStackTrace();
159             }
160         }
161     }
162 ); // fim da chamada para addActionListener
163
164 southPanel.add( saveButton );
165
166 // botão para carregar o conteúdo da tabela Properties de um arquivo
167 JButton loadButton = new JButton( "Load" );
168
169 loadButton.addActionListener(
170
171     new ActionListener() {
```

Fig. 20.4 Demonstrando a classe `Properties` (parte 3 de 5).

```

172
173     // usa o método load para ler o conteúdo do arquivo
174     public void actionPerformed( ActionEvent event )
175     {
176         // carrega conteúdo da tabela
177         try {
178             FileInputStream input =
179                 new FileInputStream( "props.dat" );
180
181             table.load( input );
182             input.close();
183             listProperties();
184         }
185
186         // processa problemas com leitura do arquivo
187         catch( IOException ioException ) {
188             ioException.printStackTrace();
189         }
190     }
191 }
192 ); // fim da chamada para addActionListener
193
194 southPanel.add( loadButton );
195
196 container.add( southPanel, BorderLayout.SOUTH );
197
198 setSize( 550, 225 );
199 setVisible( true );
200 }
201
202 // envia valores das propriedades para saída
203 public void listProperties()
204 {
205     StringBuffer buffer = new StringBuffer();
206     String name, value;
207
208     Enumeration enumeration = table.propertyNames();
209
210     while ( enumeration.hasMoreElements() ) {
211         name = enumeration.nextElement().toString();
212         value = table.getProperty( name );
213
214         buffer.append( name ).append( '\t' );
215         buffer.append( value ).append( '\n' );
216     }
217
218     displayArea.setText( buffer.toString() );
219 } // fim do método ListProperties
220
221 // exibe String no rótulo statusLabel
222 public void showstatus( String s )
223 {
224     statusLabel.setText( s );
225 }
226
227 // executa o aplicativo
228 public static void main( String args[] )
229 {
230     PropertiesTest application = new PropertiesTest();
231 }
```

Fig. 20.4 Demonstrando a classe Properties (parte 4 de 5).

```

232     application.setDefaultCloseOperation(
233         JFrame.EXIT_ON_CLOSE );
234     }
235
236 } // fim da classe PropertiesTest

```

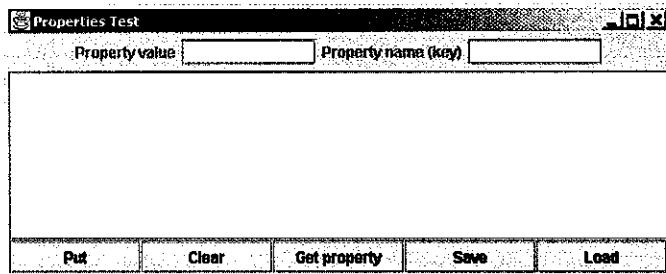


Fig. 20.4 Demonstrando a classe `Properties` (parte 5 de 5).

A linha 25 utiliza o construtor sem argumentos para criar uma tabela `Properties` vazia sem propriedades *default*. A classe `Properties` também fornece um construtor sobrecarregado que recebe uma referência para um objeto `Properties` que contém valores de propriedades *default*.

As linhas 68 e 69 chamam o método `setProperty` de `Properties` para armazenar um valor para a chave especificada. Se a chave não existir na tabela, `setProperty` devolve `null`; caso contrário, ele devolve o valor anterior para aquela chave.

As linhas 116 e 117 chamam o método `getProperty` de `Properties` para localizar o valor associado à chave especificada. Se a chave não for localizada nesse objeto `Properties`, `getProperty` usa o objeto `Properties default` (se houver um). O processo continua recursivamente até que não haja mais os objetos `Properties default` (lembre-se de que todo objeto `Properties` pode ser inicializado com um objeto `Properties default`), ponto em que `null` é devolvido. É fornecida uma versão sobreescrita desse método que recebe dois argumentos, e o segundo deles é o valor *default* a devolver se `getProperty` não puder localizar a chave.

A linha 150 chama o método `store` de `Properties` para salvar o conteúdo do objeto `Properties` no objeto `OutputStream` especificado como primeiro argumento (nesse caso um `FileOutputStream`). O argumento `String` é uma descrição do objeto `Properties`. A classe `Properties` também fornece o método `list`, que recebe um argumento `PrintStream`. Esse método é útil para exibir o conjunto de propriedades.



Dica de teste e depuração 20.2

Utilize o método `list` de `Properties` para exibir o conteúdo de um objeto `Properties` para fins de depuração.

A linha 181 chama o método `load` de `Properties` para restaurar o conteúdo do objeto `Properties` a partir do `InputStream` especificado como primeiro argumento (nesse caso um `FileInputStream`).

A linha 208 chama o método `propertyNames` de `Properties` para obter uma `Enumeration` dos nomes de propriedades. O valor de cada propriedade pode ser determinado com o método `getProperty`.

20.7 A classe Random

Discutimos a geração de números aleatórios no Capítulo 6, em que utilizamos o método `random` da classe `Math`. Java fornece muitos recursos adicionais para a geração de números aleatórios na classe `Random`. Aqui examinamos brevemente as chamadas da API.

Pode-se criar um novo gerador de números aleatórios com

```
Random r = new Random();
```

Essa forma usa a hora como uma semente diferente para seu gerador de números aleatórios toda vez que ele é chamado e, assim, gera seqüências diferentes de números aleatórios a cada vez.

Para criar um gerador de números pseudo-aleatórios com “repetitividade”, utilize

```
Random r = new Random( seedValue );
```

O argumento `seedValue` (do tipo `long`) é usado no cálculo de um número aleatório para “alimentar” o gerador de números aleatórios. Se o mesmo `seedValue` for usado a cada vez, o objeto `Random` produz a mesma seqüência de números aleatórios.



Dica de teste e depuração 20.3

Enquanto um programa está em desenvolvimento, utilize a forma `Random(seedValue)` que produz uma seqüência de números aleatórios que pode ser repetida. Se ocorrer um erro, corrija o erro e teste com o mesmo `seedValue`; isso permite reconstruir exatamente a mesma seqüência de números aleatórios que ocasionou o erro. Uma vez que os erros tenham sido removidos, utilize a forma `Random()` que gera uma nova seqüência de números aleatórios toda vez que o programa é executado.

A chamada

```
r.setSeed( seedValue );
```

inicializa novamente o valor da semente de `r` a qualquer momento.

As chamadas

```
r.nextInt()
r.nextLong()
```

geram inteiros aleatórios distribuídos uniformemente. Você pode utilizar `Math.abs` para receber o valor absoluto do número produzido por `nextInt`, obtendo, assim, um número no intervalo entre zero e aproximadamente 2 bilhões. Depois, utilize o operador `%` para dimensionar o número. Por exemplo, para lançar um dado de seis faces, se você dimensionar com um 6, obterá um número no intervalo 0 a 5. Assim, simplesmente desloque esse valor adicionando 1 para produzir um número no intervalo 1 a 6. A expressão é a seguinte:

```
Math.abs( r.nextInt() ) % 6 + 1
```

As chamadas

```
r.nextFloat()
r.nextDouble()
```

geram valores uniformemente distribuídos no intervalo $0,0 \leq x < 1,0$.

A chamada

```
r.nextGaussian()
```

gera um valor `double` com uma densidade de probabilidade de uma distribuição *gaussiana* – isto é, “normal” (média de 0,0 e desvio-padrão de 1,0).

20.8 Manipulação de bits e os operadores sobre bits

Java oferece muitos recursos para a manipulação de *bits* para os programadores que precisam descer ao chamado nível dos “*bits e bytes*”. Sistemas operacionais, *softwares* de equipamento de teste, *softwares* de rede e muitos outros tipos de *softwares* exigem que o programador se comunique “diretamente com o *hardware*”. Nesta e na próxima seção, discutimos os recursos de manipulação de *bits* de Java. Apresentamos os operadores sobre *bits* de Java e demonstramos sua utilização em exemplos de código ativo.

Todos os dados são representados internamente pelos computadores como seqüências de *bits*. Cada *bit* pode assumir o valor 0 ou 1. Na maioria dos sistemas, uma seqüência de 8 *bits* forma um *byte* – a unidade de armazenamento padrão para uma variável do tipo `byte`. Outros tipos de dados são armazenados em quantidades de *bytes* maiores. Os operadores sobre *bits* são utilizados para manipular os *bits* de operandos integrais (isto é, aqueles do tipo `byte`, `char`, `short`, `int` e `long`).

Observe que as discussões de operadores sobre *bits* nesta seção mostram as representações binárias dos operandos inteiros. Para obter uma explicação detalhada sobre o sistema de numeração binária (também chamado base 2), veja o Apêndice E.

Os operadores sobre *bits* são os seguintes: *E sobre bits* (`&`), *OU inclusivo sobre bits* (`|`), *OU exclusivo sobre bits* (`^`), *deslocamento para a esquerda* (`<<`), *deslocamento para a direita com extensão de sinal* (`>>`), *des-*

locamento para a direita com extensão de zero (>>>) e complemento (~). Os operadores E sobre *bits*, OU inclusivo sobre *bits* e OU exclusivo sobre *bits* comparam seus dois operandos *bit* por *bit*. O operador E sobre *bits* configura cada *bit* no resultado como 1 se o *bit* correspondente em ambos os operandos for 1. O operador OU inclusivo sobre *bits* configura cada *bit* no resultado como 1 se o *bit* correspondente em qualquer operando (ou nos dois) for 1. O operador OU exclusivo sobre *bits* configura cada *bit* no resultado como 1 se o *bit* correspondente em somente um operando for 1. O operador de deslocamento para a esquerda desloca os *bits* de seu operando esquerdo para a esquerda pelo número de *bits* especificado em seu operando direito. O operador de deslocamento para a direita com extensão de sinal desloca os *bits* em seu operando esquerdo para a direita pelo número de *bits* especificado em seu operando direito – se o operando esquerdo for negativo, 1s são inseridos a partir da esquerda; caso contrário, são inseridos 0s a partir da esquerda. O operador de deslocamento para a direita com extensão de zero desloca os *bits* em seu operando esquerdo para a direita pelo número de *bits* especificado em seu operando direito – são inseridos 0s a partir da esquerda. O operador de complemento sobre *bits* configura todos os *bits* 0 em seu operando como 1 no resultado e configura todos os *bits* 1 como 0 no resultado. Discussões detalhadas sobre cada operador sobre *bits* aparecem nos exemplos a seguir. Os operadores sobre *bits* estão resumidos na Fig. 20.5.

Operador	Nome	Descrição
&	E sobre <i>bits</i>	Os <i>bits</i> no resultado são configurados como 1 se os <i>bits</i> correspondentes nos dois operandos forem ambos 1.
	OU inclusivo sobre <i>bits</i>	Os <i>bits</i> no resultado são configurados como 1 se pelo menos um dos <i>bits</i> correspondentes nos dois operandos for 1.
^	OU exclusivo sobre <i>bits</i>	Os <i>bits</i> no resultado são configurados como 1 se somente um dos <i>bits</i> correspondentes nos dois operandos for 1.
<<	deslocamento para a esquerda	Desloca os <i>bits</i> do primeiro operando para a esquerda pelo número de <i>bits</i> especificado pelo segundo operando; preenche a partir da direita com <i>bits</i> 0.
>>	deslocamento para a direita com extensão de sinal	Desloca os <i>bits</i> do primeiro operando para a direita pelo número de <i>bits</i> especificado pelo segundo operando. Se o primeiro operando for negativo, preenche com 1s a partir da esquerda; caso contrário, preenche com 0s a partir da esquerda.
>>>	deslocamento para a direita com extensão de zeros	Desloca os <i>bits</i> do primeiro operando para a direita pelo número de <i>bits</i> especificado pelo segundo operando; são inseridos 0s a partir da esquerda.
~	complemento de um	Todos os <i>bits</i> 0 são configurados como 1 e todos os <i>bits</i> 1 são configurados como 0.

Fig. 20.5 Operadores sobre *bits*.

Quando se utilizam operadores sobre *bits*, é útil exibir valores em sua representação binária para ilustrar os efeitos desses operadores. O aplicativo da Fig. 20.6 permite que o usuário digite um inteiro em um **JTextField** e pressionar *Enter*. O método **actionPerformed** lê o **String** do **JTextField**, converte-o em um inteiro e invoca o método **getBits** (linhas 55 a 80) para obter a representação **String** do inteiro em *bits*. O resultado é exibido na saída **JTextField**. O inteiro é exibido em sua representação binária em grupos de oito *bits* cada. O método **getBits** utiliza o operador **AND** sobre *bits* para combinar a variável **value** com a variável **displayMask**. Freqüentemente, o operador **AND** sobre *bits* é utilizado com um operando denominado *máscara* – um valor inteiro com *bits* específicos configurados como 1. As máscaras são utilizadas para ocultar alguns *bits* em um valor enquanto se seleciona outros *bits*. Em **getBits**, a variável máscara **displayMask** recebe o valor **1 << 31** ou

```
10000000 00000000 00000000 00000000
```

O operador de deslocamento para a esquerda desloca o valor 1 a partir do *bit* menos significativo (mais à direita) para o *bit* mais significativo (mais à esquerda) em **displayMask** e preenche com *bits* 0 a partir da direita.

```

1 // Fig. 20.6: PrintBits.java
2 // Imprimindo um inteiro sem sinal em binário
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class PrintBits extends JFrame {
12     private JTextField outputField;
13
14     // configura a GUI
15     public PrintBits()
16     {
17         super( "Printing bit representations for numbers" );
18
19         Container container = getContentPane();
20         container.setLayout( new FlowLayout() );
21
22         container.add( new JLabel( "Enter an integer" ) );
23
24         // campo de texto para ler valor do usuário
25         JTextField inputField = new JTextField( 10 );
26
27         inputField.addActionListener(
28
29             new ActionListener() {
30
31                 // lê inteiro e obtém a representação bit a bit
32                 public void actionPerformed( ActionEvent event )
33                 {
34                     int value = Integer.parseInt(
35                         event.getActionCommand() );
36                     outputField.setText( getBits( value ) );
37                 }
38             }
39         );
40
41         container.add( inputField );
42
43         container.add( new JLabel( "The integer in bits is" ) );
44
45         // campo de texto para exibir inteiro no formato binário
46         outputField = new JTextField( 33 );
47         outputField.setEditable( false );
48         container.add( outputField );
49
50         setSize( 720, 70 );
51         setVisible( true );
52     }
53
54     // exibe a representação em binário do valor int especificado
55     private String getBits( int value )
56     {
57         // cria valor int com 1 no bit mais significativo e 0s nos outros
58         int displayMask = 1 << 31;
59
60         // buffer para construir a saída

```

Fig. 20.6 Exibindo a representação dos *bits* de um inteiro (parte 1 de 2).

```

61     StringBuffer buffer = new StringBuffer( 35 );
62
63     // para cada bit, acrescenta 0 ou 1 ao buffer
64     for ( int bit = 1; bit <= 32; bit++ ) {
65
66         // usa displayMask para isolar o bit e determinar
67         // se o bit tem valor 0 ou 1
68         buffer.append(
69             ( value & displayMask ) == 0 ? '0' : '1' );
70
71         // desloca valor uma posição para a esquerda
72         value <<= 1;
73
74         // acrescenta um espaço ao buffer a cada 8 bits
75         if ( bit % 8 == 0 )
76             buffer.append( ' ' );
77     }
78
79     return buffer.toString();
80 }
81
82 // executa o aplicativo
83 public static void main( String args[] )
84 {
85     PrintBits application = new PrintBits();
86
87     application.setDefaultCloseOperation(
88         JFrame.EXIT_ON_CLOSE );
89 }
90
91 } // fim da classe PrintBits

```

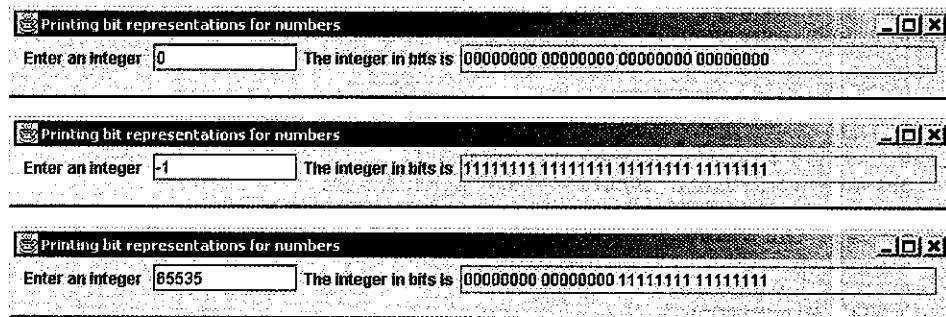


Fig. 20.6 Exibindo a representação dos *bits* de um inteiro (parte 2 de 2).

As linhas 68 e 69 determinam se um 1 ou um 0 deve ser acrescentado a um `StringBuffer` para o *bit* atual mais à esquerda da variável `value`. Suponha que `value` contenha `4000000000` (`11101110 01101011 00101000 00000000`). Quando `value` e `displayMask` são combinados com `&`, todos os *bits*, exceto o *bit* mais significativo (mais à esquerda) na variável `value`, são “mascarados” (ocultos), uma vez que qualquer *bit* que usa 1 com 0 produz 0. Se o *bit* mais à esquerda for 1, `value & displayMask` são avaliados como um valor não-zero e 1 é acrescentado; caso contrário, acrescenta-se 0. A variável `value` é então deslocada um *bit* para a esquerda pela expressão `value <<= 1` (isso é equivalente a `value = value << 1`). Esses passos são repetidos para cada *bit* na variável `value`. No final do método `getBits`, `StringBuffer` é convertido em um `String` na linha 79 e devolvido pelo método. A Fig. 20.7 resume os resultados da combinação de dois *bits* com o operador E sobre *bits* (`&`).



Erro comum de programação 20.1

Utilizar o operador E lógico (`&&`) em vez do operador E sobre bits (`&`) é um erro comum de programação.

O programa da Fig. 20.8 demonstra o uso do operador E sobre *bits*, o operador OU inclusivo sobre *bits*, o operador OU exclusivo sobre *bits* e o operador de complemento sobre *bits*. O programa utiliza o método `getBits` (linhas 163 a 188) para obter uma representação de `String` dos valores inteiros. O programa permite inserir valores em `JTextFields` (para os operadores binários, devem ser digitados dois valores) e pressionar o botão que representa a operação que você gostaria de testar. O programa exibe o resultado de cada operação na representação de inteiro e na representação binária.

A primeira janela de saída na Fig. 20.8 mostra os resultados da combinação do valor 65535 e do valor 1 com o operador E sobre *bits* (&). Todos os *bits*, exceto o menos significativo no valor 65535, são “mascarados” (ocultos) inserindo-se o “E” com o valor 1.

<i>Bit 1</i>	<i>Bit 2</i>	<i>Bit 1 & Bit 2</i>
0	0	0
1	0	0
0	1	0
1	1	1

Fig. 20.7 Resultados da combinação de dois *bits* com o operador E sobre *bits* (&).

```

1 // Fig. 20.8: MiscBitOps.java
2 // Usando os operadores E sobre bits, OU inclusivo sobre bits,
3 // OU exclusivo sobre bits e complemento sobre bits.
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.event.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class MiscBitOps extends JFrame {
13     private JTextField input1Field, input2Field,
14         bits1Field, bits2Field, bits3Field, resultField;
15     private int value1, value2;
16
17     // configura a GUI
18     public MiscBitOps()
19     {
20         super( "Bitwise operators" );
21
22         JPanel inputPanel = new JPanel();
23         inputPanel.setLayout( new GridLayout( 4, 2 ) );
24
25         inputPanel.add( new JLabel( "Enter 2 ints" ) );
26         inputPanel.add( new JLabel( "" ) );
27
28         inputPanel.add( new JLabel( "Value 1" ) );
29         input1Field = new JTextField( 8 );
30         inputPanel.add( input1Field );
31
32         inputPanel.add( new JLabel( "Value 2" ) );

```

Fig. 20.8 Demonstrando os operadores E sobre *bits*, OU inclusivo sobre *bits*, OU exclusivo sobre *bits* e o complemento sobre *bits* (parte 1 de 5).

```

33     input2Field = new JTextField( 8 );
34     inputPanel.add( input2Field );
35
36     inputPanel.add( new JLabel( "Result" ) );
37     resultField = new JTextField( 8 );
38     resultField.setEditable( false );
39     inputPanel.add( resultField );
40
41     JPanel bitsPanel = new JPanel();
42     bitsPanel.setLayout( new GridLayout( 4, 1 ) );
43     bitsPanel.add( new JLabel( "Bit representations" ) );
44
45     bits1Field = new JTextField( 33 );
46     bits1Field.setEditable( false );
47     bitsPanel.add( bits1Field );
48
49     bits2Field = new JTextField( 33 );
50     bits2Field.setEditable( false );
51     bitsPanel.add( bits2Field );
52
53     bits3Field = new JTextField( 33 );
54     bits3Field.setEditable( false );
55     bitsPanel.add( bits3Field );
56
57     JPanel buttonPanel = new JPanel();
58
59     // botão para executar E sobre bits
60     JButton andButton = new JButton( "AND" );
61
62     andButton.addActionListener(
63
64         new ActionListener() {
65
66             // executa E sobre bits e exibe resultado
67             public void actionPerformed( ActionEvent event )
68             {
69                 setFields();
70                 resultField.setText(
71                     Integer.toString( value1 & value2 ) );
72                 bits3Field.setText( getBits( value1 & value2 ) );
73             }
74         }
75     );
76
77     buttonPanel.add( andButton );
78
79     // botão para executar OU inclusivo sobre bits
80     JButton inclusiveOrButton = new JButton( "Inclusive OR" );
81
82     inclusiveOrButton.addActionListener(
83
84         new ActionListener() {
85
86             // executa OU inclusivo sobre bits e exibe resultado
87             public void actionPerformed( ActionEvent event )
88             {
89                 setFields();
90                 resultField.setText(
91                     Integer.toString( value1 | value2 ) );

```

Fig. 20.8 Demonstrando os operadores E sobre *bits*, OU inclusivo sobre *bits*, OU exclusivo sobre *bits* e o complemento sobre *bits* (parte 2 de 5).

```

92         bits3Field.setText( getBits( value1 | value2 ) );
93     }
94   };
95 };
96
97 buttonPanel.add( inclusiveOrButton );
98
99 // botão para fazer OU exclusivo sobre bits
100 JButton exclusiveOrButton = new JButton( "Exclusive OR" );
101
102 exclusiveOrButton.addActionListener(
103
104     new ActionListener() {
105
106         // executa OU exclusivo sobre bits e exibe resultado
107         public void actionPerformed( ActionEvent event )
108     {
109         setFields();
110         resultField.setText(
111             Integer.toString( value1 ^ value2 ) );
112         bits3Field.setText( getBits( value1 ^ value2 ) );
113     }
114 }
115 );
116
117 buttonPanel.add( exclusiveOrButton );
118
119 // botão para executar o complemento sobre bits
120 JButton complementButton = new JButton( "Complement" );
121
122 complementButton.addActionListener(
123
124     new ActionListener() {
125
126         // executa o complemento sobre bits e exibe resultado
127         public void actionPerformed( ActionEvent event )
128     {
129         input2Field.setText( "" );
130         bits2Field.setText( "" );
131
132         int value = Integer.parseInt( input1Field.getText() );
133
134         resultField.setText( Integer.toString( ~value ) );
135         bits1Field.setText( getBits( value ) );
136         bits3Field.setText( getBits( ~value ) );
137     }
138 }
139 );
140
141 buttonPanel.add( complementButton );
142
143 Container container = getContentPane();
144 container.add( inputPanel, BorderLayout.WEST );
145 container.add( bitsPanel, BorderLayout.EAST );
146 container.add( buttonPanel, BorderLayout.SOUTH );
147
148 setSize( 600, 150 );
149 setVisible( true );
150 }

```

Fig. 20.8 Demonstrando os operadores E sobre *bits*, OU inclusivo sobre *bits*, OU exclusivo sobre *bits* e o complemento sobre *bits* (parte 3 de 5).

```

151
152     // exibe números e sua representação binária
153     private void setFields()
154     {
155         value1 = Integer.parseInt( input1Field.getText() );
156         value2 = Integer.parseInt( input2Field.getText() );
157
158         bits1Field.setText( getBits( value1 ) );
159         bits2Field.setText( getBits( value2 ) );
160     }
161
162     // exibe a representação binária do valor int especificado
163     private String getBits( int value )
164     {
165         // cria valor int com 1 no bit mais significativo e 0s nos demais
166         int displayMask = 1 << 31;
167
168         // buffer para construir a saída
169         StringBuffer buffer = new StringBuffer( 35 );
170
171         // para cada bit, acrescenta 0 ou 1 ao buffer
172         for ( int bit = 1; bit <= 32; bit++ ) {
173
174             // usa displayMask para isolar bit e determinar
175             // se o bit tem valor de 0 ou 1
176             buffer.append(
177                 ( value & displayMask ) == 0 ? '0' : '1' );
178
179             // desloca o valor uma posição para a esquerda
180             value <=> 1;
181
182             // coloca um espaço no buffer a cada oito bits
183             if ( bit % 8 == 0 )
184                 buffer.append( ' ' );
185         }
186
187         return buffer.toString();
188     }
189
190     // executa o aplicativo
191     public static void main( String args[] )
192     {
193         MiscBitOps application = new MiscBitOps();
194
195         application.setDefaultCloseOperation(
196             JFrame.EXIT_ON_CLOSE );
197     }
198
199 } // fim da classe MiscBitOps

```

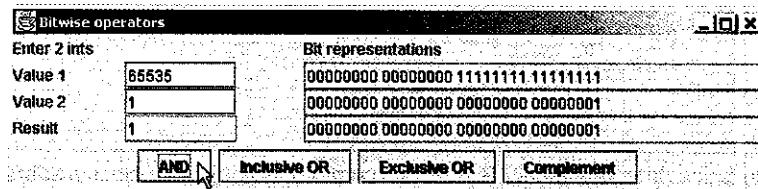


Fig. 20.8 Demonstrando os operadores E sobre *bits*, OU inclusivo sobre *bits*, OU exclusivo sobre *bits* e o complemento sobre *bits* (parte 4 de 5).

The figure consists of three vertically stacked windows from a software titled 'Bitwise operators'. Each window has fields for 'Value 1' and 'Value 2' and a 'Result' field. It also features four buttons: 'AND', 'Inclusive OR' (highlighted in yellow), 'Exclusive OR', and 'Complement'. The first window shows Value 1 as 15, Value 2 as 241, and Result as 255. The second window shows Value 1 as 139, Value 2 as 199, and Result as 76. The third window shows Value 1 as 21845, Value 2 as -21846, and Result as -21846. Each window also displays 'Bit representations' for each value, showing their binary equivalents.

Fig. 20.8 Demonstrando os operadores E sobre *bits*, OU inclusivo sobre *bits*, OU exclusivo sobre *bits* e o complemento sobre *bits* (parte 5 de 5).

O operador OU inclusivo sobre *bits* configura os *bits* específicos como 1 em um operando. A segunda janela de saída para a Fig. 20.8 mostra os resultados da combinação do valor 15 e do valor 241 com o operador OU inclusivo sobre *bits* – o resultado é 255. A Fig. 20.9 resume os resultados da combinação de dois *bits* com o operador OU inclusivo sobre *bits*.



Erro comum de programação 20.2

Utilizar o operador lógico (`||`) em vez do operador OU sobre bits (`|`) é um erro comum de programação.

O operador OU exclusivo sobre *bits* (^) configura cada *bit* no resultado como 1 se *somente* um dos *bits* correspondentes em seus dois operandos for 1. A terceira saída da Fig. 20.8 mostra os resultados da combinação do valor 139 e do valor 199 com o operador OU exclusivo – o resultado é 76. A Fig. 20.10 resume os resultados da combinação de dois *bits* com o operador OU exclusivo sobre *bits*.

Bit 1	Bit 2	Bit 1 Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 20.9 Resultados da combinação de dois *bits* com o operador OU inclusivo sobre *bits* (`|`).

O operador de complemento sobre *bits* (`-`) configura todo os *bits* 1 em seu operando como 0 no resultado e configura todos os *bits* 0 como 1 no resultado – processo que também se chama “obter o *complemento de um* do valor”. A quarta janela de saída na Fig. 20.8 mostra os resultados de se obter o complemento de um do valor 21845. O resultado é -21846.

<i>Bit 1</i>	<i>Bit 2</i>	<i>Bit 1 ^ Bit 2</i>
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 20.10 Resultados da combinação de dois *bits* com o operador OU exclusivo sobre *bits* (^).

O programa da Fig. 20.11 demonstra o *operador de deslocamento para a esquerda* (<<), o *operador de deslocamento para direita com extensão de sinal* (>>) e o *operador de deslocamento para direita com extensão de zero* (>>>). O método `getBits` (linhas 113 a 138) obtém um `String` que contém a representação binária dos valores inteiros. O programa permite que o usuário digite um inteiro em um `JTextField` e pressione `Enter` para exibir a representação binária do inteiro em um segundo `JTextField`. O usuário pode pressionar o botão que representa uma operação de deslocamento para fazer um deslocamento de 1 *bit* e visualizar os resultados do deslocamento representados como um inteiro e como binário.

```

1 // Fig. 20.11: BitShift.java
2 // Usando os operadores de deslocamento sobre bits.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Pacotes de extensão de Java
9 import javax.swing.*;
10
11 public class BitShift extends JFrame {
12     private JTextField bitsField;
13     private JTextField valueField;
14
15     // configura a GUI
16     public BitShift()
17     {
18         super( "Shifting bits" );
19
20         Container container = getContentPane();
21         container.setLayout( new FlowLayout() );
22
23         container.add( new JLabel( "Integer to shift" ) );
24
25         // campo de texto para o usuário digitar o inteiro
26         valueField = new JTextField( 12 );
27         container.add( valueField );
28
29         valueField.addActionListener(
30             new ActionListener() {
31
32                 // lê valor e exibe sua representação binária
33                 public void actionPerformed( ActionEvent event )
34                 {
35

```

Fig. 20.11 Demonstrando os operadores de deslocamento sobre *bits* (parte 1 de 4).

```

36         int value = Integer.parseInt( valueField.getText() );
37         bitsField.setText( getBits( value ) );
38     }
39 }
40 );
41
42 // campo de texto para exibir a representação binária de um inteiro
43 bitsField = new JTextField( 33 );
44 bitsField.setEditable( false );
45 container.add( bitsField );
46
47 // botão para deslocar bits para a esquerda em uma posição
48 JButton leftButton = new JButton( "<<" );
49
50 leftButton.addActionListener(
51
52     new ActionListener() {
53
54         // desloca uma posição para a esquerda e exibe novo valor
55         public void actionPerformed( ActionEvent event )
56     {
57             int value = Integer.parseInt( valueField.getText() );
58             value <<= 1;
59             valueField.setText( Integer.toString( value ) );
60             bitsField.setText( getBits( value ) );
61         }
62     }
63 );
64
65 container.add( leftButton );
66
67 // botão para deslocar valor para a direita com extensão do sinal
68 JButton rightSignButton = new JButton( ">>" );
69
70 rightSignButton.addActionListener(
71
72     new ActionListener() {
73
74         // desloca para a direita uma posição e exibe novo valor
75         public void actionPerformed( ActionEvent event )
76     {
77             int value = Integer.parseInt( valueField.getText() );
78             value >>= 1;
79             valueField.setText( Integer.toString( value ) );
80             bitsField.setText( getBits( value ) );
81         }
82     }
83 );
84
85 container.add( rightSignButton );
86
87 // botão para deslocar valor para a direita com extensão de zeros
88 JButton rightZeroButton = new JButton( ">>>" );
89
90 rightZeroButton.addActionListener(
91
92     new ActionListener() {
93
94         // desloca para a direita uma posição e exibe novo valor
95         public void actionPerformed( ActionEvent event )

```

Fig. 20.11 Demonstrando os operadores de deslocamento sobre bits (parte 2 de 4).

```

96         {
97             int value = Integer.parseInt( valueField.getText() );
98             value >>>= 1;
99             valueField.setText( Integer.toString( value ) );
100
101            bitsField.setText( getBits( value ) );
102        }
103    }
104
105    container.add( rightZeroButton );
106
107    setSize( 400, 120 );
108    setVisible( true );
109 }
110
111 // exibe representação binária do valor int especificado
112 private String getBits( int value )
113 {
114     // cria valor int com 1 no bit mais significativo e 0s nos demais
115     int displayMask = 1 << 31;
116
117     // buffer para construir a saída
118     StringBuffer buffer = new StringBuffer( 35 );
119
120     // para cada bit acrescenta 0 ou 1 ao buffer
121     for ( int bit = 1; bit <= 32; bit++ ) {
122
123         // usa displayMask para isolar bit e determinar
124         // se o bit tem valor 0 ou 1
125         buffer.append(
126             ( value & displayMask ) == 0 ? '0' : '1' );
127
128         // desloca valor uma posição para a esquerda
129         value <<= 1;
130
131         // coloca um espaço no buffer a cada oito bits
132         if ( bit % 8 == 0 )
133             buffer.append( ' ' );
134     }
135
136     return buffer.toString();
137 }
138
139
140 // executa o aplicativo
141 public static void main( String args[] )
142 {
143     BitShift application = new BitShift();
144
145     application.setDefaultCloseOperation(
146         JFrame.EXIT_ON_CLOSE );
147 }
148
149 } // fim da classe BitShift

```

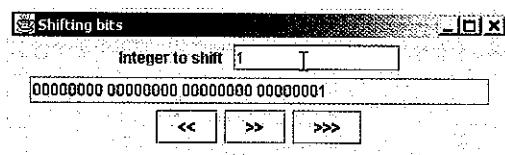


Fig. 20.11 Demonstrando os operadores de deslocamento sobre bits (parte 3 de 4).

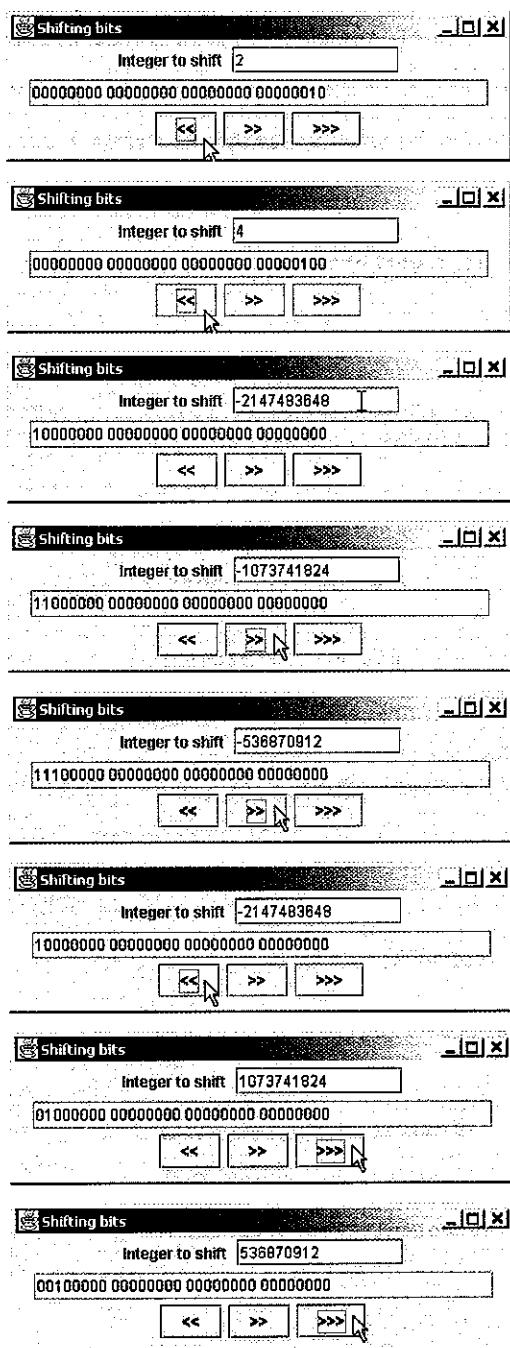


Fig. 20.11 Demonstrando os operadores de deslocamento sobre bits (parte 4 de 4).

O operador de deslocamento para a esquerda (`<<`) desloca os *bits* de seu operando esquerdo para a esquerda pelo número de *bits* especificado em seu operando direito (o que é feito na linha 58 do programa). Os *bits* vagos à direita são substituídos por 0s; os 1s deslocados à esquerda se perdem. As quatro primeiras janelas de saída da Fig. 20.11 demonstram o operador de deslocamento para a esquerda. Começando com o valor 1, o botão de deslocamento para a esquerda foi pressionado duas vezes, resultando nos valores 2 e 4, respectivamente. A quarta janela de saída mostra o resultado de `value1` sendo deslocado 31 vezes. Observe que o resultado é um valor negativo. Isso ocorre porque se utiliza um 1 no *bit* mais significativo para indicar um valor negativo em um inteiro.

O operador de deslocamento para a direita com extensão de sinal (`>>`) desloca os *bits* de seu operando esquerdo para a direita pelo número de *bits* especificado em seu operando direito (o que é feito na linha 78 do programa). Realizar um deslocamento para a direita faz com que os *bits* vagos à esquerda sejam substituídos por 0s, se o número for positivo, e por 1s, se o número for negativo. Quaisquer 1s deslocados para a direita se perdem. A quinta e a sexta janelas de saída mostram os resultados de se deslocar para a direita (com extensão de sinal) o valor da quarta janela de saída duas vezes.

O operador de deslocamento para a direita com extensão de zero (`>>>`) desloca os *bits* de seu operando esquerdo para a direita pelo número de *bits* especificado em seu operando direito (o que é feito na linha 98 do programa). Realizar um deslocamento para a direita faz com que os *bits* vagos à esquerda sejam substituídos por 0s. Quaisquer 1s deslocados para a direita se perdem. A oitava e a nona janela de saída mostram os resultados de se deslocar para a direita (com extensão zero) o valor da sétima janela de saída duas vezes.

Cada operador sobre *bits* (exceto o operador de complemento sobre *bits*) tem um operador de atribuição correspondente. Esses *operadores de atribuição sobre bits* são mostrados na Fig. 20.12.

Operadores de atribuição sobre bits

<code>&=</code>	Operador de atribuição E sobre bits.
<code> =</code>	Operador de atribuição OU inclusivo sobre bits.
<code>^ =</code>	Operador de atribuição OU exclusivo sobre bits.
<code><<=</code>	Operador de atribuição de deslocamento para a esquerda.
<code>>>=</code>	Operador de atribuição de deslocamento para a direita com extensão de sinal.
<code>>>>=</code>	Operador de atribuição de deslocamento para a direita com extensão de zero.

Fig. 20.12 Os operadores de atribuição sobre *bits*.

20.9 A classe BitSet

A classe `BitSet` torna fácil criar e manipular *conjuntos de bits*. Os conjuntos de *bits* são úteis para representar um conjunto de indicadores `boolean`. `BitSets` são redimensionáveis dinamicamente. Conforme necessário, mais *bits* podem ser adicionados e um objeto `BitSet` crescerá para acomodar os *bits* adicionais. A instrução

```
BitSet b = new BitSet();
```

cria um `BitSet` que está inicialmente vazio. O programa também pode especificar o tamanho de um `BitSet` com a instrução

```
BitSet b = new BitSet( size );
```

que cria um `BitSet` com *bits* `size`.

A instrução

```
b.set( bitNumber );
```

configura o bit `bitNumber` do `BitSet b` como “ligado”. Isso torna o valor subjacente daquele `bit` igual a 1. Observe que os números dos `bits` começam em zero, como em `Vector`. A instrução

```
b.clear( bitNumber );
```

configura o `bit bitNumber` do `BitSet b` como “desligado”. Isso torna o valor subjacente daquele `bit` igual a 0. A instrução

```
b.get( bitNumber );
```

obtém o valor do `bit bitNumber`. O resultado é devolvido como `true`, se o `bit` estiver ligado, ou `false`, se o `bit` estiver desligado.

A instrução

```
b.and( b1 );
```

realiza um E lógico `bit` por `bit` entre os `BitSets b e b1`. O resultado é armazenado em `b`. O OU lógico sobre `bits` e o XOU lógico sobre `bits` são realizados pelas instruções

```
b.or( b1 );
b.xor( b2 );
```

A expressão

```
b.size()
```

devolve o tamanho do `BitSet b`. A expressão

```
b.equals( b1 )
```

```

1 // Fig. 20.13: BitSetTest.java
2 // Usando um BitSet para demonstrar o Crivo de Eratóstenes.
3
4 // Pacotes do núcleo de Java
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.util.*;
8
9 // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class BitSetTest extends JFrame {
13     private BitSet sieve;
14     private JLabel statusLabel;
15     private JTextField inputField;
16
17     // configura a GUI
18     public BitSetTest()
19     {
20         super( "BitSets" );
21
22         sieve = new BitSet( 1024 );
23
24         Container container = getContentPane();
25
26         statusLabel = new JLabel( "" );
27         container.add( statusLabel, BorderLayout.SOUTH );
28
29         JPanel inputPanel = new JPanel();
30
31         inputPanel.add( new JLabel(
32             "Enter a value from 2 to 1023" ) );
33
34         // campo de texto para o usuário digitar um valor de 2 a 1023

```

Fig. 20.13 Demonstrando o Crivo de Eratóstenes com um `BitSet` (parte 1 de 3).

```
35     inputField = new JTextField( 10 );
36
37     inputField.addActionListener(
38
39         new ActionListener() {
40
41             // determina se o valor é um número primo
42             public void actionPerformed( ActionEvent event ) {
43                 int value = Integer.parseInt( inputField.getText() );
44
45                 if ( sieve.get( value ) )
46                     statusLabel.setText(
47                         value + " is a prime number" );
48
49                 else
50                     statusLabel.setText( value +
51                         " is not a prime number" );
52             }
53         }
54     );
55
56
57     inputPanel.add( inputField );
58     container.add( inputPanel, BorderLayout.NORTH );
59
60     JTextArea primesArea = new JTextArea();
61
62     container.add( new JScrollPane( primesArea ),
63                     BorderLayout.CENTER );
64
65     // configura todos os bits, de 1 a 1023
66     int size = sieve.size();
67
68     for ( int i = 2; i < size; i++ )
69         sieve.set( i );
70
71     // executa o Crivo de Eratóstenes
72     int finalBit = ( int ) Math.sqrt( sieve.size() );
73
74     for ( int i = 2; i < finalBit; i++ )
75
76         if ( sieve.get( i ) )
77
78             for ( int j = 2 * i; j < size; j += i )
79                 sieve.clear( j );
80
81     // exibe os números primos de 2 a 1023
82     int counter = 0;
83
84     for ( int i = 2; i < size; i++ )
85
86         if ( sieve.get( i ) ) {
87             primesArea.append( String.valueOf( i ) );
88             primesArea.append( ++counter % 7 == 0 ? "\n" : "\t" );
89         }
90
91     setSize( 600, 450 );
92     setVisible( true );
93 }
94 }
```

Fig. 20.13 Demonstrando o Crivo de Eratóstenes com um BitSet (parte 2 de 3).

```

95 // executa o aplicativo
96 public static void main( String args[] )
97 {
98     BitSetTest application = new BitSetTest();
99
100    application.setDefaultCloseOperation(
101        JFrame.EXIT_ON_CLOSE );
102 }
103
104 } // fim da classe BitSetTest

```

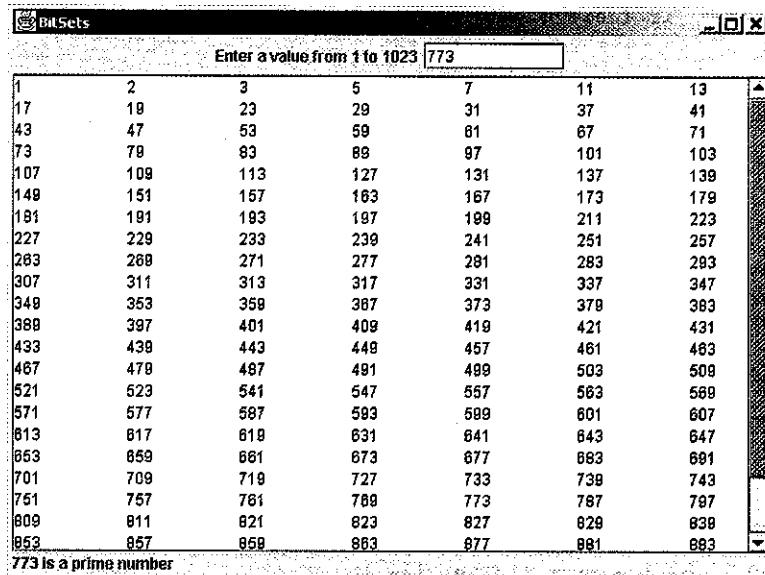


Fig. 20.13 Demonstrando o Crivo de Eratóstenes com um **BitSet** (parte 3 de 3).

verifica se os dois **BitSets** são iguais. A expressão

b.toString()

cria uma representação do conteúdo do **BitSet b** como um **String**. Isso é útil para depuração.

A Fig. 20.13 examina novamente o Crivo de Eratóstenes para localizar números primos, exercício que discutimos no Exercício 7.27. Este exemplo usa um **BitSet** em vez de um *array* para implementar o algoritmo. O programa exibe todos os números primos de 2 a 1023 em uma **JTextArea** e fornece um **JTextField** em que o usuário pode digitar qualquer número de 2 a 1023 para determinar se esse número é primo (caso em que uma mensagem é exibida em um **JLabel**).

A linha 22 cria um **BitSet** de 1024 bits. Ignoramos o bit com índice 0 nesse programa. As linhas 68 e 69 configuram todos os bits no **BitSet** como ligados, com o método **set** de **BitSet**. As linhas 72 a 79 determinam todos os números primos de 2 a 1023. O inteiro **finalBit** especifica quando o algoritmo está completo. O algoritmo básico é que um número é primo se ele não tiver nenhum divisor diferente de 1 e dele próprio. Começando com o número 2, uma vez que sabemos que um número é primo, podemos eliminar todos os múltiplos desse número. O número 2 só é divisível por 1 e por ele próprio, então é primo. Portanto, podemos eliminar 4, 6, 8 e assim por diante.

te. A eliminação de um valor consiste em configurar seu *bit* como desligado com o método `clear` de `BitSet`. O número 3 é divisível por 1 e por si próprio. Portanto, podemos eliminar todos os múltiplos de 3 (tenha em mente de que todos os números pares já foram eliminados). Após a lista dos primos ser exibido, o usuário pode digitar um valor entre 2 e 1023 no campo de texto e pressionar *Enter* para determinar se o número é primo. O método `actionPerformed` (linhas 42 a 53) utiliza o método `get` (linha 46) de `BitSet` para determinar se o *bit* para o número que o usuário digitou está ligado. Se estiver, as linhas 47 e 48 exibem uma mensagem indicando que o número é primo. Caso contrário, as linhas 51 e 52 exibem uma mensagem que indica que o número não é primo.

Resumo

- A classe `Vector` gerencia *arrays* dinamicamente redimensionáveis. A qualquer momento, o `Vector` contém um certo número de elementos que é menor que ou igual à sua *capacidade*. A capacidade é o espaço que foi reservado para o *array*.
- Se um `Vector` precisa crescer, ele cresce por um incremento que você especifica ou por um *default* assumido pelo sistema. Se você não especificar um incremento de capacidade, o sistema automaticamente dobrará o tamanho do `Vector` toda vez que for necessária capacidade adicional.
- `Vectors` armazenam referências a `Objects`. Para armazenar valores de tipos primitivos de dados em `Vectors`, você deve utilizar as classes empacotadoras de tipo (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` e `Character`) para criar objetos que contêm os valores de tipos primitivos de dados.
- A classe `Vector` fornece três construtores. O construtor sem argumentos cria um `Vector` vazio. O construtor que recebe um argumento cria um `Vector` com uma capacidade inicial especificada pelo argumento. O construtor que recebe dois argumentos cria um `Vector` com uma *capacidade inicial* especificada pelo primeiro argumento e um *incremento de capacidade* especificado pelo segundo argumento.
- O método `addElement` de `Vector` adiciona seu argumento ao final do `Vector`. O método `insertElementAt` insere um elemento na posição especificada. O método `setElementAt` configura o elemento em uma posição específica.
- O método `removeElement` de `Vector` remove a primeira ocorrência de seu argumento. O método `removeAllElements` remove todos os elementos do `Vector`. O método `removeElementAt` remove o elemento no índice especificado.
- O método `firstElement` de `Vector` devolve uma referência para o primeiro elemento. O método `lastElement` devolve uma referência para o último elemento.
- O método `isEmpty` de `Vector` determina se o `Vector` está vazio. O método `contains` determina se o `Vector` contém a `searchKey` especificada como argumento.
- O método `indexOf` obtém o índice da primeira localização de seu argumento. O método devolve `-1` se o argumento não for localizado no `Vector`.
- O método `trimToSize` de `Vector` reduz a capacidade do `Vector` ao tamanho do `Vector`. Os métodos `size` e `capacity` determinam o número de elementos atualmente no `Vector` e o número de elementos que pode ser armazenado no `Vector` sem alocar mais memória, respectivamente.
- O método `elements` de `Vector` devolve uma referência para uma `Enumeration` que contém os elementos do `Vector`.
- O método `hasMoreElements` de `Enumeration` determina se há mais elementos. O método `nextElement` devolve uma referência para o próximo elemento.
- A classe `Stack` estende a classe `Vector`. O método `push` de `Stack` adiciona seu argumento à parte superior da pilha. O método `pop` remove o elemento superior da pilha. O método `peek` devolve uma referência `Object` para o elemento superior da pilha sem remover o elemento. O método `empty` de `Stack` determina se a pilha está vazia.
- Um `Dictionary` transforma chaves em valores.
- `Hashing` é um esquema de alta velocidade para converter chaves em subscritos únicos de *array* para armazenamento e recuperação de informações. O fator de carga é a relação entre o número de células ocupadas em uma tabela de *hash* e o tamanho da tabela. Quanto mais a proporção se aproximar de 1, maior será a chance de colisões.
- O construtor `Hashtable` sem argumentos cria uma `Hashtable` com uma capacidade *default* de 101 elementos e um fator de carga *default* de 0,75. O construtor `Hashtable` que recebe um argumento especifica a capacidade inicial e o construtor que recebe dois argumentos especifica a capacidade inicial e o fator de carga, respectivamente.
- O método `put` de `Hashtable` adiciona uma *chave* e um *valor* a uma `Hashtable`. O método `get` localiza o valor associado com a chave especificada. O método `remove` exclui o valor associado com a chave especificada. O método `isEmpty` determina se a tabela está vazia.

- O método `containsKey` de `Hashtable` determina se a chave especificada como argumento está na `Hashtable` (isto é, um valor está associado com essa chave). O método `contains` determina se o `Object` especificado como seu argumento está na `Hashtable`. O método `clear` esvazia a `Hashtable`. O método `elements` obtém uma `Enumeration` dos valores. O método `keys` obtém uma `Enumeration` das chaves.
- O objeto `Properties` é um objeto `Hashtable` persistente. A classe `Properties` estende `Hashtable`. As chaves e valores em um objeto `Properties` devem ser `Strings`.
- O construtor sem argumentos `Properties` cria uma tabela vazia `Properties` sem propriedades *default*. Há também um construtor sobrecarregado que recebe uma referência para um objeto `Properties default` que contém valores de propriedade *default*.
- O método `getProperty` de `Properties` localiza o valor da chave especificada como argumento. O método `store` salva o conteúdo do objeto `Properties` no objeto `OutputStream` especificado como primeiro argumento. O método `load` restaura o conteúdo do objeto `Properties` a partir do objeto `InputStream` especificado como argumento. O método `propertyNames` obtém uma `Enumeration` dos nomes de propriedades.
- Java fornece muitos recursos para a geração de números aleatórios na classe `Random`. O construtor sem argumentos da classe `Random` utiliza a hora para alimentar seu gerador de números aleatórios diferentemente toda vez que ele é chamado. Para criar um gerador de números pseudo-aleatórios com repetição, utilize o construtor `Random` que recebe um argumento de semente.
- O método `setSeed` de `Random` configura a semente. Os métodos `nextInt` e `nextLong` geram inteiros aleatórios distribuídos uniformemente. Os métodos `nextFloat` e `nextDouble` geram valores uniformemente distribuídos no intervalo $0,0 \leq x < 1,0$.
- O operador E sobre *bits* (&) configura cada *bit* no resultado como 1 se o *bit* correspondente em ambos os operandos for 1.
- O operador OU inclusivo sobre *bits* (|) configura cada *bit* no resultado como 1 se o *bit* correspondente em qualquer operando (ou em ambos) for 1.
- O operador OU exclusivo sobre *bits* (^) configura cada *bit* no resultado como 1 se o *bit* correspondente em somente um operando for 1.
- O operador de deslocamento para a esquerda (<<) desloca os *bits* de seu operando esquerdo para a esquerda pelo número de *bits* especificado em seu operando direito.
- O operador de deslocamento para a direita com extensão de sinal (>>) desloca os *bits* em seu operando esquerdo para a direita pelo número de *bits* especificado em seu operando direito – se o operando esquerdo for negativo, são inseridos 1s à esquerda; caso contrário, são inseridos 0s à esquerda.
- O operador de deslocamento para a direita com extensão de zero (>>>) desloca os *bits* em seu operando esquerdo para a direita pelo número de *bits* especificado em seu operando direito – são inseridos 0s à esquerda.
- O operador complemento sobre *bits* (~) configura todos os *bits* 0 em seu operando como 1 no resultado e configura todos os *bits* 1 como 0 no resultado.
- Cada operador sobre *bits* (exceto o de complemento) tem um operador de atribuição correspondente.
- O construtor `BitSet` sem argumento cria um `BitSet` vazio. O construtor `BitSet` com um argumento cria um `BitSet` com o número de *bits* especificado por seu argumento.
- O método `set` de `BitSet` configura o *bit* especificado como “ligado”. O método `clear` configura o *bit* especificado como “desligado”. O método `get` devolve `true` se o *bit* estiver ligado ou `false` se estiver desligado.
- O método `and` de `BitSet` realiza um E lógico *bit* por *bit* entre os `BitSets`. O resultado é armazenado no `BitSet` que invocou o método. O OU lógico sobre *bits* ou o XOU lógico sobre *bits* é realizado pelos métodos `or` e `xor`.
- O método `size` de `BitSet` devolve o tamanho de um `BitSet`. O método `toString` converte um `BitSet` em um `String`.

Terminologia

aparar um Vector para seu tamanho exato
array dinamicamente redimensionável
ArrayList
OutOfBoundsException
capacidade de um Vector
capacidade inicial de um Vector
caracteres de espaço em branco
chave em um Dictionary
classe BitSet

classe Dictionary
classe EmptyStackException
classe Hashtable
classe NoSuchElementException
classe Properties
classe Random
classe Stack
classe Vector

colisão em hashing
 conjunto de bits
 elementos sucessivos enumerados
 fator de carga em hashing
 hashing
IllegalArgumentException
 incremento da capacidade de um **Vector**
interface Enumeration
 iterar pelos elementos de um contêiner
 método **addElement** da classe **Vector**
 método **and** da classe **BitSet**
 método **capacity** da classe **Vector**
 método **clear** da classe **BitSet**
 método **clear** da classe **Hashtable**
 método **clone** da classe **BitSet**
 método **contains** da classe **Vector**
 método **containsKey** da classe **Hashtable**
 método **elementAt** da classe **Vector**
 método **elements** da classe **Dictionary**
 método **elements** da classe **Vector**
 método **equals** da classe **Object**
 método **firstElement** da classe **Vector**
 método **get** da classe **BitSet**
 método **get** da classe **Dictionary**

método **getProperty** da classe **Properties**
 método **hashCode** da classe **Object**
 método **hasMoreElements** (**Enumeration**)
 método **indexOf** da classe **Vector**
 método **insertElementAt** da classe **Vector**
 método **isEmpty** da classe **Dictionary**
 método **isEmpty** da classe **Vector**
 método **keys** da classe **Dictionary**
 método **lastElement** da classe **Vector**
 método **list** da classe **Properties**
 método **load** da classe **Properties**
 método **nextDouble** da classe **Random**
 método **nextElement** de **Enumeration**
 método **nextFloat** da classe **Random**
 método **nextInt** da classe **Random**
 método **nextLong** da classe **Random**
 método **or** da classe **BitSet**
 método **peek** da classe **Stack**
 método **pop** da classe **Stack**
 método **propertyNames** de **Properties**
 método **push** da classe **Stack**
 método **put** da classe **Dictionary**
 método **remove** da classe **Dictionary**
 método **removeAllElements** de **Vector**

Exercícios de auto-revisão

- 20.1** Preencha as lacunas em cada uma das frases seguintes:
- A classe Java _____ fornece as capacidades de estruturas de dados do tipo *array* que podem redimensionar dinamicamente a si próprias.
 - Se você não especificar um incremento de capacidade, o sistema automaticamente irá _____ o tamanho do **Vector** toda vez que a capacidade adicional for necessária.
 - Se memória for um recurso escasso, utilize o método _____ da classe **Vector** para aparar um **Vector** para seu tamanho exato.
- 20.2** Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
- Os valores de tipos primitivos de dados podem ser diretamente armazenados em um **Vector**.
 - Com *hashing*, enquanto o fator de carga aumenta, diminui a chance de colisões.
- 20.3** Sob quais circunstâncias uma **EmptyStackException** é disparada?
- 20.4** Preencha as lacunas em cada uma das frases seguintes:
- Os *bits* no resultado de uma expressão que usa o operador _____ são configurados como 1 se os *bits* correspondentes em cada operando estão configurados como 1. Caso contrário, os *bits* são configurados como zero.
 - Os *bits* no resultado de uma expressão que usa o operador _____ são configurados como 1 se pelo menos um dos *bits* correspondentes em qualquer operando estiver configurado como 1. Caso contrário, os *bits* são configurados como zero.
 - Os *bits* no resultado de uma expressão que usa o operador _____ são configurados como 1 se somente um dos *bits* correspondentes em qualquer operando estiver configurado como 1. Caso contrário, os *bits* são configurados como zero.
 - O operador E sobre *bits* (**&**) é freqüentemente utilizado para _____ os *bits*, isto é, para selecionar certos *bits* de um *string* de *bits* enquanto coloca zero em outros.
 - Utiliza-se o operador _____ para deslocar os *bits* de um valor para a esquerda.
 - O _____ desloca os *bits* de um valor para a direita com extensão de sinal e o _____ desloca os *bits* de um valor para a direita com extensão zero.

Respostas dos exercícios de auto revisão

20.1 a) `Vector`. B) dobrar. c) `trimToSize`.

20.2 a) Falsa; `Vector` armazena somente `Objects`. Você deve utilizar as classes empacotadoras de tipos (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` e `Character`) do pacote `java.lang` para criar `Objects` que contêm valores de tipos primitivos de dados. b) Falsa; à medida que aumenta o fator de carga, há menos posições disponíveis em relação ao número total de posições, assim aumenta a chance de se selecionar uma posição ocupada com uma operação de *hashing* (isto é, uma colisão).

20.3 Quando um programa chama `pop` ou `peek` para um objeto `Stack` vazio, ocorre uma `EmptyStackException`.

20.4 a) E sobre *bits* (&). b) OU inclusivo sobre *bits* (|). c) OU exclusivo sobre *bits* (^). d) mascarar. e) operador de deslocamento para a esquerda (<<). f) operador de deslocamento para a direita com extensão de sinal (>>), operador de deslocamento para a direita com extensão zero (>>>).

Exercícios

20.5 Defina cada um dos seguintes termos no contexto de *hashing*:

- a) chave
- b) colisão
- c) transformação de *hashing*
- d) fator de carga
- e) compromisso espaço/tempo
- f) classe `Hashtable`
- g) capacidade de uma `Hashtable`

20.6 Explique brevemente a operação de cada um dos seguintes métodos da classe `Vector`:

- a) `addElement`
- b) `insertElementAt`
- c) `setElementAt`
- d) `removeElement`
- e) `removeAllElements`
- f) `removeElementAt`
- g) `firstElement`
- h) `lastElement`
- i) `isEmpty`
- j) `contains`
- k) `indexOf`
- l) `trimToSize`
- m) `size`
- n) `capacity`

20.7 Explique por que é uma operação relativamente lenta inserir elementos adicionais em um objeto `Vector` cujo tamanho atual é menor que sua capacidade é uma operação relativamente rápida e por que inserir elementos adicionais em um objeto `Vector` cujo tamanho atual é igual à capacidade.

20.8 No texto, afirmamos que o incremento *default* da capacidade de dobrar o tamanho de um `Vector` pode parecer desperdício de armazenamento, mas é realmente uma maneira eficiente de fazer os `Vectors` crescerem rapidamente até estarem “próximos do tamanho correto”. Explique essa afirmação. Explique as vantagens e desvantagens desse algoritmo de duplicação. O que um programa pode fazer quando ele determina que a duplicação está desperdiçando espaço?

20.9 Explique o uso da interface `Enumeration` com objetos da classe `Vector`.

20.10 Estendendo a classe `Vector`, os projetistas de Java foram capazes de criar a classe `Stack` rapidamente. Quais são os aspectos negativos dessa utilização de herança, particularmente para a classe `Stack`?

20.11 Explique brevemente a operação de cada um dos seguintes métodos da classe `Hashtable`:

- a) `put`
- b) `get`
- c) `remove`

- d) `isEmpty`
- e) `containsKey`
- f) `contains`
- g) `clear`
- h) `elements`
- i) `keys`

20.12 Explique como utilizar a classe `Random` para criar números pseudo-aleatórios com o tipo de repetição necessária para depuração.

20.13 Utilize uma `Hashtable` para criar uma classe reutilizável a fim de escolher uma das 13 cores predefinidas da classe `Color`. Os nomes das cores devem ser utilizados como chaves e os objetos `Color` predefinidos devem ser utilizados como valores. Coloque essa classe em um pacote que pode ser importado em qualquer programa Java. Utilize sua nova classe em um aplicativo que permite selecionar uma cor e desenhar uma forma nessa cor.

20.14 Modifique sua solução do Exercício 13.18 – o programa de pintura polimórfico – para armazenar cada forma que o usuário desenhar em um `Vector` de objetos `MyShape`. Para os objetivos desse exercício, crie sua própria subclasse `Vector` denominada `ShapeVector` que manipula só objetos `MyShape`. Forneça os seguintes recursos em seu programa:

- a) Permita que o usuário do programa remova qualquer número de formas de `Vector` clicando em um botão `Undo`.
- b) Permita que o usuário selecione qualquer forma na tela e a mova para uma nova posição. Isso exige a adição de um novo método para a hierarquia `MyShape`. A primeira linha do método deve ser

```
public boolean isInside()
```

Esse método deve ser sobreescrito para cada subclasse de `MyShape` para determinar se as coordenadas em que o usuário pressionou o botão do mouse estão dentro da forma.

- c) Permita que o usuário selecione qualquer forma na tela e altere a cor.
- d) Permita que o usuário selecione qualquer forma na tela, que pode ser preenchida ou não preenchida, e altere o estado de preenchimento.

20.15 O que significa quando declaramos que um objeto `Properties` é um objeto `Hashtable` “persistente”? Explique a operação de cada um dos seguintes métodos da classe `Properties`:

- a) `load`
- b) `store`
- c) `getProperty`
- d) `propertyNames`
- e) `list`

20.16 Por que você desejará utilizar objetos da classe `BitSet`? Explique a operação de cada um dos seguintes métodos da classe `BitSet`:

- a) `set`
- b) `clear`
- c) `get`
- d) `and`
- e) `or`
- f) `xor`
- g) `size`
- h) `equals`
- i) `clone`
- j) `toString`
- k) `hashCode`

20.17 Escreva um programa que desloca para a direita uma variável inteira por 4 *bits* com a extensão de sinal e depois desloca para a direita a mesma variável inteira por 4 *bits* com extensão de zero. O programa deve imprimir o inteiro em binário antes e depois de cada operação de deslocamento. Execute o programa uma vez com um inteiro positivo e execute novamente com um inteiro negativo.

20.18 Mostre como se pode deslocar um inteiro para a esquerda por 1 *bit* para simular a multiplicação por 2 e como se pode deslocar um inteiro para direita por 1 *bit* para simular a divisão por 2. Tenha cuidado em considerar questões relacionadas com o sinal de um inteiro.

20.19 Escreva um programa que inverte a ordem dos *bits* em um valor inteiro. O programa deve ler um valor do usuário e usar uma chamada ao método `reverseBits` para imprimir os *bits* em ordem inversa. Imprima o valor em binário antes e depois de os *bits* serem invertidos para confirmar que os *bits* foram corretamente invertidos. Você pode querer implementar uma solução recursiva e uma iterativa.

20.20 Modifique a solução do Exercício 19.10 para utilizar a classe `Stack`.

20.21 Modifique a solução do Exercício 19.12 para utilizar a classe `Stack`.

20.22 Modifique a solução do Exercício 19.13 para utilizar a classe `Stack`.

Coleções

Objetivos

- Entender o que são coleções.
- Entender os novos recursos para *arrays* de Java 2.
- Ser capaz de utilizar implementações da estrutura de coleções.
- Ser capaz de utilizar algoritmos da estrutura de coleções para manipular diversas coleções.
- Ser capaz de utilizar interfaces da estrutura de coleções para programar polimorficamente.
- Ser capaz de utilizar iteradores para percorrer os elementos de uma coleção.
- Entender empacotadores de sincronização e empacotadores de modificabilidade.

Acho que essa é a coleção mais extraordinária de talento e conhecimento humano já reunida na Casa Branca – com a possível exceção de quando Thomas Jefferson jantava sozinho.

John F. Kennedy

As formas que um contêiner brilhante pode conter!

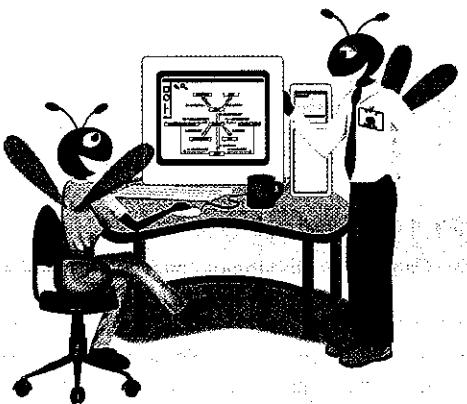
Theodore Roethke

Viaje por todo o universo em um mapa.

Miguel de Cervantes

É uma lei imutável nos negócios que palavras são palavras, explicações são explicações, promessas são promessas – mas somente desempenho é realidade.

Harold S. Green



Sumário do capítulo

- 21.1 Introdução
- 21.2 Visão geral das coleções
- 21.3 A classe `Arrays`
- 21.4 A interface `Collection` e a classe `Collections`
- 21.5 Listas
- 21.6 Algoritmos
 - 21.6.1 O algoritmo `sort`
 - 21.6.2 O algoritmo `shuffle`
 - 21.6.3 Os algoritmos `reverse`, `fill`, `copy`, `max` e `min`
 - 21.6.4 O algoritmo `binarySearch`
- 21.7 Conjuntos
- 21.8 Mapeamentos
- 21.9 Empacotadores de sincronização
- 21.10 Empacotadores não-modificáveis
- 21.11 Implementações abstratas
- 21.12 (Opcional) Descobrindo padrões de projeto: padrões de projeto usados no pacote `java.util`

[Resumo](#) • [Terminologia](#) • [Exercícios de auto-revisão](#) • [Respostas aos exercícios de auto-revisão](#) •

[Exercícios](#)

21.1 Introdução

No Capítulo 19, discutimos como criar e manipular estruturas de dados. A discussão foi “de baixo nível” no sentido que criamos meticulosamente cada elemento de cada estrutura de dados dinamicamente com `new` e modificamos as estruturas de dados manipulando diretamente seus elementos e as referências a seus elementos. Neste capítulo, analisamos a *estrutura de coleções Java* (*Java collections framework*), que oferece ao programador acesso a estruturas de dados pré-empacotadas e a algoritmos para manipular essas estruturas de dados.

Com as coleções, em vez de ter de criar estruturas de dados, o programador simplesmente utiliza estruturas de dados existentes sem se preocupar com a maneira como essas estruturas de dados são implementadas. Esse é um exemplo maravilhoso de reutilização de código. Os programadores podem codificar mais rápido e esperar excelente desempenho, maximizando a velocidade de execução e minimizando o consumo de memória. Discutiremos as interfaces da estrutura de coleções, as classes de implementação, os algoritmos que as processam e os *iteradores* que as “percorrem”.

Alguns exemplos de coleções são as cartas que você segura em um jogo de cartas, suas músicas favoritas armazenadas em computador e os registros de bens imobiliários em seu registro de escrituras local (que mapeiam números de livro/números de página para proprietários de bens imóveis). Java 2 fornece toda uma estrutura de coleções, enquanto as versões anteriores de Java forneciam apenas algumas classes de coleção, como `Hashtable`, `Stack` e `Vector` (veja o Capítulo 20), e alguns recursos para *array* predefinidos. Se você conhece C++, está familiarizado com sua estrutura de coleções, que se chama “biblioteca-padrão de gabaritos” [veja o Capítulo 20 de *C++ Como Programar, Terceira Edição*, de H. M. Deitel e P. J. Deitel].

A estrutura de coleções Java fornece componentes reutilizáveis prontos para trabalhar; você não precisa escrever suas próprias classes de coleção. As coleções são padronizadas de modo que os aplicativos possam compartilhá-las facilmente, sem se preocupar com detalhes de sua implementação. Essas coleções são escritas para ampla reutilização. Elas são ajustadas para rápida execução e utilização eficiente da memória. A estrutura de coleções incentiva a reutilização adicional. À medida que são desenvolvidos novas estruturas de dados e novos algoritmos que se ajustam a essa estrutura, haverá uma base cada vez maior de programadores familiarizados com as interfaces e algoritmos implementados por aquelas estruturas.

21.2 Visão geral das coleções

A *coleção* é uma estrutura de dados – na realidade um objeto – que pode armazenar outros objetos. As interfaces de coleção definem as operações que um programa pode executar sobre cada tipo de coleção. As implementações de coleção executam as operações de maneiras particulares, algumas mais apropriadas que outras para tipos de aplicativos específicos. As implementações da coleção são cuidadosamente construídas para execução rápida e utilização eficiente da memória. As coleções incentivam a reutilização de software fornecendo funcionalidade conveniente.

A estrutura de coleções fornece interfaces que definem as operações a serem realizadas genericamente sobre vários tipos de coleções. Algumas das interfaces são **Collection**, **Set**, **List** e **Map**. Diversas implementações dessas interfaces são fornecidas dentro da estrutura. Os programadores também podem fornecer implementações específicas para seus próprios requisitos.

A estrutura de coleções inclui uma variedade de outros recursos que minimizam a quantidade de código que os programadores precisam escrever para criar e manipular coleções.

As classes e interfaces que compreendem a estrutura de coleções são membros do pacote **java.util**. Na próxima seção, iniciamos nossa discussão examinando os recursos que foram adicionados para manipulação de *arrays*.

21.3 A classe Arrays

Iniciamos nossa discussão da estrutura de coleções dando uma olhada na classe **Arrays**, que fornece os métodos **static** para manipular *arrays*. No Capítulo 7, nossa discussão sobre manipulação de *array* foi “de baixo nível” no sentido de que escrevemos o próprio código para ordenar e pesquisar *arrays*. A classe **Arrays** fornece métodos “de alto nível”, como **binarySearch**, para pesquisar um *array* ordenado, **equals**, para comparar *arrays*, **fill**, para colocar valores em um *array*, e **sort**, para ordenar um *array*. Esses métodos são sobrecarregados para *arrays* de tipo primitivo e *arrays Object*. A Fig. 21.1 demonstra o uso desses métodos.

```

1 // Fig. 21.1: UsingArrays.java
2 // Usando arrays de Java.
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class UsingArrays {
8     private int intValues[] = { 1, 2, 3, 4, 5, 6 };
9     private double doubleValues[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
10    private int filledInt[], intValuesCopy[];
11
12    // inicializa arrays
13    public UsingArrays()
14    {
15        filledInt = new int[ 10 ];
16        intValuesCopy = new int[ intValues.length ];
17
18        Arrays.fill( filledInt, 7 );      // preenche com 7s
19
20        Arrays.sort( doubleValues );    // ordena doubleValues
21
22        System.arraycopy( intValues, 0, intValuesCopy,
23                          0, intValues.length );
24    }
25
26    // envia para a saída valores em cada array
27    public void printArrays()
28    {
29        System.out.print( "doubleValues: " );
30
31        for ( int count = 0; count < doubleValues.length; count++ )

```

Fig. 21.1 Utilizando métodos da classe **Arrays** (parte 1 de 3).

```

32         System.out.print( doubleValues[ count ] + " " );
33
34     System.out.print( "\nintValues: " );
35
36     for ( int count = 0; count < intValues.length; count++ )
37         System.out.print( intValues[ count ] + " " );
38
39     System.out.print( "\nfilledInt: " );
40
41     for ( int count = 0; count < filledInt.length; count++ )
42         System.out.print( filledInt[ count ] + " " );
43
44     System.out.print( "\nintValuesCopy: " );
45
46     for ( int count = 0; count < intValuesCopy.length; count++ )
47         System.out.print( intValuesCopy[ count ] + " " );
48
49     System.out.println();
50 }
51
52 // encontra valor no array intValues
53 public int searchForInt( int value )
54 {
55     return Arrays.binarySearch( intValues, value );
56 }
57
58 // compara conteúdo dos arrays
59 public void printEquality()
60 {
61     boolean b = Arrays.equals( intValues, intValuesCopy );
62
63     System.out.println( "intValues " + ( b ? "==" : "!=" )
64                         + " intValuesCopy" );
65
66     b = Arrays.equals( intValues, filledInt );
67
68     System.out.println( "intValues " + ( b ? "==" : "!=" )
69                         + " filledInt" );
70 }
71
72 // executa o aplicativo
73 public static void main( String args[] )
74 {
75     UsingArrays usingArrays = new UsingArrays();
76
77     usingArrays.printArrays();
78     usingArrays.printEquality();
79
80     int location = usingArrays.searchForInt( 5 );
81     System.out.println( ( location >= 0 ?
82                         "Found 5 at element " + location : "5 not found" ) +
83                         " in intValues" );
84
85     location = usingArrays.searchForInt( 8763 );
86     System.out.println( ( location >= 0 ?
87                         "Found 8763 at element " + location :
88                         "8763 not found" ) + " in intValues" );
89 }
90
91 } // fim da classe UsingArrays

```

Fig. 21.1 Utilizando métodos da classe Arrays (parte 2 de 3).

```

doubleValues: 0.2 3.4 7.9 8.4 9.3
intValues: 1 2 3 4 5 6
filledInt: 7 7 7 7 7 7 7 7 7 7
intValuesCopy: 1 2 3 4 5 6
intValues == intValuesCopy
intValues != filledInt
Found 5 at element 4 in intValues
8763 not found in intValues

```

Fig. 21.1 Utilizando métodos da classe `Arrays` (parte 3 de 3).

A linha 18 chama o método `static fill` de `Arrays` para preencher todos os 10 elementos do array `filledInt` com 7s. As versões sobrecarregadas de `fill` permitem que o programador preencha um intervalo específico de elementos com o mesmo valor.

A linha 20 ordena os elementos do array `doubleValues`. As versões sobrecarregadas de `sort` permitem que o programador ordene um intervalo específico de elementos. O método `static sort` de `Arrays` ordena os elementos do array em ordem crescente por *default*. Discutimos mais adiante neste capítulo como ordenar em ordem decrescente.

As linhas 22 e 23 copiam o array `intValues` para o array `intValuesCopy`. O primeiro argumento (`intValues`) passado para o método `static arraycopy` de `System` é o array do qual os elementos são copiados. O segundo argumento (0) é o subscrito de array (isto é, `intValues`) que especifica o ponto inicial no intervalo de elementos a copiar. Esse valor pode ser qualquer subscrito de array válido. O terceiro argumento (`intValuesCopy`) especifica o array que armazena a cópia. O quarto argumento (0) especifica o subscrito no array de destino (isto é, `intValuesCopy`) em que o primeiro elemento copiado é armazenado. O último argumento (`intValues.length`) especifica o número de elementos do array de origem (isto é, `intValues`) a copiar.

A linha 55 chama o método `static binarySearch` de `Array` para realizar uma pesquisa binária em `intValues` utilizando `value` como chave. Se `value` for encontrado, `binarySearch` devolve o subscrito da posição em que `value` foi localizado. Se o `int` não for localizado, `binarySearch` devolve um valor negativo. O valor negativo devolvido baseia-se no *ponto de inserção* da chave – o índice em que a chave seria inserida se esta fosse uma operação de inserção. Depois que `binarySearch` determina o ponto de inserção, ele muda o sinal do ponto de inserção para negativo e subtrai 1 para obter o valor devolvido. Por exemplo, na Fig. 21.1, espera-se encontrar o número 8763 no elemento com subscrito 6 no array. O método `binarySearch` muda o ponto de inserção para -6 e subtrai 1 dele, depois devolve o valor -7. Esse valor devolvido é útil para adicionar elementos a um array ordenado.



Erro comum de programação 21.1

Passar um array não-ordenado para binarySearch é um erro de lógica. O valor devolvido por binarySearch é indefinido.

As linhas 61 e 66 chamam o método `equals` para determinar se os elementos de dois arrays são equivalentes. Se eles forem iguais, o método `equals` devolve `true`; caso contrário, ele devolve `false`.

Uma das características mais importantes da estrutura de coleção é a capacidade de manipular os elementos de um tipo de coleção através de um tipo de coleção diferente, independentemente da implementação interna da coleção. O conjunto de métodos `public` através do qual as coleções são manipuladas é chamado de *visão*.

A classe `Arrays` fornece o método `static asList` para visualizar um array como um tipo de coleção `List` (um tipo que encapsula comportamento semelhante ao das listas encadeadas criadas no Capítulo 19; falaremos mais sobre `Lists` mais adiante neste capítulo). Uma visão de `List` permite manipular o array no programa como se ele fosse uma `List`, chamando os métodos de `List`. Quaisquer modificações feitas por meio da visão de `List` alteram o array e quaisquer modificações feitas no array alteram a visão de `List`. A Fig. 21.2 demonstra o método `asList`.

A linha 9 declara uma referência `List` denominada `list`. A linha 14 utiliza o método `static asList` de `Arrays` para obter uma visão de `List` de tamanho fixo do array `values`.

```

1 // Fig. 21.2: UsingAsList.java
2 // Usando o método asList
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class UsingAsList {
8     private String values[] = { "red", "white", "blue" };
9     private List list;
10
11    // inicializa List e configura valor na posição 1
12    public UsingAsList()
13    {
14        list = Arrays.asList( values ); // obtém List
15        list.set( 1, "green" ); // muda um valor
16    }
17
18    // envia para a saída List e o array
19    public void printElements()
20    {
21        System.out.print( "List elements : " );
22
23        for ( int count = 0; count < list.size(); count++ )
24            System.out.print( list.get( count ) + " " );
25
26        System.out.print( "\nArray elements: " );
27
28        for ( int count = 0; count < values.length; count++ )
29            System.out.print( values[ count ] + " " );
30
31        System.out.println();
32    }
33
34    // executa o aplicativo
35    public static void main( String args[] )
36    {
37        new UsingAsList().printElements();
38    }
39
40 } // fim da classe UsingAsList

```

```

List elements: red green blue
Array elements: red green blue

```

Fig. 21.2 Utilizando o método `static asList`.



Dica de desempenho 21.1

Utilizar `Arrays.asList` cria uma `List` de tamanho fixo que opera mais rápido que qualquer uma das implementações fornecidas de `List`.



Erro comum de programação 21.2

Uma `List` criada com `Arrays.asList` tem tamanho fixo; chamar os métodos `add` ou `remove` dispara uma `UnsupportedOperationException`.

A linha 15 chama o método `set` de `List` para mudar o conteúdo do elemento 1 de `List` para "green". Como o programa visualiza o `array` como uma `List`, a linha 15 muda o elemento do `array values[1]` de "white" para "green". Quaisquer alterações feitas na visão de `List` são feitas no objeto `array` subjacente.



Observação de engenharia de software 21.1

Com a estrutura de coleções, há muitos métodos que se aplicam a **Lists** e **Collections** que você gostaria de ser capaz de utilizar em arrays. **Arrays.asList** permite que você passe um array para um parâmetro **List** ou **Collection**.

A linha 23 chama o método **size** de **List** para obter o número de itens na **List**. A linha 24 chama o método **get** de **List** para recuperar um item individual da **List**. Observe que o valor devolvido por **size** é igual ao número de elementos no **array values** e que os itens devolvidos por **get** são os elementos do **array values**.

21.4 A interface Collection e a classe Collections

A interface **Collection** é a interface da raiz na hierarquia de coleções a partir da qual se derivam as interfaces **Set** (uma coleção que não contém duplicatas – discutida na Seção 21.7) e **List**. A interface **Collection** contém *operações de volume* (isto é, operações realizadas na coleção inteira) para adicionar, limpar, comparar e reter objetos (também chamados de *elementos*) na coleção. **Collections** também podem ser convertidas em **arrays**. Além disso, a interface **Collection** fornece um método que devolve um **Iterator**. **Iterators** são semelhantes às **Enumerations** apresentadas no Capítulo 20. A principal diferença entre um **Iterator** e uma **Enumeration** é que **Iterators** podem remover elementos de uma coleção, enquanto **Enumerations** não podem. Outros métodos da interface **Collection** permitem que um programa determine o tamanho de uma coleção, o código de hash de uma coleção e se a coleção está vazia.



Observação de engenharia de software 21.2

Collection é comumente utilizada como um tipo de parâmetro de método para permitir processamento polimórfico de todos os objetos que implementam a interface **Collection**.



Observação de engenharia de software 21.3

A maioria das implementações de coleção fornece um construtor que recebe um argumento **Collection**, permitindo, assim, que um tipo de coleção seja tratado como outro tipo de coleção.

A classe **Collections** fornece os métodos **static** que manipulam as coleções polimorficamente. Esses métodos implementam algoritmos para pesquisar, ordenar, etc. Você aprenderá mais sobre esses algoritmos na Seção 21.6. Outros métodos de **Collections** incluem *métodos empacotadores (wrapper methods)* que devolvem novas coleções. Discutimos os métodos empacotadores nas Seções 21.9 e 21.10.

21.5 Listas

A **List** é uma **Collection** ordenada que pode conter elementos duplicados. Às vezes é chamada de *sequência*. Como os **arrays**, as **Lists** baseiam-se em zero (isto é, o índice do primeiro elemento é zero). Além dos métodos da interface herdados de **Collection**, **List** fornece métodos para manipular elementos utilizando seus índices, manipular um intervalo especificado de elementos, pesquisar os elementos e obter um **ListIterator** para acessar os elementos.

A interface **List** é implementada pelas classes **ArrayList**, **LinkedList** e **Vector**. A classe **ArrayList** é uma implementação de uma **List** como um array redimensionável. O comportamento e os recursos da classe **ArrayList** são semelhantes aos da classe **Vector**, apresentada no Capítulo 20. A **LinkedList** é uma implementação de uma **List** como uma lista encadeada.



Dica de desempenho 21.2

ArrayLists se comportam como **Vectors** não *synchronized* e, portanto, rodam mais rápido que **Vectors**, porque não são seguras quanto a threads.



Observação de engenharia de software 21.4

Podem-se utilizar **LinkedLists** para criar pilhas, filas, árvores e dequeues (*double-ended queues* – filas com duas extremidades).

A Fig. 21.3 utiliza uma **ArrayList** para demonstrar alguns dos recursos da interface **Collection**. O programa coloca **Strings** e **Colors** em uma **ArrayList** e utiliza um **Iterator** para remover os **Strings** da coleção **ArrayList**.

```

1 // Fig. 21.3: CollectionTest.java
2 // Usando a interface Collection
3
4 // Pacotes do núcleo de Java
5 import java.awt.Color;
6 import java.util.*;
7
8 public class CollectionTest {
9     private String colors[] = { "red", "white", "blue" };
10
11    // cria ArrayList, adiciona objetos a ela e a manipula
12    public CollectionTest()
13    {
14        ArrayList list = new ArrayList();
15
16        // adiciona objetos à lista
17        list.add( Color.magenta );    // adiciona um objeto cor
18
19        for ( int count = 0; count < colors.length; count++ )
20            list.add( colors[ count ] );
21
22        list.add( Color.cyan );    // adiciona um objeto cor
23
24        // envia para a saída o conteúdo da lista
25        System.out.println( "\nArrayList: " );
26
27        for ( int count = 0; count < list.size(); count++ )
28            System.out.print( list.get( count ) + " " );
29
30        // remove todos os objetos String
31        removeStrings( list );
32
33        // envia para a saída o conteúdo da lista
34        System.out.println( "\n\nArrayList after calling" +
35                           " removeStrings: " );
36
37        for ( int count = 0; count < list.size(); count++ )
38            System.out.print( list.get( count ) + " " );
39    }
40
41    // remove objetos String da Collection
42    public void removeStrings( Collection collection )
43    {
44        // obtém iterador
45        Iterator iterator = collection.iterator();
46
47        // repete enquanto houver itens na coleção
48        while ( iterator.hasNext() )
49
50            if ( iterator.next() instanceof String )
51                iterator.remove();    // remove objeto String
52    }
53
54    // executa o aplicativo
55    public static void main( String args[] )

```

Fig. 21.3 Utilizando uma **ArrayList** para demonstrar a interface **Collection** (parte 1 de 2).

```

56      {
57          new CollectionTest();
58      }
59
60  } // fim da classe CollectionTest

ArrayList:
java.awt.Color[r=255,g=0,b=255] red white blue java.awt.Color
[r=0,g=255,b=255]

ArrayList after calling removeStrings:
java.awt.Color[r=255,g=0,b=255] java.awt.Color[r=0,g=255,b=255]

```

Fig. 21.3 Utilizando uma `ArrayList` para demonstrar a interface `Collection` (parte 2 de 2).

A linha 14 cria uma referência `list` e a inicializa com uma instância de uma `ArrayList`. As linhas 17 a 22 preenchem `list` com objetos `Colors` e `Strings`. As linhas 25 a 28 enviam para a saída cada elemento de `list`. A linha 27 chama o método `size` para obter o número de elementos da `ArrayList`. A linha 28 usa o método `get` para recuperar valores de elementos individuais. A linha 31 chama o método definido pelo programador `removeStrings` (definido nas linhas 42 a 52), passando `list` para ele como argumento. O método `removeStrings` exclui `Strings` de uma coleção. As linhas 34 a 38 imprimem os elementos de `list` depois que `removeStrings` remove os objetos `String` da lista. Observe que a saída na Fig. 21.3 contém somente `Colors`.

O método `removeStrings` declara um parâmetro do tipo `Collection` que permite que qualquer `Collection` seja passada como argumento para esse método. O método acessa os elementos da `Collection` utilizando um `Iterator`. A linha 45 chama o método `iterator` para obter um `Iterator` para a `Collection`. A condição do laço `while` na linha 48 chama o método `hasNext` de `Iterator` para determinar se a `Collection` contém mais algum elemento. O método `hasNext` devolve `true` se existe outro elemento, e `false` caso contrário.

A condição `if` na linha 50 chama o método `next` de `Iterator` para obter uma referência para o próximo elemento e depois usa `instanceOf` para determinar se o objeto é um `String`. Se for, a linha 51 chama o método `remove` de `Iterator` para remover o `String` da `Collection`.



Erro comum de programação 21.3

Ao interar através de uma coleção com um Iterator, use o método remove de Iterator para excluir um elemento da coleção. Usar o método remove da coleção vai resultar em uma ConcurrentModificationException.

A Fig. 21.4 demonstra as operações sobre `LinkedLists`. O programa cria duas `LinkedLists`, e cada uma delas contém `Strings`. Os elementos de uma `List` são adicionados à outra. Assim, todos os elementos são convertidos para letras maiúsculas e um intervalo de elementos é excluído.

```

1 // Fig. 21.4: ListTest.java
2 // Usando LinkLists
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class ListTest {
8     private String colors[] = { "black", "yellow", "green",
9         "blue", "violet", "silver" };
10    private String colors2[] = { "gold", "white", "brown",
11        "blue", "gray", "silver" };
12
13    // configura e manipula objetos LinkedList
14    public ListTest()

```

Fig. 21.4 Utilizando `Lists` e `ListIterators` (parte 1 de 3).

```

15  {
16      LinkedList link = new LinkedList();
17      LinkedList link2 = new LinkedList();
18
19      // adiciona elementos a cada lista
20      for ( int count = 0; count < colors.length; count++ ) {
21          link.add( colors[ count ] );
22          link2.add( colors2[ count ] );
23      }
24
25      link.addAll( link2 );           // concatena as listas
26      link2 = null;                // libera recursos
27
28      printList( link );
29
30      uppercaseStrings( link );
31
32      printList( link );
33
34      System.out.print( "\nDeleting elements 4 to 6..." );
35      removeItems( link, 4, 7 );
36
37      printList( link );
38  }
39
40  // envia para a saída o conteúdo de List
41  public void printList( List list )
42  {
43      System.out.println( "\nlist: " );
44
45      for ( int count = 0; count < list.size(); count++ )
46          System.out.print( list.get( count ) + " " );
47
48      System.out.println();
49  }
50
51  // localiza objetos String e converte para maiúsculas
52  public void uppercaseStrings( List list )
53  {
54      ListIterator iterator = list.listIterator();
55
56      while ( iterator.hasNext() ) {
57          Object object = iterator.next();    // obtém item
58
59          if ( object instanceof String )    // verifica se é String
60              iterator.set(
61                  ( String ) object ).toUpperCase() );
62      }
63  }
64
65  // obtém sublistas e usa o método clear para apagar itens da sublistas
66  public void removeItems( List list, int start, int end )
67  {
68      list.subList( start, end ).clear();    // remove itens
69  }
70
71  // executa o aplicativo
72  public static void main( String args[] )
73  {
74      new ListTest();

```

Fig. 21.4 Utilizando Lists e ListIterators (parte 2 de 3).

```

75      }
76
77 } // fim da classe ListTest

list:
black yellow green blue violet silver gold white brown blue gray silver
list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER
Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

```

Fig. 21.4 Utilizando Lists e ListIterators (parte 3 de 3).

As linhas 16 e 17 criam as `LinkedLists` `link` e `link2`, respectivamente. As linhas 20 a 23 chamam o método `add` para acrescentar elementos dos `arrays` `colors` e `colors2` ao final das `LinkedLists` `link` e `link2`, respectivamente.

A linha 25 chama o método `addAll` para acrescentar todos os elementos de `link2` ao final de `link`. A linha 26 configura `link2` como `null`, para que `LinkedList` possa ser coletada como lixo. A linha 28 chama o método definido pelo programador `printList` (linhas 41 a 49) para enviar para a saída o conteúdo de `link`. A linha 30 chama o método definido pelo programador `uppercaseStrings` (linhas 52 a 63) para converter os elementos `String` para letras maiúsculas; assim, a linha 32 chama `printList` para exibir os `Strings` modificados. A linha 34 chama o método definido pelo programador `removeItems` (linhas 66 a 69) para remover os elementos das posições 4 a 6 da lista.

O método `uppercaseStrings` (linhas 52 a 63) transforma os elementos `String` em letras minúsculas da `List` passada para ele em `Strings` em letras maiúsculas. A linha 54 chama o método `listIterator` para obter um *iterador bidirecional* para a `List` (isto é, um iterador que pode percorrer uma `List` para a frente e para trás). A condição `while` chama o método `hasNext` para determinar se a `List` contém outro elemento. A linha 57 obtém o próximo `Object` da `List` e o atribui a `object`. A condição no `if` testa `object` para saber se é uma instância (`instanceof`) da classe `String`. Caso seja, as linhas 60 e 61 convertem `object` em um `String`, chamam o método `toUpperCase` para obter uma versão em letras maiúsculas do `String` e chamam o método `set` para substituir o `String` ao qual `iterator` faz referência atualmente pelo `String` devolvido pelo método `toUpperCase`.

O método definido pelo programador `removeItems` (linhas 66 a 69) remove um intervalo de itens da lista. A linha 68 chama o método `subList` para obter uma parte da `List` chamada de `sublista`. Esta sublista é simplesmente uma visão da `List` sobre a qual `subList` é chamado. O método `subList` recebe dois argumentos – o índice inicial da sublista e o índice final da sublista. Note que o índice final não faz parte do intervalo da sublista. Nesse exemplo, passamos 4 para o índice inicial e 7 para o índice final da `subList`. A sublista devolvida são os elementos com índices 4 a 6. A seguir, o programa chama o método `clear` sobre `subList` para remover os elementos da sublista da `List`.

```

1 // Fig. 21.5: UsingToArray.java
2 // Usando o método toArray
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class UsingToArray {
8     // cria LinkedList, adiciona elementos e converte para array

```

Fig. 21.5 Utilizando o método `toArray` (parte 1 de 2).

```

10    public UsingToArray()
11    {
12        LinkedList links;
13        String colors[] = { "black", "blue", "yellow" };
14
15        links = new LinkedList( Arrays.asList( colors ) );
16
17        links.addLast( "red" );           // adiciona como último item
18        links.add( "pink" );           // adiciona ao fim
19        links.add( 3, "green" );        // adiciona no terceiro índice
20        links.addFirst( "cyan" );       // adiciona como primeiro item
21
22        // obtém elementos de LinkedList como um array
23        colors = ( String[] ) links.toArray(
24            new String[ links.size() ] );
25
26        System.out.println( "colors: " );
27
28        for ( int count = 0; count < colors.length; count++ )
29            System.out.println( colors[ count ] );
30    }
31
32    // executa o aplicativo
33    public static void main( String args[] )
34    {
35        new UsingToArray();
36    }
37
38 } // fim da classe UsingToArray

```

```

colors:
cyan
black
blue
yellow
green
red
pink

```

Fig. 21.5 Utilizando o método `toArray` (parte 2 de 2).

A Fig. 21.5 utiliza o método `toArray` para obter um *array* a partir de uma coleção (isto é, `LinkedList`). O programa adiciona uma série de `Strings` a uma `LinkedList` e chama o método `toArray` para obter um *array* da `LinkedList`.

A linha 15 constrói uma `LinkedList` que contém os elementos do *array* `colors` e atribui o `LinkedList` aos `links`. A linha 17 chama o método `addLast` para adicionar "red" ao final de `links`. As linhas 18 e 19 chamam o método `add` para adicionar "pink" como o último elemento e "green" como o elemento de índice 3 (isto é, o quarto elemento). A linha 20 chama `addFirst` para adicionar "cyan" como o novo primeiro item na `LinkedList`. [Nota: quando "cyan" é adicionado como primeiro elemento, "green" torna-se o quinto elemento na `LinkedList`.]

As linhas 23 e 24 chamam o método `toArray` para obter um *array* `String` a partir de `links`. O *array* é uma cópia dos elementos de `links` – modificar o conteúdo do *array* não modifica a `LinkedList`. O *array* passado ao método `toArray` é o mesmo tipo de dados que o devolvido por `toArray`. Se o número de elementos no *array* for maior que o número de elementos na `LinkedList`, `toArray` copia os elementos da lista em seu argumento *array* e devolve aquele *array*. Se a `LinkedList` tiver mais elementos que a quantidade de elementos no *array* passado para `toArray`, `toArray` aloca um novo *array* do mesmo tipo que ele recebe como argumento, copia os elementos da lista para o novo *array* e devolve aquele novo *array*.



Erro comum de programação 21.4

Passar um array que contém dados para `toArray` pode causar erros de lógica. Se o número de elementos no array for menor que o número de elementos no `Object` que está chamando `toArray`, nova memória será alocada para armazenar os elementos de `Object` – sem preservar os elementos do array. Se o número de elementos no array for maior que o número de elementos no `Object`, os elementos do array (iniciando em subscrito 0) serão sobrescritos com os elementos do `Object`. Os elementos do array não-sobrescritos retêm seus valores.

21.6 Algoritmos

A estrutura de coleções fornece diversos *algoritmos* de alto desempenho para manipular os elementos de coleções. Esses algoritmos são implementados como métodos `static`. Os algoritmos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` e `copy` operam sobre `Lists`. Os algoritmos `min` e `max` operam sobre `Collections`.

O algoritmo `reverse` inverte os elementos de uma `List`, `fill` configura cada elemento `List` para fazer referência a um `Object` especificado e `copy` copia as referências de uma `List` para outra `List`.



Observação de engenharia de software 21.5

Os algoritmos da estrutura de coleções são polimórficos. Isto é, cada algoritmo pode operar sobre objetos que oferecem interfaces dadas sem se preocupar com as implementações subjacentes.

21.6.1 O algoritmo `sort`

O algoritmo `sort` ordena os elementos de uma `List`. A ordem é determinada pela ordem natural do tipo dos elementos. A chamada `sort` pode especificar como segundo argumento o objeto `Comparator` que especifica como determinar a ordenação dos elementos.

O algoritmo `sort` utiliza uma *classificação estável* (isto é, uma classificação que não reordena elementos equivalentes durante a classificação). O algoritmo `sort` é rápido. Para os leitores que estudaram algo de teoria de complexidade em estruturas de dados ou cursos de algoritmos, essa classificação executa em tempo $n \log(n)$. Para os leitores não-familiarizados com a teoria da complexidade, fiquem certos de que esse é um algoritmo extremamente rápido.



Observação de engenharia de software 21.6

A documentação Java API às vezes fornece detalhes de implementação. Por exemplo, `sort` é implementado como uma classificação por intercalação modificada. Evite escrever código que é dependente de detalhes de implementação – porque eles podem mudar.

```

1 // Fig. 21.6: Sort1.java
2 // Usando o algoritmo sort
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class Sort1 {
8     private static String suits[] =
9     { "Hearts", "Diamonds", "Clubs", "Spades" };
10
11    // exibe elementos do array
12    public void printElements()
13    {
14        // cria ArrayList
15        ArrayList list = new ArrayList( Arrays.asList( suits ) );
16
17        // envia lista para a saída
18        System.out.println( "Unsorted array elements:\n" + list );
19
20        // ordena ArrayList
21        Collections.sort( list );

```

Fig. 21.6 Utilizando o algoritmo `sort` (parte 1 de 2).

```

22      // envia a lista para a saída
23      System.out.println( "Sorted array elements:\n" + list );
24  }
25
26      // executa o aplicativo
27  public static void main( String args[] )
28  {
29      new Sort1().printElements();
30  }
31
32  } // fim da classe Sort1

```

```

Unsorted array elements:
[Hearts, Diamonds, Clubs, Spades]
Sorted array elements:
[Clubs, Diamonds, Hearts, Spades]

```

Fig. 21.6 Utilizando o algoritmo `sort` (parte 2 de 2).

A Fig. 21.6 utiliza o algoritmo `sort` para ordenar os elementos de uma `ArrayList` em ordem crescente com a instrução (linha 21).

```

1 // Fig. 21.7: Sort2.java
2 // Usando um objeto Comparator com o algoritmo sort
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class Sort2 {
8     private static String suits[] =
9         { "Hearts", "Diamonds", "Clubs", "Spades" };
10
11    // envia elementos da List para a saída
12    public void printElements()
13    {
14        // cria List
15        List list = Arrays.asList( suits );
16
17        // envia elementos da List para a saída
18        System.out.println( "Unsorted array elements:\n" + list );
19
20        // ordena em ordem crescente usando um comparador
21        Collections.sort( list, Collections.reverseOrder() );
22
23        // envia elementos da List para a saída
24        System.out.println( "Sorted list elements:\n" + list );
25    }
26
27    // executa o aplicativo
28  public static void main( String args[] )
29  {
30      new Sort2().printElements();
31  }
32
33 } // fim da classe Sort2

```

Fig. 21.7 Utilizando um objeto `Comparator` em `sort` (parte 1 de 2).

```

Unsorted array elements:
[Hearts, Diamonds, Clubs, Spades]
Sorted list elements:
[Spades, Hearts, Diamonds, Clubs]

```

Fig. 21.7 Utilizando um objeto **Comparator** em **sort** (parte 2 de 2).

A Fig. 21.7 ordena os mesmos **Strings** utilizados na Fig. 21.6 em ordem decrescente. O exemplo introduz o objeto **Comparator**, para ordenar elementos de uma **Collection** em uma ordem diferente.

A linha 21 chama o método **sort** de **Collections** para ordenar a visão de **List** do **array** em ordem decrescente. O método **static** de **Collections reverseOrder** devolve um objeto **Comparator** que representa a ordem inversa da coleção. Para ordenar uma visão de **List** de um **array String**, a ordem inversa é uma *comparação lexicográfica* – o **Comparator** compara os valores Unicode que representam cada elemento – em ordem decrescente.

21.6.2 O algoritmo **shuffle**

O algoritmo **shuffle** ordena aleatoriamente os elementos de uma **List**. No Capítulo 10, apresentamos uma simulação de embaralhamento e distribuição de cartas na qual utilizamos um laço para embaralhar cartas. Na Fig. 21.8, utilizamos o algoritmo **shuffle** para embaralhar as cartas do baralho. Grande parte do código é o mesmo da Fig. 10.21.

O embaralhamento é implementado na linha 63 que chama o método **static shuffle** de **Collections** para embaralhar o **array** por meio da visão de **List** do **array**.

```

1 // Fig. 21.8: Cards.java
2 // Usando o algoritmo para embaralhar
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 // classe que representa uma carta em um baralho
8 class Card {
9     private String face;
10    private String suit;
11
12    // inicializa uma Card
13    public Card( String initialface, String initialSuit ) {
14        face = initialface;
15        suit = initialSuit;
16    }
17
18    // devolve a face da Card
19    public String getFace()
20    {
21        return face;
22    }
23
24    // devolve o naipe da Card
25    public String getSuit()
26    {
27        return suit;
28    }
29
30    // devolve a representação da Card como String
31    public String toString()
32    {

```

Fig. 21.8 Exemplo de embaralhamento e distribuição de cartas (parte 1 de 3).

```

34     StringBuffer buffer =
35         new StringBuffer( face + " of " + suit );
36
37     buffer.setLength( 20 );
38
39     return buffer.toString();
40 }
41
42 } // fim da classe Card
43
44 // definição da classe Cards
45 public class Cards {
46     private static String suits[] =
47         { "Hearts", "Clubs", "Diamonds", "Spades" };
48     private static String faces[] = { "Ace", "Deuce", "Three",
49         "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten",
50         "Jack", "Queen", "King" };
51     private List list;
52
53     // configura baralho e embaralha
54     public Cards()
55     {
56         Card deck[] = new Card[ 52 ];
57
58         for ( int count = 0; count < deck.length; count++ )
59             deck[ count ] = new Card( faces[ count % 13 ],
60                 suits[ count / 13 ] );
61
62         list = Arrays.asList( deck ); // obtém List
63         Collections.shuffle( list ); // embaralha
64     }
65
66     // envia baralho para a saída
67     public void printCards()
68     {
69         int half = list.size() / 2 - 1;
70
71         for ( int i = 0, j = half; i <= half; i++, j++ )
72             System.out.println(
73                 list.get( i ).toString() + list.get( j ) );
74     }
75
76     // executa o aplicativo
77     public static void main( String args[] )
78     {
79         new Cards().printCards();
80     }
81
82 } // fim da classe Cards

```

King of Diamonds	Ten of Spades
Deuce of Hearts	Five of Spades
King of Clubs	Five of Clubs
Jack of Diamonds	Jack of Spades
King of Spades	Ten of Clubs
Six of Clubs	Three of Clubs
Seven of Clubs	Jack of Clubs
Seven of Hearts	Six of Spades

Fig. 21.8 Exemplo de embaralhamento e distribuição de cartas (parte 2 de 3).

Eight of Hearts	Six of Diamonds
King of Hearts	Nine of Diamonds
Ace of Hearts	Four of Hearts
Jack of Hearts	Queen of Diamonds
Queen of Clubs	Six of Hearts
Seven of Diamonds	Ace of Spades
Three of Spades	Deuce of Spades
Seven of Spades	Five of Diamonds
Ten of Hearts	Queen of Hearts
Ten of Diamonds	Eight of Clubs
Nine of Spades	Three of Diamonds
Four of Spades	Ace of Clubs
Four of Clubs	Four of Diamonds
Nine of Clubs	Three of Hearts
Eight of Diamonds	Deuce of Diamonds
Deuce of Clubs	Nine of Hearts
Eight of Spades	Five of Hearts
Ten of Spades	Queen of Spades

Fig. 21.8 Exemplo de embaralhamento e distribuição de cartas (parte 3 de 3).

21.6.3 Os algoritmos `reverse`, `fill`, `copy`, `max` e `min`

A classe `Collections` fornece algoritmos para inverter, preencher e copiar `Lists`. O algoritmo `reverse` inverte a ordem dos elementos em uma `List`, e o algoritmo `fill` sobrescreve elementos em uma `List` com um valor especificado. A operação `fill` é útil para reinicializar uma `List`. O algoritmo `copy` recebe dois argumentos: uma `List` de destino e uma `List` de origem. Cada elemento da `List` de origem é copiado para a `List` de destino. A `List` de destino deve ser pelo menos tão longa quanto a `List` de origem – caso contrário, é disparada uma `IndexOutOfBoundsException`. Se a `List` de destino for mais longa, os elementos não-sobrescritos permanecem inalterados.

Cada um dos algoritmos que vimos até agora opera sobre `Lists`. Os algoritmos `min` e `max` operam sobre `Collections`.

O algoritmo `min` devolve o menor elemento em uma `List` (lembre-se de que uma `List` é uma `Collection`) e o algoritmo `max` devolve o maior elemento em uma `List`. Esses dois algoritmos podem ser chamados com um objeto `Comparator` como segundo argumento. A Fig. 21.9 demonstra o uso de algoritmos `reverse`, `fill`, `copy`, `min` e `max`.

```

1 // Fig. 21.9: Algorithms1.java
2 // Usando os algoritmos reverse, fill, copy, min e max
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class Algorithms1 {
8     private String letters[] = { "P", "C", "M" }, lettersCopy[];
9     private List list, copyList;
10
11    // cria uma List e a manipula com algoritmos
12    // da classe Collections
13    public Algorithms1()
14    {
15        list = Arrays.asList( letters ); // obtém List
16        lettersCopy = new String[ 3 ];

```

Fig. 21.9 Utilizando os algoritmos `reverse`, `fill`, `copy`, `max` e `min` (parte 1 de 2).

```

17     copyList = Arrays.asList( lettersCopy );
18
19     System.out.println( "Printing initial statistics: " );
20     printStatistics( list );
21
22     Collections.reverse( list );    // inverte a ordem
23     System.out.println( "\nPrinting statistics after " +
24         "calling reverse: " );
25     printStatistics( list );
26
27     Collections.copy( copyList, list ); // copia List
28     System.out.println( "\nPrinting statistics after " +
29         "copying: " );
30     printStatistics( copyList );
31
32     System.out.println( "\nPrinting statistics after " +
33         "calling fill: " );
34     Collections.fill( list, "R" );
35     printStatistics( list );
36 }
37
38 // envia informações de List para a saída
39 private void printStatistics( List listRef )
40 {
41     System.out.print( "The list is: " );
42
43     for ( int k = 0; k < listRef.size(); k++ )
44         System.out.print( listRef.get( k ) + " " );
45
46     System.out.print( "\nMax: " + Collections.max( listRef ) );
47     System.out.println(
48         " Min: " + Collections.min( listRef ) );
49 }
50
51 // executa o aplicativo
52 public static void main( String args[] )
53 {
54     new Algorithms1();
55 }
56
57 } // fim da classe Algorithms1

```

```

Printing initial statistics:
The list is: P C M
Max: P Min: C
Printing statistics after calling reverse:
The list is: M C P
Max: P Min: C
Printing statistics after copying:
The list is: M C P
Max: P Min: C
Printing statistics after calling fill:
The list is: R R R
Max: R Min: R

```

Fig. 21.9 Utilizando os algoritmos `reverse`, `fill`, `copy`, `max` e `min` (parte 2 de 2).

A linha 22 chama o método `reverse` de `Collections` para inverter a ordem de `list`. O método `reverse` recebe um argumento `List` (`list` é uma visão de `List` do array de `String` `letters`). O array `letters` tem agora seus elementos em ordem inversa.

A linha 27 copia os elementos da `list` para a `copyList` com o método `copy` de `Collections`. As alterações em `copyList` não mudam as `letters` – esta é uma `List` separada que não é uma visão de `List` para `letters`. O método `copy` exige dois argumentos `List`.

A linha 34 chama o método `fill` de `Collections` para colocar o `String "R"` em cada elemento de `list`. Uma vez que `list` é uma visão de `letters` como `List`, esta operação muda cada elemento em `letters` para `"R"`. O método `fill` exige uma `List` para o primeiro argumento e um `Object` para o segundo argumento.

As linhas 46 e 48 chamam os métodos `max` e `min` de `Collection` para localizar o maior elemento e o menor elemento, respectivamente, em `list`.

21.6.4 O algoritmo `binarySearch`

Anteriormente, estudamos o algoritmo de pesquisa binária de alta velocidade. Esse algoritmo está diretamente disponível na estrutura de coleções Java. O algoritmo `binarySearch` localiza um `Object` em uma `List` (isto é, `LinkedList`, `Vector` e `ArrayList`). Se o `Object` for localizado, o índice (posição em relação a 0) daquele `Object` é devolvido. Se o `Object` não for localizado, `binarySearch` devolve um valor negativo. O algoritmo `binarySearch` determina esse valor negativo primeiramente calculando o ponto de inserção e alterando o sinal do ponto de inserção para negativo. Por fim, `binarySearch` subtrai um do ponto de inserção para obter o valor de retorno.

A Fig. 21.10 utiliza o algoritmo `binarySearch` para procurar uma série de `Strings` em uma `ArrayList`.

```

1 // Fig. 21.10: BinarySearchTest.java
2 // Usando o algoritmo binarySearch
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class BinarySearchTest {
8     private String colors[] = { "red", "white", "blue", "black",
9         "yellow", "purple", "tan", "pink" };
10    private ArrayList list; // referência para ArrayList
11
12    // cria, ordena e envia para a saída a lista
13    public BinarySearchTest()
14    {
15        list = new ArrayList( Arrays.asList( colors ) );
16        Collections.sort( list ); // ordena a ArrayList
17        System.out.println( "Sorted ArrayList: " + list );
18    }
19
20    // pesquisa diversos valores na lista
21    public void printSearchResults()
22    {
23        printSearchResultsHelper( colors[ 3 ] ); // primeiro item
24        printSearchResultsHelper( colors[ 0 ] ); // item do meio
25        printSearchResultsHelper( colors[ 7 ] ); // último item
26        printSearchResultsHelper( "aardvark" ); // abaixo do menor
27        printSearchResultsHelper( "goat" ); // não existe
28        printSearchResultsHelper( "zebra" ); // não existe
29    }
30
31    // método auxiliar para fazer pesquisas
32    private void printSearchResultsHelper( String key )
33    {
34        int result = 0;
35
36        System.out.println( "\nSearching for: " + key );

```

Fig. 21.10 Utilizando o algoritmo `binarySearch` (parte 1 de 2).

```

37     result = Collections.binarySearch( list, key );
38     System.out.println(
39         ( result >= 0 ? "Found at index " + result :
40             "Not Found (" + result + ")" ) );
41 }
42
43 // executa o aplicativo
44 public static void main( String args[] )
45 {
46     new BinarySearchTest().printSearchResults();
47 }
48
49 } // fim da classe BinarySearchTest

```

```

Sorted ArrayList: black blue pink purple red tan white yellow
Searching for: black
Found at index 0

Searching for: red
Found at index 4

Searching for: pink
Found at index 2

Searching for: aardvark
Not Found (-1)

Searching for: goat
Not Found (-3)

Searching for: zebra
Not Found (-9)

```

Fig. 21.10 Utilizando o algoritmo `binarySearch` (parte 2 de 2).

A linha 16 chama o método `sort` de `Collections` para ordenar a `list` em ordem crescente. A linha 37 chama o método `binarySearch` de `Collections` para procurar a `key` especificada em `list`. O método `binarySearch` recebe uma `List` como primeiro argumento e um `Object` como segundo argumento. Uma versão sobrecarregada de `binarySearch` recebe um objeto `Comparator` como terceiro argumento, para especificar como `binarySearch` deve comparar os elementos.

Se a chave de pesquisa for localizada, o método `binarySearch` devolve o índice `List` do elemento que contém a chave de pesquisa. Quando uma chave de pesquisa é localizada na `List`, o valor devolvido por `binarySearch` é maior que ou igual a zero. Se a chave de pesquisa não for localizada, o método `binarySearch` devolve um número negativo.



Observação de engenharia de software 21.7

Java não garante qual item será localizado primeiro quando uma `binarySearch` for realizada em uma `List` que contém múltiplos elementos equivalentes à chave de pesquisa.

21.7 Conjuntos

O `Set` é uma `Collection` que contém elementos únicos (isto é, elementos não-duplicados). A estrutura de coleções contém duas implementações de `Set` – `HashSet` e `TreeSet`. `HashSet` armazena seus elementos em uma tabela de `hash` e `TreeSet` armazena seus elementos em uma árvore. A Fig. 21.11 utiliza um `HashSet` para remover `Strings` duplicados de uma `ArrayList`.

O método definido pelo programador `printNonDuplicates` (linhas 23 a 35) recebe um argumento `Collection`. A linha 26 constrói um `HashSet` da `Collection` recebida como argumento para `printNonDuplicates`. Quando o `HashSet` é construído, ele remove quaisquer duplicatas na `Collection`. Por definição, `Sets` não contêm nenhuma duplicata. A linha 27 obtém um `Iterator` para o `HashSet`. O laço

`while` (linhas 31 e 32) chama os métodos `hasNext` e `next` de `Iterator` para acessar os elementos de `HashSet`.

A interface `SortedSet` estende `Set` e mantém seus elementos em ordem de classificação (isto é, a ordem natural dos elementos ou uma ordem especificada por um `Comparator`). A classe `TreeSet` implementa `SortedSet`.

```

1 // Fig. 21.11: SetTest.java
2 // Usando um HashSet para eliminar duplicatas
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class SetTest {
8     private String colors[] = { "red", "white", "blue",
9         "green", "gray", "orange", "tan", "white", "cyan",
10        "peach", "gray", "orange" };
11
12    // cria e envia para a saída uma ArrayList
13    public SetTest()
14    {
15        ArrayList list;
16
17        list = new ArrayList( Arrays.asList( colors ) );
18        System.out.println( "ArrayList: " + list );
19        printNonDuplicates( list );
20    }
21
22    // cria conjunto a partir do array para eliminar duplicatas
23    public void printNonDuplicates( Collection collection )
24    {
25        // cria um HashSet e obtém seu iterador
26        HashSet set = new HashSet( collection );
27        Iterator iterator = set.iterator();
28
29        System.out.println( "\nNonduplicates are: " );
30
31        while ( iterator.hasNext() )
32            System.out.print( iterator.next() + " " );
33
34        System.out.println();
35    }
36
37    // executa o aplicativo
38    public static void main( String args[] )
39    {
40        new SetTest();
41    }
42
43 } // fim da classe SetTest

```

```

ArrayList: [red, white, blue, green, gray, orange, tan, white, cyan,
peach, gray, orange]

Nonduplicates are:
orange cyan green tan white blue peach red gray

```

Fig. 21.11 Utilizando um `HashSet` para remover duplicatas.

O programa da Fig. 21.12 coloca `Strings` em uma `TreeSet`. Os `Strings` são automaticamente ordenados à medida que são adicionados a `TreeSet`. Além disso, são demonstrados nesse exemplo métodos de *visão de intervalo* (isto é, métodos que permitem visualizar uma parte de uma coleção).

```

1 // Fig. 21.12: SortedSetTest.java
2 // Usando TreeSet e SortedSet
3
4 // Pacotes do núcleo de Java
5 import java.util.*;
6
7 public class SortedSetTest {
8     private static String names[] = { "yellow", "green", "black",
9         "tan", "grey", "white", "orange", "red", "green" };
10
11    // cria um conjunto ordenado com TreeSet, depois o manipula
12    public SortedSetTest()
13    {
14        TreeSet tree = new TreeSet( Arrays.asList( names ) );
15
16        System.out.println( "set: " );
17        printSet( tree );
18
19        // obtém headSet baseado em "orange"
20        System.out.print( "\nheadSet (\"orange\"): " );
21        printSet( tree.headSet( "orange" ) );
22
23        // obtém tailSet baseado em "orange"
24        System.out.print( "tailSet (\"orange\"): " );
25        printSet( tree.tailSet( "orange" ) );
26
27        // obtém o primeiro e o último elementos
28        System.out.println( "first: " + tree.first() );
29        System.out.println( "last : " + tree.last() );
30    }
31
32    // envia o conjunto para a saída
33    public void printSet( SortedSet set )
34    {
35        Iterator iterator = set.iterator();
36
37        while ( iterator.hasNext() )
38            System.out.print( iterator.next() + " " );
39
40        System.out.println();
41    }
42
43    // executa o aplicativo
44    public static void main( String args[] )
45    {
46        new SortedSetTest();
47    }
48
49 } // fim da classe SortedSetTest

```

```

set:
black green grey orange red tan white yellow

headSet ("orange"): black green grey
tailSet ("orange"): orange red tan white yellow
first: black
last: yellow

```

Fig. 21.12 Utilizando SortedSets e TreeSets.

A linha 14 constrói um objeto `TreeSet` que contém o elemento de `names` e atribui uma referência para esse objeto a `tree`. A linha 21 chama o método `headSet` para obter um subconjunto de `TreeSet` menor que "orange". Quaisquer alterações feitas no subconjunto são feitas em `TreeSet` (isto é, o subconjunto devolvido é uma visão do `TreeSet`). A linha 25 chama o método `tailSet` para obter um subconjunto maior que ou igual a "orange". Como `headSet`, quaisquer alterações feitas por meio da visão `tailSet` são feitas em `TreeSet`. As linhas 28 e 29 chamam os métodos `first` e `last` para obter os menores e maiores elementos, respectivamente.

O método definido pelo programador `printSet` (linhas 33 a 41) recebe um `SortedSet` (por exemplo, um `TreeSet`) como argumento e o imprime. A linha 35 obtém um `Iterator` para o `Set`. O corpo do laço `while` imprime cada elemento do `SortedSet`.

21.8 Mapeamentos

`Maps` associam chaves a valores e não podem conter chaves duplicadas (isto é, cada chave pode mapear somente um valor – este tipo de mapeamento se chama *mapeamento um para um*). `Maps` diferem de `Sets` no fato de que `Maps` contêm chaves e valores, ao passo que `Sets` contêm somente chaves. As classes `HashMap` e `TreeMap` implementam a interface `Map`. `HashMaps` armazenam elementos em `Hashtables` e `TreeMaps` armazenam os elementos em árvores. A interface `SortedMap` estende `Map` e mantém suas chaves ordenadas (isto é, a ordem natural dos elementos ou uma ordem especificada por um `Comparator`). A classe `TreeMap` implementa `SortedMap`.

A Fig. 21.13 utiliza um `HashMap` para contar o número de `Strings` que iniciam com uma determinada letra. [Nota: diferentemente da classe `Hashtable`, a classe `HashMap` permite chaves `null` e valores `null`.]

```

1 // Fig. 21.13: MapTest.java
2 // Usando um HashMap para armazenar a quantidade de
3 // palavras que começam com uma determinada letra
4
5 // Pacotes do núcleo de Java
6 import java.util.*;
7
8 public class MapTest {
9     private static String names[] = { "one", "two", "three",
10         "four", "five", "six", "seven", "two", "ten", "four" };
11
12     // constrói um HashMap e envia seu conteúdo para a saída
13     public MapTest()
14     {
15         HashMap map = new HashMap();
16         Integer i;
17
18         for ( int count = 0; count < names.length; count++ ) {
19             i = ( Integer ) map.get(
20                 new Character( names[ count ].charAt( 0 ) ) );
21
22             // se a chave não estiver no mapa, então atribui a ela o valor um
23             // caso contrário, incrementa seu valor por 1
24             if ( i == null )
25                 map.put(
26                     new Character( names[ count ].charAt( 0 ) ),
27                     new Integer( 1 ) );
28             else
29                 map.put(
30                     new Character( names[ count ].charAt( 0 ) ),
31                     new Integer( i.intValue() + 1 ) );
32     }

```

Fig. 21.13 Utilizando `HashMaps` e `Maps` (parte 1 de 2).

```

33
34     System.out.println(
35         "\nnumber of words beginning with each letter: " );
36     printMap( map );
37 }
38
39 // envia o conteúdo do mapa para a saída
40 public void printMap( Map mapRef )
41 {
42     System.out.println( mapRef.toString() );
43     System.out.println( "size: " + mapRef.size() );
44     System.out.println( "isEmpty: " + mapRef.isEmpty() );
45 }
46
47 // executa o aplicativo
48 public static void main( String args[] )
49 {
50     new MapTest();
51 }
52
53 } // fim da classe MapTest

```

```

number of words beginning with each letter:
{t=4, s=2, o=1, f=3}
size: 4
isEmpty: false

```

Fig. 21.13 Utilizando HashMaps e Maps (parte 2 de 2).

A linha 15 constrói **HashMap map**. O laço **for** das linhas 18 a 32 utiliza **map** para armazenar o número de palavras de **names** que iniciam com uma determinada letra. As linhas 19 e 20 chamam o método **get** para recuperar um **Character** (a primeira letra de um **String** em **names**) do **HashMap**. Se o **HashMap** não contiver um mapeamento para o **Character**, **get** devolve **null**. Se o **HashMap** contiver o mapeamento para o **Character**, seu valor de mapeamento é devolvido como um **Object**. O valor devolvido é convertido para **Integer** e é atribuído a **i**.

Se **i** for **null**, o **Character** não está no **HashMap** e as linhas 25 a 27 chamam o método **put** para gravar um **Integer** que contém 1 no **HashMap**. O valor **Integer** armazenado no **HashMap** é o número de palavras que iniciam com esse **Character**.

Se o **Character** estiver no **HashMap**, as linhas 29 a 31 incrementam o contador **Integer** de 1 e escrevem o contador atualizado no **HashMap**. O **HashMap** não pode conter duplicatas, de modo que **put** substitui o objeto **Integer** anterior pelo novo.

O método definido pelo programador **printMap** recebe um argumento **Map** e imprime, utilizando o método **toString**. As linhas 43 e 44 chamam os métodos **size** e **isEmpty** para obter o número de valores no **Map** e um booleano indicando se o **Map** está vazio, respectivamente.

21.9 Empacotadores de sincronização

No Capítulo 15, discutimos *multithreading*. As coleções predefinidas não são sincronizadas. O acesso simultâneo a uma **Collection** por múltiplas **threads** poderia ocasionar resultados indeterminados ou erros fatais. Para evitar problemas em potencial com **threads**, são utilizados *empacotadores de sincronização* em volta de classes de coleção que podem ser acessadas por múltiplas **threads**. A classe **empacotadora** recebe chamadas dos métodos, adiciona alguma funcionalidade para segurança quanto a **threads** e então delega as chamadas à classe empacotada.

A API **Collections** fornece um conjunto de métodos **public static** para converter coleções em versões sincronizadas. Os cabeçalhos de método para os empacotadores de sincronização estão listados na Fig. 21.14.

21.10 Empacotadores não-modificáveis

A API `Collections` fornece um conjunto de métodos `public static` para converter coleções em versões não-modificáveis (denominadas *empacotadores não-modificáveis*) dessas coleções. Os cabeçalhos de método para aqueles métodos estão listados na Fig. 21.15. Os empacotadores não-modificáveis disparam `UnsupportedOperationException` se forem feitas tentativas de modificar a coleção.



Observação de engenharia de software 21.8

Ao criar um empacotador não-modificável, não manter uma referência à coleção de apoio assegura a não-modificabilidade.

Cabeçalho de método `public static`

```
Collection synchronizedCollection( Collection c )
List synchronizedList( List aList )
Set synchronizedSet( Set s )
SortedSet synchronizedSortedSet( SortedSet s )
Map synchronizedMap( Map m )
SortedMap synchronizedSortedMap( SortedMap m )
```

Fig. 21.14 Métodos empacotadores de sincronização.

Cabeçalho de método `public static`

```
Collection unmodifiableCollection( Collection c )
List unmodifiableList( List aList )
Set unmodifiableSet( Set s )
SortedSet unmodifiableSortedSet( SortedSet s )
Map unmodifiableMap( Map m )
SortedMap unmodifiableSortedMap( SortedMap m )
```

Fig. 21.15 Métodos empacotadores não-modificáveis.



Observação de engenharia de software 21.9

Você pode utilizar um empacotador não-modificável para criar uma coleção que oferece acesso somente para leitura às outras pessoas enquanto permite o acesso para leitura e gravação para você. Você faz isto simplesmente dando às outras pessoas uma referência para o empacotador não-modificável enquanto mantém uma referência para a própria coleção empacotada.

21.11 Implementações abstratas

A estrutura de coleções fornece várias *implementações abstratas* (isto é, “esqueletos” de implementações de interfaces de coleção a partir das quais o programador pode rapidamente “dar corpo” a implementações personalizadas completas). Essas implementações abstratas são: uma implementação “magra” de `Collection`, chamada `AbstractCollection`, uma implementação “magra” de `List`, com suporte para acesso aleatório, chamada `AbstractList`, uma implementação “magra” de `Map` chamada `AbstractMap`, uma implementação “magra” de `List`, com suporte para acesso seqüencial, chamada `AbstractSequentialList` e uma implementação “magra” de `Set` chamada `AbstractSet`.

Para escrever uma implementação personalizada, comece selecionando como base a classe de implementação abstrata que melhor atenda às suas necessidades. A seguir, implemente cada um dos métodos `abstract` da classe. Depois, se sua coleção precisar ser modificável, sobrescreva qualquer método concreto que impeça a modificação.

21.12 (Opcional) Descobrindo padrões de projeto: padrões de projeto usados no pacote `java.util`

Nesta seção, usamos o material sobre estruturas de dados e coleções discutido nos Capítulos 19, 20 e 21 para identificar classes do pacote `java.util` que usam padrões de projeto. Esta seção conclui nossa abordagem de padrões de projeto.

21.12.1 Padrões de criação de projeto

Concluímos a discussão de padrões de criação discutindo o padrão de projeto *Prototype*.

Prototype

Algumas vezes, o sistema precisa fazer uma cópia de um objeto, mas só saberá qual é a classe daquele objeto durante a execução. Por exemplo, considere o programa para desenhar do Exercício 9.28 – as classes `MyLine`, `MyOval` e `MyRect` representam classes de “formas” que estendem a superclasse abstrata `MyShape`. Poderíamos modificar este exercício para permitir que o usuário crie, copie e cole novas instâncias da classe `MyLine` dentro do programa. O padrão de projeto *Prototype* permite a um objeto – chamado de *protótipo* – devolver uma cópia daquele protótipo para um objeto que a solicite – chamado de *cliente*. Cada protótipo deve pertencer a uma classe que implementa uma interface comum que permite ao protótipo fazer um clone de si mesmo. Por exemplo, a API Java fornece o método `clone` da classe `java.lang.Object` e a interface `java.lang.Cloneable` – qualquer objeto de uma classe que implemente `Cloneable` pode usar o método `clone` para copiar a si mesmo. Especificamente, o método `clone` cria uma cópia de um objeto e depois devolve uma referência para aquele objeto. Se designarmos a classe `MyLine` como o protótipo para o Exercício 9.28, então a classe `MyLine` deve implementar a interface `Cloneable`. Para criar uma nova linha em nosso desenho, fazemos um clone do protótipo `MyLine`. Para copiar uma linha já existente, fazemos um clone daquele objeto. O método `clone` também é útil em métodos que devolvem uma referência para um objeto, mas o desenvolvedor não quer que aquele objeto seja alterado através daquela referência – o método `clone` devolve uma referência para a cópia do objeto em vez de devolver uma referência para aquele objeto. Para obter mais informações sobre a interface `Cloneable`, visite

www.java.sun.com/j2se/1.3/docs/api/java/lang/Cloneable.html

21.12.2 Padrões comportamentais de projeto

Concluímos a discussão de padrões comportamentais discutindo o padrão de projeto *Iterator*.

Iterator

Os projetistas usam estruturas de dados como *arrays*, listas encadeadas e tabelas de *hash*, para organizar dados em um programa. O padrão de projeto *Iterator* permite que os objetos acessem objetos individuais de qualquer estrutura de dados, sem conhecer o comportamento da estrutura de dados (tal como percorrer a estrutura de dados ou remover um elemento daquela estrutura) ou como aquela estrutura de dados armazena objetos. As instruções para percorrer a estrutura de dados e acessar seus elementos são armazenadas em um objeto separado que se chama *iterador*. Cada estrutura de dados pode criar um iterador – cada um implementa métodos de uma interface comum para percorrer a estrutura de dados e acessar seus dados. Um objeto pode percorrer duas estruturas de dados estruturadas de forma diferente – como uma lista encadeada e uma tabela de *hash* – da mesma maneira, porque as duas estruturas de dados contêm um objeto iterador que pertence a uma classe que implementa uma interface comum. Java oferece a interface `Iterator` do pacote `java.util`, que discutimos na Seção 21.5 – a classe `CollectionTest` (Fig. 21.3) usa um objeto `Iterator`.

21.12.3 Conclusão

Em nossas seções opcionais “Descobrindo padrões de projeto”, apresentamos a importância, a utilidade e a predominância dos padrões de projeto. Mencionamos que, em seu livro *Design Patterns, Elements of Reusable Object-Oriented Software*, a “gangue dos quatro” descreveu 23 padrões de projeto que oferecem estratégias já testadas para a construção de sistemas. Cada padrão pertence a uma de três categorias de padrões: de criação, voltada para aspectos relacionados com a criação de objetos; estruturais, que fornecem maneiras de organizar classes e objetos em um sistema; e comportamentais, que oferecem estratégias para modelar como os objetos colaboram uns com os outros em um sistema.

Dos 23 padrões de projeto, discutimos 18 dos mais populares usados pela comunidade de Java. Nas Seções 9.24, 13.18, 15.13, 17.11 e 21.12, dividimos a discussão de acordo com a maneira como certos pacotes de Java – como os pacotes `java.awt`, `javax.swing`, `java.io`, `java.net` e `java.util` – usam aqueles padrões de projeto. Também discutimos padrões não-descritos pela “gangue dos quatro”, como padrões de simultaneidade, que são úteis em sistemas com múltiplas *threads*, e padrões de arquitetura, que ajudam os projetistas a atribuir funcionalidade aos vários subsistemas de um sistema. Mostramos a motivação de cada padrão – isto é, explicamos porque aquele padrão é importante e como ele pode ser usado. Quando apropriado, fornecemos diversos exemplos na forma de analogias com o mundo real (por exemplo, o adaptador no padrão de projeto *Adapter* é semelhante a um adaptador para um *plug* de um aparelho elétrico). Também demos exemplos de como os pacotes de Java tiram proveito de padrões de projeto (por exemplo, os componentes da GUI Swing usam o padrão de projeto *Observer* para colaborar com seus ouvintes (*listeners*) para responder a interações do usuário). Também fornecemos exemplos de como certos programas em *Java Como Programar – 4^a Edição* usaram padrões de projeto (por exemplo o estudo de caso de simulação de elevador em nossas seções opcionais “Pensando em objetos” usa o padrão de projeto *State* para representar a posição de um objeto *Person* na simulação).

Esperamos que você encare nossas seções “Descobrindo padrões de projeto” como um começo para o estudo adicional de padrões de projeto. Se você ainda não o fez, recomendamos que você visite os muitos URLs que fornecemos na Seção 9.24.5. Recomendamos que você leia o livro da gangue dos quatro. Estas informações vão ajudá-lo a construir sistemas melhores usando a sabedoria coletiva do setor da tecnologia de objetos.

Se você estudou as seções opcionais deste livro, você foi apresentado a sistemas Java mais substanciais. Se você leu nossas seções opcionais “Pensando em objetos”, você mergulhou a si mesmo em uma experiência de projeto e implementação substancial em Java, aprendendo uma abordagem disciplinada para o projeto orientado a objetos com a UML. Se você leu nossas seções opcionais “Descobrindo padrões de projeto”, você aumentou sua conscientização sobre os tópicos mais avançados de padrões de projeto.

Esperamos que você continue seu estudo de padrões de projeto e agradeceríamos muito se você nos enviasse seus comentários, suas críticas e sugestões para melhorar *Java Como Programar*, para o endereço deitel@deitel.com. Boa sorte!

Resumo

- A estrutura de coleções de Java dá ao programador acesso a estruturas de dados pré-empacotadas e a algoritmos para manipular essas estruturas de dados.
- Java 2 fornece uma estrutura inteira de coleções, enquanto as versões anteriores de Java forneciam apenas algumas classes de coleção, como `Hashtable` e `Vector`, além de recursos para *array* predefinidos.
- A coleção é uma estrutura de dados; na realidade, é um objeto que pode armazenar outros objetos. As interfaces de coleção definem as operações que podem ser realizadas sobre cada tipo de coleção.
- A estrutura de coleções inclui diversos outros recursos que minimizam a quantidade de trabalho de que os programadores necessitam para criar e manipular coleções. Essa estrutura é uma implementação efetiva do conceito de reutilização.
- As classes e interfaces que compõem a estrutura de coleções são membros do pacote `java.util`.
- A classe `Arrays` fornece métodos `static` para manipular *arrays*. Os métodos da classe `Arrays` incluem `binarySearch` para pesquisar um *array* ordenado, `equals` para comparar *arrays*, `fill` para colocar itens em um *array*, `sort` para ordenar um *array* e `asList`.
- A classe `Arrays` fornece o método `asList` para obter uma “visão de *List*” do *array*. A visão de *List* permite manipular o *array* como se ele fosse uma *List*. Isso permite que o programador trate um *array* como uma coleção.

Quaisquer modificações feitas por meio da visão de `List` alteram o `array` e quaisquer modificações no `array` alteram a visão de `List`.

- O método `size` obtém o número de itens em uma `List` e o método `get` obtém um elemento `List` individual.
- A interface `Collection` é a interface-raiz da hierarquia de coleções a partir da qual as interfaces `Set` e `List` são derivadas. A interface `Collection` contém operações de volume para adicionar, limpar, comparar e reter objetos na coleção.
- A interface `Collection` fornece um método `iterator` para obter um `Iterator`.
- A classe `Collections` fornece os métodos `static` para manipular coleções. Muitos dos métodos são implementações de algoritmos polimórficos para pesquisar, ordenar, etc.
- A `List` é uma `Collection` ordenada que pode conter elementos duplicados. Às vezes, ela é chamada de *sequência*.
- A interface `List` é implementada pelas classes `ArrayList`, `LinkedList` e `Vector`. A classe `ArrayList` é uma implementação de uma `List` como um `array` redimensionável. Comportamento e capacidades da `ArrayList` são semelhantes àqueles da classe `Vector`. A `LinkedList` é uma implementação de uma `List` como lista encadeada.
- O método `hasNext` de `Iterator` determina se uma `Collection` contém outro elemento. O método `hasNext` devolve `true` se existir outro elemento e `false` caso contrário. O método `next` devolve o próximo objeto na `Collection` e avança o `Iterator`.
- O método `subList` obtém uma parte da `List` chamada de *sublista*. Quaisquer alterações feitas em uma sublista também são feitas na `List` (isto é, a sublista é uma “visão de lista” de seus elementos `List` correspondentes).
- O método `clear` remove elementos de uma `List`.
- O método `toArray` devolve o conteúdo de uma coleção como um `array`.
- Os algoritmos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` e `copy` operam sobre `Lists`. Os algoritmos `min` e `max` operam sobre `Collections`. O algoritmo `reverse` inverte os elementos de uma `List`, `fill` configura cada elemento da `List` como um `Object` especificado e `copy` copia os elementos de uma `List` em outra `List`. O algoritmo `sort` ordena os elementos de uma `List`.
- Os algoritmos `min` e `max` localizam o menor item e o maior item em uma `Collection`.
- O objeto `Comparator` fornece um meio de ordenar elementos de uma `Collection` em uma ordem diferente da ordem natural da `Collection`.
- O método `reverseOrder` devolve um objeto `Comparator` que representa a ordem inversa para uma coleção.
- O algoritmo `shuffle` ordena aleatoriamente os elementos de uma `List`.
- O algoritmo `binarySearch` localiza um `Object` em uma `List`.
- O `Set` é uma `Collection` que não contém elementos duplicados. A estrutura de coleções contém duas implementações de `Set` – `HashSet` e `TreeSet`. `HashSet` armazena seus elementos em uma tabela de hash; `TreeSet` armazena seus elementos em uma árvore.
- A interface `SortedSet` estende `Set` e mantém seus elementos em ordem de classificação. A classe `TreeSet` implementa `SortedSet`.
- O método `headSet` obtém um subconjunto de um `TreeSet` menor que um elemento especificado. Quaisquer alterações feitas ao subconjunto são feitas em `TreeSet`. O método `tailSet` obtém um subconjunto maior que ou igual a um elemento especificado. Quaisquer alterações feitas na visão através `tailSet` são feitas no `TreeSet`.
- `Maps` mapeiam chaves para valores e não podem conter chaves duplicadas. `Maps` diferem de `Sets` pois eles contêm a chave e o valor, ao passo que `Sets` contêm somente a chave. As classes `HashMap` e `TreeMap` implementam a interface `Map`. `HashMaps` armazenam elementos em uma `Hashtable` e `TreeMaps` armazenam elementos em uma árvore.
- A interface `SortedMap` estende `Map` e mantém seus elementos ordenados. A classe `TreeMap` implementa `SortedMap`.
- As coleções predefinidas são não-sincronizadas. O acesso simultâneo a uma `Collection` por *threads* independentes poderia causar resultados indeterminados. Para evitar isso, são utilizados empacotadores de sincronização para envolver classes que talvez sejam acessadas por múltiplas *threads*.
- A API `Collections` fornece um conjunto de métodos `public static` para converter coleções em versões não-modificáveis. Os empacotadores não-modificáveis disparam `UnsupportedOperationExceptions` se forem feitas tentativas de modificar a coleção.
- A estrutura de coleções fornece várias implementações abstratas (isto é, “esqueletos” de implementações de interfaces de coleção às quais o programador pode “dar corpo” rapidamente, para completar implementações personalizadas).

Terminologia

<i>algoritmo binarySearch</i>	<i>excluir um elemento de uma coleção</i>
<i>algoritmo copy</i>	<i>fila com dupla extremidade (deque)</i>
<i>algoritmo fill</i>	<i>implementação de tabela de hash</i>
<i>algoritmo max</i>	<i>inserir um elemento em uma coleção</i>
<i>algoritmo min</i>	<i>interface</i>
<i>algoritmo reverse</i>	<i>interface Collection</i>
<i>algoritmo shuffle</i>	<i>interface de coleção Map</i>
<i>algoritmo sort</i>	<i>interface de coleção SortedMap</i>
<i>algoritmos</i>	<i>interface de coleção SortedSet</i>
<i>ArrayList</i>	<i>interface Enumeration</i>
<i>arrays</i>	<i>interface Iterator</i>
<i>arrays como coleções</i>	<i>interface List</i>
<i>Arrays.asList</i>	<i>interface Set</i>
<i>chave</i>	<i>iterador</i>
<i>classe AbstractCollection</i>	<i>iterador bidirecional</i>
<i>classe AbstractList</i>	<i>ListIterator</i>
<i>classe AbstractMap</i>	<i>mapa</i>
<i>classe AbstractSequentialList</i>	<i>mapeamento um para um</i>
<i>classe AbstractSet</i>	<i>mapeamentos</i>
<i>classe Collections</i>	<i>mapeamentos como coleções</i>
<i>classe empacotadora</i>	<i>mapeando chaves para valores</i>
<i>classe HashMap</i>	<i>método add</i>
<i>classe HashSet</i>	<i>método addFirst</i>
<i>classe Hashtable</i>	<i>método addLast</i>
<i>classe LinkedList</i>	<i>método clear</i>
<i>classe TreeMap</i>	<i>método hasNext</i>
<i>classe TreeSet</i>	<i>método isEmpty</i>
<i>classe Vector</i>	<i>método next</i>
<i>classes de implementação</i>	<i>método reverseOrder</i>
<i>classificação estável</i>	<i>método size</i>
<i>coleção ordenada</i>	<i>métodos de visão de intervalo</i>
<i>coleções</i>	<i>objeto Comparator</i>
<i>coleções colocadas em arrays</i>	<i>ordem lexicográfica</i>
<i>coleções modificáveis</i>	<i>ordenação</i>
<i>coleções não-modificáveis</i>	<i>ordenação natural</i>
<i>deque</i>	<i>ordenar uma List</i>
<i>elementos duplicados</i>	<i>sequência</i>
<i>empacotadores de sincronização</i>	<i>visão</i>
<i>estrutura de coleções</i>	<i>visualizar um array como uma List</i>
<i>estruturas de dados</i>	

Exercícios de auto-revisão

- 21.1** Preencha as lacunas em cada uma das frases seguintes:
- Objects em uma coleção são chamados de _____.
 - O elemento em uma List pode ser acessado com o _____ do elemento.
 - Às vezes, Lists são chamadas de _____.
 - Você pode utilizar um _____ para criar uma coleção que oferece acesso somente para leitura a outras pessoas enquanto permite acesso para leitura e gravação para você.
 - _____ podem ser utilizadas para criar pilhas, filas, árvores e deque (filas com dupla extremidade).
- 21.2** Determine se cada uma das afirmações seguintes é verdadeira ou falsa. Se for falsa, explique por quê.
- O Set pode conter duplicatas.
 - O Map pode conter chaves duplicadas.

- c) A `LinkedList` pode conter duplicatas.
- d) `Collections` é uma interface.
- e) Os `Iterators` podem remover elementos, mas as `Enumerations` não podem.

Respostas aos exercícios de auto-revisão

21.1 a) elementos. b) índice. c) seqüências. d) empacotador não-modificável. e) `LinkedLists`.

21.2 a) Falsa. O `Set` não pode conter valores duplicados.

b) Falsa. O `Map` não pode conter chaves duplicadas.

c) Verdadeira.

d) Falsa. `Collections` é uma classe e `Collection` é uma interface.

e) Verdadeira.

Exercícios

21.3 Defina cada um dos seguintes termos:

- a) `Collection`
- b) `Collections`
- c) `Comparator`
- d) `List`

21.4 Responda resumidamente às seguintes perguntas:

a) Qual é a principal diferença entre um `Set` e um `Map`?

b) Um array bidimensional pode ser passado para o método `asList` de `Arrays`? Caso possa, como um elemento individual seria acessado?

c) O que você deve fazer primeiro antes de adicionar um tipo primitivo de dados (por exemplo, `double`) a uma coleção?

21.5 Explique resumidamente a operação principal de cada um dos seguintes métodos relacionados com `Iterator`:

- a) `iterator`
- b) `hasNext`
- c) `next`

21.6 Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.

a) Os elementos em uma `Collection` devem ser ordenados em ordem crescente antes de se realizar uma `binarySearch`.

b) O método `first` obtém o primeiro elemento em uma `TreeSet`.

c) A `List` criada com `Arrays.asList` é redimensionável.

d) A classe `Arrays` fornece o método estático `sort` para ordenar elementos do `array`.

21.7 Reescreva o método `printList` da Fig. 21.4 para utilizar um `ListIterator`.

21.8 Reescreva as linhas 16 a 23 da Fig. 21.4 para que fiquem mais concisas, utilizando o método `asList` e o construtor `LinkedList` que recebe um argumento `Collection`.

21.9 Escreva um programa que lê uma série de nomes e os armazena em uma `LinkedList`. Não armazene nomes duplicados. Permita que o usuário procure um nome.

21.10 Modifique o programa da Fig. 21.13 para contar o número de ocorrências de todas as letras (por exemplo, cinco ocorrências de “o” no exemplo). Exiba os resultados.

21.11 Escreva um programa que determina e imprime o número de palavras duplicadas em uma frase. Trate da mesma maneira letras minúsculas e letras maiúsculas. Ignore a pontuação.

21.12 Reescreva a solução do Exercício 19.8 para utilizar uma coleção `LinkedList`.

21.13 Reescreva a solução do Exercício 19.9 para utilizar uma coleção `LinkedList`.

21.14 Escreva um programa que recebe um número inteiro digitado pelo usuário e determina se o número é primo. Se o número for primo, adicione-o à `JTextArea`. Se o número não for primo, exiba os fatores primos do número em um `JLabel`. Lembre-se de que os fatores primos de um número primo são apenas 1 e o próprio número primo. Cada número que não é primo tem uma fatoração única em primos. Por exemplo, considere o número 54. Os fatores de 54 são 2, 3, 3 e 3. Quando os valores são multiplicados, o resultado é 54. Para o número 54, os fatores primos mostrados na saída devem ser 2 e 3. Utilize `Sets` como parte de sua solução.

21.15 Reescreva a solução do Exercício 19.21 para utilizar uma `LinkedList`.

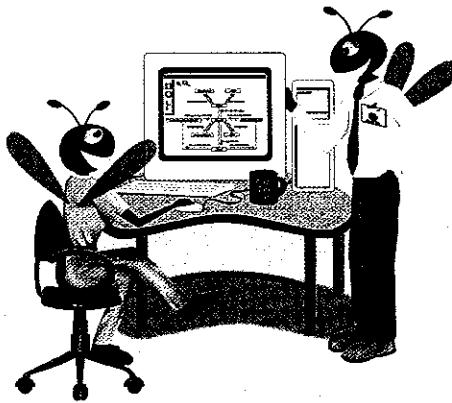
21.16 Escreva um programa que separa em *tokens* (utilizando a classe `StreamTokenizer`) uma linha de texto digitada pelo usuário e coloca cada *token* em uma árvore. Imprima os elementos da árvore ordenada.

Java Media Framework e Java Sound (no CD)

Objetivos

- Entender para que serve o Java Media Framework (JMF).
- Entender para que serve a API Java Sound.
- Ser capaz de reproduzir mídia de áudio e vídeo com JMF.
- Ser capaz de transmitir fluxos de mídia através de uma rede.
- Ser capaz de capturar, formatar e salvar mídia.
- Ser capaz de reproduzir sons com a API Java Sound.
- Ser capaz de reproduzir, gravar e sintetizar MIDI com a API Java Sound.

O Capítulo 22 está no CD que acompanha este livro, em formato PDF do Adobe® Acrobat® e pode ser impresso. O capítulo inclui as páginas 1112 a 1205. Além de apresentar as APIs Java Media Framework e Java Sound, este capítulo dá continuidade ao estudo de caso opcional sobre projeto orientado a objetos de “Pensando em objetos”, apresentando as características de multimídia do simulador de elevador.





Demos de Java

A.1 Introdução¹

Neste apêndice relacionamos as melhores demos de Java que encontramos na Web. Iniciamos nossa jornada em software.dev.earthweb.com/java. Este site é um incrível recurso para Java e possui algumas das melhores demos de Java, incluindo uma enorme compilação de jogos. O código varia do básico ao complexo. Muitos dos autores destes jogos e outros recursos forneceram o código-fonte. Esperamos que você aproveite estes sites tanto quanto nós.

A.2 Os sites

softwaredev.earthweb.com/java

O site *Gamelan*, que pertence à *EarthWeb*, tem sido uma maravilhosa fonte de recursos para Java, desde os primórdios da linguagem. Tratava-se originalmente de um grande repositório em Java no qual as pessoas trocavam idéias a respeito de Java e exemplos de programação em Java. Uma de suas vantagens iniciais era o volume de código-fonte disponível para muitas pessoas que estavam aprendendo Java. Agora é um abrangente recurso com referências Java, *downloads* em Java gratuitos, áreas em que você pode consultar especialistas em Java, grupos de discussões sobre Java, um glossário sobre terminologia relacionada a Java, anúncio de eventos relacionados a Java, diretórios com tópicos especiais dirigidos a empresas e centenas de outros recursos Java.

www.jars.com

Outro site da *EarthWeb* Web é *JARS* – originalmente denominado *Java Applet Rating Service*. O site JARS se autodenomina “#1 Java Review Service”. Este site originalmente era um grande repositório de *applets* Java. Ele classificava cada *applet* registrado no site como *top 1%*, *top 5%* e *top 25%*, de modo que você podia ver imediatamente os melhores *applets* na Web. No início do desenvolvimento da linguagem Java, ter seu *applet* classificado aqui era uma boa maneira de demonstrar suas habilidades de programação em Java. JARS é agora um recurso abrangente para os programadores de Java.

www.javashareware.com

Java Shareware contém centenas de aplicativos Java, *applets*, classes e outros recursos para Java. O site inclui um grande número de *links* para outros sites para os desenvolvedores de Java.

¹ Existem muitos sites da Web relacionados a Java que cobrem tópicos mais avançados de Java, como *servlets*, Java Server Pages (JSP), Enterprise Java Beans (EJB), bancos de dados, Java 2 Enterprise Edition (J2EE), Java 2 Micro Edition (J2ME), segurança, XML e muitos outros. Estes sites são fornecidos em nosso livro *Advanced Java 2 Platform How to Program*.

javaboutique.internet.com

Java Boutique é um grande recurso para qualquer programador Java. Oferece *applets*, aplicativos, fóruns, tutoriais, revisões, glossários e outros.

www.thejmaker.com

O *J Maker* possui inúmeras demos Java, incluindo jogos, menus e efeitos visuais animados. Muitos dos programas neste site são aplaudidos mundialmente e alguns receberam prêmios.

www.demicron.com/gallery/photoalbum2/index.shtml

O *PhotoAlbum II* possui alguns dos melhores efeitos visuais que encontramos na Web. Os efeitos incluem um efeito de líquido, de dobrar uma imagem como um avião de papel e outros.

www.blaupunkt.de/simulations/svdef_en.html

O *Sevilla RDM 168* é uma simulação, construída com Java, de um rádio e CD de um carro. Você pode sintonizar diversas estações de rádio *on-line* ou CD, ajustar o volume, acertar o relógio, etc.

www.frontier.net/~imaging/play_a_piano.html

Play A Piano é um *applet* Java que permite tocar piano ou olhar as ondas de som e ouvir o piano tocar sozinho.

www.gamesdomain.co.uk/GamesArena/goldmine

Goldmine é um jogo divertido que usa animação simples.

www.javaonthebrain.com

Oferece muitos jogos novos e originais escritos em Java para a Web. É fornecido o código-fonte de muitos dos programas.

www.javagamepark.com

Se você está procurando jogos, o site *Java Game Park* tem um monte de jogos. Todos eles são escritos em Java. O código-fonte é fornecido em alguns casos.

teamball.sdsu.edu/~boyns/java/

Este site possui jogos interessantes escritos em Java e inclui o código-fonte de muitos deles.

www.cruzio.com/~sabweb/arcade/index.html

SABames Arcade é uma outra fonte de vídeo games Java. O código-fonte é incluído em muitos dos jogos. Não perca o jogo de boliche SabBowl.

dogfeathers.com/java/hyprcube.html

Stereoscopic Animated Hypercube. Se por acaso você possui um dos antigos óculos 3D azul e vermelho, confira este site. O programador conseguiu criar imagens 3D com o Java. Não é realmente uma imagem complicada, somente alguns cubos; no entanto, a idéia de criar imagens que saltam fora de sua tela é um grande conceito!

www.npac.syr.edu/projects/vishuman/VisibleHuman.html

Este site já ganhou vários prêmios. Você pode ver cortes transversais do corpo humano.

www-groups.dcs.st-and.ac.uk/~history/Java/

Famous Curves Applet Index. Fornece gráficos de curvas complexas. Permite que o usuário altere os parâmetros das equações que calculam as curvas.



Recursos para Java

B.1 Recursos¹

java.sun.com

O site Java Web da Sun Microsystems, Inc. é uma parada obrigatória para quem procura na Web informações sobre Java. Vá a este site para baixar o Java 2 Software Development Kit. O site oferece notícias, informações, suporte *on-line*, exemplos de código, etc.

java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

O site revisa as convenções de código da Sun Microsystems, Inc. para a linguagem de programação Java.

www.softwaredev.earthweb.com

Fornece a grande variedade de informações sobre Java e outros tópicos relacionados à Internet. A página do diretório Java contém *links* para milhares de *applets* Java e outros recursos para Java.

softwaredev.earthweb.com/java

O site *Gamelan*, que pertence à *EarthWeb*, tem sido uma maravilhosa fonte de recursos para Java, desde os primórdios dessa linguagem. Tratava-se originalmente de um grande repositório Java no qual as pessoas trocavam idéias a respeito de Java e exemplos de programação em Java. Uma de suas vantagens iniciais era o volume de código-fonte disponível para muitas pessoas que estavam aprendendo Java. Agora é um abrangente recurso com referências Java, *downloads* Java gratuitos, áreas em que você pode consultar especialistas em Java, grupos de discussões sobre Java, um glossário sobre terminologia relacionada a Java, anúncio de eventos relacionados a Java, diretórios com tópicos especiais dirigidos a empresas e centenas de outros recursos Java.

www.jars.com

Outro site da EarthWeb Web é JARS – originalmente denominado Java Applet Rating Service. O site JARS se autodenomina “#1 Java Review Service”. Este site originalmente era um grande repositório de *applets* Java. Ele classificava cada *applet* registrado no site como *top 1%*, *top 5%* e *top 25%*, de modo que você podia ver imediatamente os melhores *applets* na Web. No início do desenvolvimento da linguagem Java, ter seu *applet* classificado aqui era uma boa maneira de demonstrar sua habilidades de programação em Java. JARS é agora um recurso abrangente para os programadores de Java.

¹ Existem muitos sites da Web relacionados a Java que cobrem tópicos mais avançados de Java, como *servlets*, Java Server Pages (JSP), Enterprise Java Beans (EJB), bancos de dados, Java 2 Enterprise Edition (J2EE), Java 2 Micro Edition (J2ME), segurança, XML e muitos outros. Estes sites são fornecidos em nosso livro *Advanced Java 2 Platform How to Program*.

developer.java.sun.com/developer

No site Java Web da Sun Microsystems, visite a *Java Developer Connection*. Este site inclui suporte técnico, fóruns de discussão, cursos de treinamento *on-line*, artigos técnicos, recursos, divulgação de novos recursos para Java, acesso antecipado a novas tecnologias Java e *links* para outros sites Java importantes. Mesmo sendo livre, você deve registrar-se para usar o site.

www.acme.com/java

Esta página possui diversos *applets* Java animados, juntamente com seu código-fonte. Este site é um excelente recurso para informações sobre Java. A página fornece *software*, notas e uma lista de *hyperlinks* para outros recursos. No link softwares, você encontrará *applets* animados, classes utilitárias e aplicativos.

www.javaworld.com/index.html

A revista *on-line* JavaWorld fornece uma coleção de artigos sobre Java, dicas, novidades e discussões. Uma área de perguntas abrange problemas gerais e específicos enfrentados pelos programadores.

www.nikos.com/javatoys

O site da Web Java Toys inclui *links* para as últimas novidades sobre Java, Java User Groups (JUGs), FAQs, ferramentas, listas de endereços relacionados a Java, livros e artigos.

<http://www.java-zone.com/>

O site Development Exchange Java Zone inclui grupos de discussão sobre Java, uma seção “Pergunte ao Java Pro” e algumas novidades recentes sobre Java.

www.ibiblio.org/javafaq

Fornece as últimas novidades sobre Java. Também possui alguns recursos úteis, incluindo uma lista Java FAQ, um tutorial chamado Brewing Java, Java User Groups, Java Links, uma lista de livros sobre Java, feiras comerciais sobre Java, treinamento em Java e exercícios.

dir.yahoo.com/Computers_and_Internet/Programming_Languages/Java

Yahoo, um popular mecanismo de busca na World Wide Web, fornece muitos recursos sobre Java. Você pode iniciar a pesquisa usando palavras-chave ou explorar categorias relacionadas no site, incluindo jogos, concursos, eventos, tutoriais e documentação, listas de endereço, segurança e outros.

www-106.ibm.com/developerworks/java

O site IBM Developers Java Technology Zone relaciona novidades recentes, ferramentas, código, estudos de caso e eventos relacionados à IBM e Java.

B.2 Produtos

java.sun.com/products

Baixe o Java 2 SDK e outros produtos Java da página Java Products da Sun Microsystems.

www.sun.com/forte/ffj/index.cgi

O IDE NetBeans é ambiente visual de desenvolvimento de programas adaptável e independente de plataforma.

www.borland.com/jbuilder

A homepage do Borland JBuilder IDE contém novidades, informações sobre produtos e suporte a clientes.

www.towerj.com

Neste site você encontra informações sobre como melhorar o desempenho de aplicativos de servidores em Java, assim como cópias de avaliação gratuita de compiladores Java para código nativo.

www.symantec.com/domain/cafe

Visite o site Symantec para obter informações sobre seu Visual Café Integrated Development Environment.

www-4.ibm.com/software/ad/vajava

Baixe e leia o ambiente de desenvolvimento Visual Age para Java da IBM.

www.metrowerks.com

O IDE Metrowerks CodeWarrior oferece suporte para algumas linguagens de programação, incluindo Java.

B.3 FAQs

www.ibiblio.org/javafaq/javafaq.html

Este é um amplo recurso tanto para a linguagem básica Java como para tópicos mais avançados em programação Java. A seção 6, *Language Issues*, e a seção 11, *Common Errors and Problems*, podem ser particularmente úteis, uma vez que esclarecem algumas situações que freqüentemente não são bem explicadas.

www.afu.com/javafaq.html

Este é outro FAQ que cobre uma grande variedade de tópicos sobre Java. Inclui alguns bons exemplos de código e dicas para compilar e executar projetos.

www.nikos.com/javatoys/faqs.html

O site da Web *Java Toys* inclui *links* para FAQs sobre uma grande variedade de tópicos relacionados a Java.

www.jguru.com/faq/index.jsp

Esta é uma compilação cuidadosa de FAQs sobre Java e assuntos correlatos. As perguntas podem ser lidas em ordem ou pesquisadas por assunto.

B.4 Tutoriais

Diversos tutoriais encontram-se nos *sites* relacionados na seção sobre Recursos.

java.sun.com/docs/books/tutorial/

O site *Java Tutorial* possui diversos tutoriais, incluindo seções sobre JavaBeans, JDBC, RMI, Servlets, Collections e Java Native Interface.

www.ibiblio.org/javafaq/

Este site fornece novidades recentes sobre Java. Também possui recursos úteis sobre Java incluindo a Java FAQ List, um tutorial chamado Brewing Java, Java User Groups, Java Links, Java Book List, Java Conferences, Java Course Notes e Java Seminar Slides.

B.5 Revistas

www.javaworld.com

A revista *on-line JavaWorld* é um excelente recurso para se obter informações atuais sobre Java. Você encontrará recortes de notícias, informações sobre conferências e *links* para *sites* relacionados a Java.

www.sys-con.com/java

Capture as últimas novidades sobre Java, no site do *Java Developer's Journal*. Esta revista é um dos recursos mais importantes para novidades sobre Java.

www.javareport.com

O *Java Report* é um grande recurso para os desenvolvedores Java. Você encontrará as últimas novidades das empresas, exemplos de código, relação de eventos, produtos e empregos.

www.javapro.com

O *JAVAPro* é um excelente recurso para os desenvolvedores Java, com os mais recentes artigos técnicos.

B.6 Applets Java

java.sun.com/applets/index.html

Esta página contém diversos recursos sobre *applets* Java, incluindo *applets* gratuitos que você pode usar em seu próprio site da World Wide Web, os *applets* de demonstração de J2SDK e diversos outros *applets* (muitos deles podem ser baixados e usados em seu próprio computador). Também existe uma seção intitulada “*Applets at Work*” na qual você pode ler a respeito do uso de *applets* nas empresas.

developer.java.sun.com/developer/

No site Java Web da Sun Microsystems, visite a *Java Developer Connection*. Este site livre tem perto de um milhão de membros. Inclui suporte técnico, fóruns de discussão, cursos de treinamento *on-line*, artigos técnicos, recursos, divulgação de novos recursos de Java, acesso antecipado a novas tecnologias Java e *links* para outros sites Java importantes. Mesmo sendo um site livre, você deve registrar-se para usá-lo.

www.gamelan.com

O site *Gamelan*, que pertence à *EarthWeb*, tem sido uma maravilhosa fonte de recursos para Java, desde os primórdios dessa linguagem. Tratava-se originalmente de um grande repositório Java no qual as pessoas trocavam idéias a respeito de Java e exemplos de programação em Java. Uma de suas vantagens iniciais era o volume de código-fonte disponível para muitas pessoas que estavam aprendendo Java. Agora é um abrangente recurso com referências Java, *downloads* Java gratuitos, áreas em que você pode consultar especialistas em Java, grupos de discussões sobre Java, um glossário sobre terminologia relacionada a Java, anúncio de eventos relacionados a Java, diretórios com tópicos especiais dirigidos a empresas e centenas de outros recursos Java.

www.jars.com

Outro site da *EarthWeb* Web é JARS – originalmente denominado *Java Applet Rating Service*. O site JARS se autodenomina “#1 Java Review Service”. Este site originalmente era um grande repositório de *applets* Java. Ele classificava cada *applet* registrado no site como *top 1%*, *top 5%* e *top 25%*, de modo que você podia ver imediatamente os melhores *applets* na Web. No início do desenvolvimento da linguagem Java, ter seu *applet* classificado aqui era uma boa maneira de demonstrar sua habilidades de programação em Java. JARS é agora um recurso abrangente para os programadores Java.

B.7 Multimídia

java.sun.com/products/java-media/jmf

Esta é a *homepage* do *Java Media Framework*. Aqui você encontra as mais recentes implementações da Sun de JMF (veja o Capítulo 22). O site também contém a documentação do JMF.

www.nasa.gov/gallery/index.html

A *multimedia gallery* da NASA contém uma grande variedade de imagens, clipes de áudio e vídeo que você pode baixar e usar para testar seus programas de multimídia Java.

www.sunsite.sut.ac.jp/multimed

A *Sunsite Japan Multimedia Collection* também fornece uma grande variedade de imagens, clipes de áudio e vídeo que você pode baixar e usar para fins educacionais.

www.anbg.gov.au/anbg/index.html

O site da Web *Australian National Botanic Gardens* fornece *links* para sons de muitos animais.

java.sun.com/products/java-media/jmf/index.html

Este site fornece um guia *on-line* baseado em HTML sobre a API *Java Media Framework*.

B.8 Grupos de notícias

Os grupos de notícias são fóruns na Internet nos quais as pessoas podem colocar perguntas, respostas, dicas e esclarecimentos para outros usuários. Podem ser um recurso valioso para quem está aprendendo Java ou para quem tem perguntas sobre tópicos específicos. Ao fazer suas próprias perguntas em um grupo de notícias, forneça detalhes específicos do problema que você está tentando resolver (como um problema existente num programa que você está escrevendo). Isto irá permitir que as outras pessoas que lerem o grupo de notícias entendam sua colocação e (espera-se) lhe dar uma resposta. Certifique-se de especificar um título de assunto que claramente indique seu problema. Se você estiver respondendo a uma pergunta de outra pessoa, verifique sua resposta antes de enviá-la, para assegurar que está correta. Os grupos de notícias não devem ser usados para promover produtos ou serviços, nem as informações de contato (como endereços de *e-mail* de outros usuários do grupo de notícias) devem ser usadas para propósitos não-relacionados. Em geral, estes não são fóruns para bate-papo, de modo que as publicações devem ser corteses e ir direto ao assunto.

Normalmente, o *software* para leitura de grupos de notícias é necessário para você interagir com um grupo de notícias. Este tipo de *software* é fornecido como parte do *Netscape Navigator* e do *Microsoft Outlook Express* e existem muitos outros programas para grupos de notícias disponíveis. Você também pode acessar os grupos de notícias através da Web. Se você ainda não usar os grupos de notícias, experimente o *site do Google groups.google.com*

Digite o nome do grupo que você gostaria de ver (você também pode pesquisar os grupos de notícias) e você receberá uma lista dos tópicos e perguntas atuais daquele grupo de notícias. A seguir estão relacionados alguns grupos de notícias sobre Java que você pode achar úteis.

news://comp.lang.java.advocacy/

Este grupo é um centro ativo de discussões sobre a cultura Java atual, incluindo os méritos de diferentes linguagens de programação.

news://comp.lang.java.announce/

Faz a divulgação das principais extensões de Java, novas bibliotecas de classes e conferências.

news://comp.lang.java.api/

Contém perguntas sobre *bugs*, erros de compilação, especificações de Java e quais classes são mais apropriadas para diferentes situações.

news://comp.lang.java.gui/

Responde aos problemas encontrados ao se trabalhar com interfaces gráficas com usuário em Java. Se você tiver problemas com um componente particular, leiaute ou evento, este pode ser um bom lugar para iniciar.

news://comp.lang.java.help/

Trata-se de um grupo particularmente ativo. Aborda muitas questões sobre linguagem e ambiente. Entre as colocações incluem-se solicitações de classes ou algoritmos para resolver um problema específico.

news://comp.lang.java.machine/

Para as pessoas interessadas no funcionamento interno de Java, este grupo se concentra na *Java Virtual Machine*.

news://comp.lang.java.misc/

Este grupo contém de tudo, de ofertas de emprego a perguntas sobre a documentação Java, e se destina a colocações que não se enquadram em outras categorias de grupo de notícias sobre Java.

news://comp.lang.java.programmer/

Outro fórum extremamente ativo, que trata de uma variedade de perguntas. As colocações tendem a ser orientadas a projetos e a preocupações sobre estilo geral de programação e estrutura.

news://comp.lang.java.softwaretools/

Este grupo de notícias centra-se em *softwares* escritos em Java, seus usos, suas falhas e possíveis modificações. Algumas perguntas sobre escrita de *softwares* eficiente também são encontradas aqui.

news://comp.lang.java.tech/

Trata-se de um grupo de notícias que se dedica a aspectos técnicos de Java e seu funcionamento interno.



Tabela de precedência de operadores

Os operadores são mostrados em ordem decrescente de precedência, de cima para baixo.

Operador	Tipo	Associatividade
()	parênteses	da esquerda para a direita
[]	subscrito de <i>array</i>	
.	seleção de membro	
++	pós-incremento unário	da direita para a esquerda
--	pós-decremento unário	
++	pré-incremento unário	da direita para a esquerda
--	pré-decremento unário	
+	mais unário	
-	menos unário	
!	negação lógica unária	
~	complemento unário sobre <i>bits</i>	
(<i>tipo</i>)	coerção unária	
*	multiplicação	da esquerda para a direita
/	divisão	
%	módulo	
+	adição	da esquerda para a direita
-	subtração	
<<	deslocamento de <i>bits</i> para a esquerda	da esquerda para a direita
>>	deslocamento de <i>bits</i> para a direita com extensão de sinal	
>>>	deslocamento de <i>bits</i> para a direita com extensão de zeros	

Fig. C.1 Tabela de precedência de operadores (parte 1 de 2).

Operador	Tipo	Associatividade
Operador	Tipo	Associatividade
<	relacional menor que	da esquerda para a direita
<=	relacional menor que ou igual a	
>	relacional maior que	
>=	relacional maior que ou igual a	
instanceof	comparação de tipo	
==	relacional é igual a	da esquerda para a direita
!=	relacional não é igual a	
&	E sobre <i>bits</i>	da esquerda para a direita
^	OU exclusivo sobre <i>bits</i> OU exclusivo lógico booleano	da esquerda para a direita
	OU inclusivo sobre <i>bits</i> OU inclusivo lógico booleano	da esquerda para a direita
&&	E lógico	da esquerda para a direita
	OU lógico	da esquerda para a direita
? :	ternário condicional	da direita para a esquerda
=	atribuição	da direita para a esquerda
+=	atribuição de adição	
-=	atribuição de subtração	
*=	atribuição de multiplicação	
/=	atribuição de divisão	
%=	atribuição de módulo	
&=	atribuição E sobre <i>bits</i>	
^=	atribuição OU exclusiva sobre <i>bits</i>	
=	atribuição OU inclusiva sobre <i>bits</i>	
<<=	atribuição de deslocamento para a esquerda	
>>=	atribuição de deslocamento de <i>bits</i> para a direita com extensão de sinal	
>>>=	atribuição de deslocamento de <i>bits</i> para a direita com extensão de zeros	

Fig. C.1 Tabela de precedência de operadores (parte 2 de 2).



Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	í	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

Fig. D.1 Conjunto de caracteres ASCII.

Os dígitos à esquerda da tabela são os dígitos mais à esquerda do equivalente em decimal (0-127) do código do caractere e os dígitos na parte superior da tabela são os dígitos à direita (unidade) do código de caractere. Por exemplo, o código de caractere para “F” é 70 e o código de caractere para “&” é 38.

A maioria dos usuários deste livro está interessada no conjunto de caracteres ASCII utilizado para representar caracteres da língua inglesa em muitos computadores. O conjunto de caracteres ASCII é um subconjunto do conjunto de caracteres Unicode utilizado por Java para representar caracteres da maioria dos idiomas do mundo. Para obter mais informações sobre o conjunto de caracteres Unicode, consulte o Apêndice K.

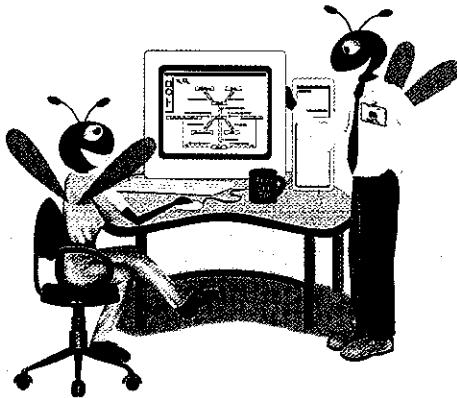
E

Sistemas de numeração (no CD)

Objetivos

- Entender conceitos básicos de sistemas de numeração, como base, valor posicional e valor de símbolo.
- Entender como trabalhar com números representados nos sistemas de numeração binária, octal e hexadecimal.
- Ser capaz de abreviar números binários como números octais ou números hexadecimais.
- Ser capaz de converter números octais e números hexadecimais em números binários.
- Ser capaz de fazer conversão nos dois sentidos entre números decimais e seus equivalentes em binário, octal e hexadecimal.
- Entender a aritmética binária e como os números binários negativos são representados com notação de complemento de dois.

O Apêndice E está no CD que acompanha este livro em formato PDF do Adobe® Acrobat® e pode ser impresso. O apêndice inclui as páginas 1216 a 1227.



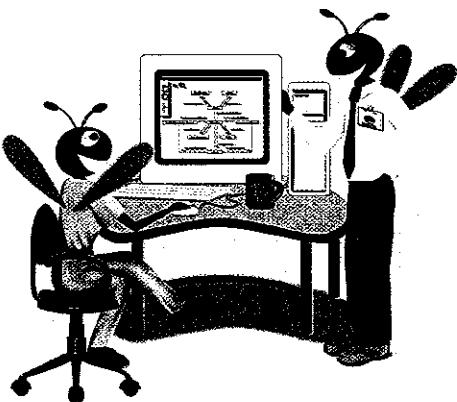
3

Criando documentação em HTML com **javadoc** (no CD)

Objetivos

- Apresentar a ferramenta **javadoc** do J2SDK.
- Apresentar comentários de documentação
- Entender as marcas de **javadoc**.
- Ser capaz de gerar documentação de API HTML com **javadoc**.
- Entender os arquivos de documentação gerados por **javadoc**.

O Apêndice F está no CD que acompanha este livro em formato PDF do Adobe® Acrobat® e pode ser impresso. O apêndice inclui as páginas 1229 a 1241.





Eventos e interfaces *listener* do elevador (no CD)

O Apêndice G está no CD que acompanha este livro em formato PDF do Adobe® Acrobat® e pode ser impresso. O apêndice inclui as páginas 1242 a 1249. Este apêndice é o primeiro dos três apêndices incluídos no CD que dão continuidade ao nosso estudo de caso opcional “Pensando em objetos”. O apêndice apresenta as classes de eventos e as interfaces *listener* para a implementação do simulador de elevador orientado a objetos.



Modelo do elevador (no CD)

O Apêndice H está no CD que acompanha este livro em formato PDF do Adobe® Acrobat® e pode ser impresso. O apêndice inclui as páginas 1251 a 1289. Este apêndice é o segundo dos três apêndices incluídos no CD que dão continuidade ao nosso estudo de caso opcional “Pensando em objetos”. O apêndice apresenta o código para todas as 10 classes que representam, em conjunto, o modelo e conclui a discussão do modelo do elevador. Discutimos cada classe separadamente e em detalhes.

7

Visão do elevador (no CD)

O Apêndice I está no CD que acompanha este livro em formato PDF do Adobe® Acrobat® e pode ser impresso. O apêndice inclui as páginas 1290 a 1312. Este apêndice completa nosso estudo de caso opcional “Pensando em objetos”. Ele contém a implementação para a classe **ElevatorView** – a maior classe na simulação. Dividimos a discussão de **ElevatorView** em cinco tópicos – *objetos da classe, constantes da classe, construtor da classe, tratamento de eventos e Revisitando diagramas de componentes*.

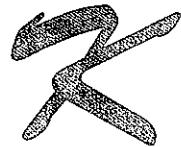
J

Oportunidades para a carreira profissional (no CD)

- Explorar os diversos serviços *on-line* para a carreira profissional.
- Examinar as vantagens e desvantagens de anunciar e procurar empregos *on-line*.
- Comentar os principais *sites* da Web com serviços *on-line* para a carreira profissional disponíveis para quem está procurando trabalho.
- Explorar os diversos serviços *on-line* disponíveis para os empregadores que desejam montar sua equipe de trabalho.

O Apêndice J está no CD que acompanha este livro em formato PDF do Adobe® Acrobat® e pode ser impresso. O apêndice inclui as páginas 1313 a 1335.

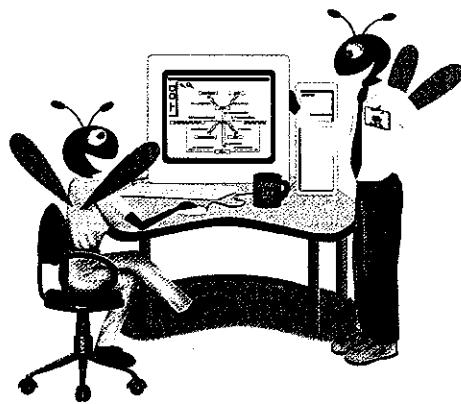




Unicode® (no CD)

- Familiarizar-se com o Unicode.
- Discutir a missão do Consórcio Unicode.
- Discutir as bases de projeto de Unicode.
- Entender as três formas de codificação Unicode: UTF-8, UTF-16 e UTF-32.
- Apresentar caracteres e hieróglifos.
- Discutir as vantagens e as desvantagens de usar Unicode.
- Fornecer uma breve visão do *site* do Consórcio Unicode

O Apêndice K está no CD que acompanha este livro em formato PDF do Adobe® Acrobat® e pode ser impresso. O apêndice inclui as páginas 1336 a 1345.



Bibliografia

Recursos da Sun Microsystems

- Block, J., "Tutorial: Collections", java.sun.com/docs/books/tutorial/collections/index.html, 1999
- Fisher, M., "The JDBC Tutorial and Reference: Second Edition Chapter 3 Excerpt" developer.java.sun.com/developer/Books/JDBCTutorial/index.html, 1999
- Gosling, J., e Gilton, H., "The Java Language Environment: a White Paper", java.sun.com/docs/white/langenv/, 1996
- Kluyt, O., "JavaBeans: Unlocking the BeanContext API", developer.java.sun.com/developer/technicalArticles/Beans/BeanContext/index.html
- Meloan, M.D. "The Science Of Java Sound" developer.java.sun.com/developer/technicalArticles/Media/JavaSoundAPI/index.html
- Papageorge, J., "Getting Started with JDBC" developer.java.sun.com/developer/technicalArticles/Interviews/StartJDBC/index.html, 1999
- Sundsted, T., "XML and JAVA Tackle Entries Application Integration" developer.java.sun.com/developer/technicalArticles/Networking/XMLAndJava/index.html, 1999
- Sun Microsystems, "JavaBeans Specifications and Tutorials", java.sun.com/beans/docs/spec.html
- Sun Microsystems, "Enterprise JavaBeans Specifications and Tutorials", java.sun.com/products/ejb/newspec.html
- Sun Microsystems, "Java 2D API Specifications and Tutorials", java.sun.com/products/java-media/2D/forDevelopers/2Dapi/index.html
- Sun Microsystems, "RMI Specifications and Tutorials", java.sun.com/products/jdk/1.2/docs/guide/rmi/
- Sun Microsystems, "Java Servlets API Specifications and Tutorials", java.sun.com/products/servlet/index.html, 1999
- Sun Microsystems, "Java Foundation Classes: White Paper", java.sun.com/marketing/collateral/foundation_classes.html, 1999
- Sun Microsystems, "The Collections Framework Overview", java.sun.com/products/jdk/1.2/docs/guide/collections/overview.html, 1999
- Sun Microsystems, "Java Database Connectivity", java.sun.com/marketing/collateral/jdbc_ds.html

- Sun Microsystems, "Getting Started with Swing", java.sun.com/docs/books/tutorial/uiswing/start/index.html
- Sun Microsystems "Swing Features and Concepts", java.sun.com/docs/books/tutorial/uiswing/overview/index.html
- Sun Microsystems "Using Swing Components", java.sun.com/docs/books/tutorial/uiswing/components/index.html
- Sun Microsystems "Converting to Swing", java.sun.com/docs/books/tutorial/uiswing/converting/index.html
- Sun Microsystems "Laying Out Components within a Container", java.sun.com/docs/books/tutorial/uiswing/layout/index.html
- Zucowski, J., "Mastering Java 2 Chapter 16: Transferring Data" developer.java.sun.com/developer/Books/MasteringJava/Ch16/index.html, 1999
- Zucowski, J., "Mastering Java 2 Chapter 17: Java Collections" developer.java.sun.com/developer/Books/MasteringJava/Ch17/index.html, 1999

Outros Recursos

- Arnold, K., J. Gosling, e D. Holmes. *The Java™ Programming Language: Third Edition*. Reading, MA: Addison-Wesley, 2000.
- Barker, J. *Beginning Java Objects: From Concepts to Code*. Birmingham: Wrox Press, 2000.
- Barnebee, J., "Java NT Services: Migrating you Java Server from Unix to an NT Boot-Time Environment", *Java Developer's Journal*, February 1999, pp. 54-56.
- Bell, D. e Parr, M., *Java for Students Second Edition*, London, UK: Prentice Hall Europe, 1999.
- Bennet, S., J. Skelton, e K. Lunn. *Schaum's Outline of UML*. New York, NY: McGraw Hill, 2001.
- Berg, C.J., *Advanced Java Development for Enterprise Applications*, Upper Saddle River, NJ: Prentice Hall, 1998.
- Berg, D. e Fritzinger, J.S., *Advanced Techniques for Java Developers*, Nova York, NY: John Wiley & Sons, Inc. 1997.
- Bloch, J. *Effective Java™ Programming Language Guide*. Reading, MA: Addison-Wesley, 2001.
- Booch, G. *Object-Oriented Analysis and Design with Applications*. Reading, MA: Addison-Wesley, 1994.
- Booch, G., J. Rumbaugh, e I. Jacobson. *The Complete UML Training Course*. Reading, MA: Addison-Wesley, 2000.
- Brodsky, S. e T. Grose. *Mastering XMI: Java Programming XMI, XML, and UML*. New York, NY: John Wiley & Sons, 2002.
- Brogden, B., *Exam Cram: Java 2 Exam 310-025*, Scottsdale, AZ: The Coriolis Group, 1999.
- Bryson, T., "Exploring the Java 3D API", *Performance Computing*, April 1999, pp. 28-34.
- Callahan, T., "So You Want a Standalone Database for Java", *Java Developer's Journal*, December 1998, pp. 28-36.
- Campione, M., Walrath, K., Huml, A. e o Tutorial Team, *The Java Tutorial Continued: The Rest of the JDK*, Reading, MA: Addison-Wesley, 1999.
- Campione, M., Walrath, K. e Huml, A., *The Java™ Tutorial, Third Edition: A Short Course on the Basics*. Reading, MA: Addison-Wesley, 2000.
- Carey, J., B. Carlson e T. Graser. *San Francisco™ Design Patterns: Blueprint for Business Software*. Reading, MA: Addison-Wesley, 2000.
- Catalano, C., "Java Applets", *ComputerWorld*, May 3, 1999, p. 72.
- Cheesman, J. e J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software (The Component Software Series)*. Reading, MA: Addison-Wesley, 2000.
- Coad, P., Mayfeild, M. e Kern, J., *Java Design: Building Better Apps and Applets: Second Edition*, Upper Saddle River, NJ: Yourdon Press, 1999.

- Coffee, P., "Java, Thin Clients Give Video Provider Tele-'Vision'", *PC Week*, April 12, 1999, p. 55.
- Cooper, J. *Java Design Patterns; A Tutorial*. Reading, MA: Addison-Wesley, 2000.
- Daconta, M. C., E. Monk, J. P. Keller e K. Bohnenberger. *Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs*. New York, NY: John Wiley & Sons, 2000.
- Detlefs, D., "Concurrent Coffee", *Performance Computing*, July 1999, pp.25-29.
- Eckel, B. *Thinking In Java: 2nd Edition*. Upper Saddle River, NJ: Prentice Hall, 2000.
- Flynn, J. e B. Clarke, "The World Wakes up to Java!" *Computer Technology Review*, 1996, pp. 33-37.
- Flanagan, D., *Java Examples: In a Nutshell*, Sebastopol, CA: O'Reilly & Associates, Inc., 2000.
- Fowler, M. e K. Scott. *UML Distilled Second Edition; A Brief Guide to the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 1999.
- Gamma, E., R. Helm, R. Johnson e J. Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- George, J., "Java – Into Its Fourth Year", *Java Developer's Journal*, February 1999, p. 66.
- Gilbert, S. e McCarty, B., *Object-Oriented Design in Java*, Corte Madera, CA: Waite Group Press, 1998.
- Grand, M. *Patterns in Java; A Catalog Reusable Design Patterns Illustrated with UML*. New York, NY: John Wiley & Sons, 1998.
- Haggar, P., "Effective Exception Handling in Java", *Java Report*, April 1999, pp. 55-64.
- Halfhill, T. R. "How to Soup up Java: Part 1", *Byte Magazine*, May 1998, pp. 60-80.
- Haverlock, K. "Object Serialization, Java, and C++", *Dr. Dobb's Journal*, August 1998, pp. 32-34.
- Heller, P. e Roberts, S., *Java 2 Developers Handbook*, Alameda, CA: Sybex, Inc., 1999.
- Hemrajani, A., "Programming with I/O Streams: Part 3", *Java Developer's Journal*, February 1999, pp. 18-22.
- Hof, R. D. e J. Verity, "Scott McNealy's Rising Sun", Cover Story, *Business Week*, January 22, 1996, pp. 66-73.
- Horstmann, C. S. e Cornell, G., *Core Java 2 Volume I: Fundamentals*, Upper Saddle River, NJ: Prentice Hall, 2001.
- Horstmann, C. S. e Cornell, G., *Core Java: Volume II: Advanced Features*, Upper Saddle River, NJ: Prentice Hall, 2001.
- Hunt J., "The Collection API", *Java Report*, April 1999, pp. 17-32.
- Hunter, J. e Kadel, R., "Everywhere You Look: Enterprise Server-Side Java", *Javaworld*, March 27, 1998, *on-line*.
- Larman, C. *Applying UML and Patterns; An Introduction to Object-Oriented Analysis and Design*. Upper Saddle River, NJ: Prentice Hall, 1998.
- Lea, D. *Concurrent Programming in Java™ Second Edition Design Principles and Patterns*. Reading, MA: Addison-Wesley, 2000.
- Lauinger, T., "Object-Oriented Software Development in Java", *Java Report*, February 1999, pp. 59-61.
- Malarvannan, M., "A Multithreaded Server in Java", *Web Techniques*, October 1998, pp. 47-51.
- Maruyama, H.; Tamura, K. e Uramoto Naohiko, *XML and JAVA: Developing Web Applications*, Reading, MA: Addison-Wesley Publishing Company, Inc., 1999.
- Oaks, S., "How Do I Create My Own UI Component?" *Java Report*, March/April 1996, pp. 64, 63.
- Oaks, S., "Two Techniques for Handling Events", *Java Report*, July/August 1996. p. 80.
- Oaks, S. e H. Wong, *Java Threads*, Sebastopol, CA: O'Reilly & Associates, Inc., 1997.
- Page-Jones, M. *Fundamentals of Object-Oriented Design in UML*. Reading, MA: Addison-Wesley, 1999.
- Penker, M. e Hans-Erik E. *Business Modeling with UML: Business Patterns At Work*. New York, NY: John Wiley & Sons, 2000.
- Pratik P. e Moss, K. *Java Database Programming with JDBC: Second Edition*, Scottsdale, AZ: The Coriolis Group, 1997.
- Rinehart, M., *Java Database Development*, Berkeley, CA: Osborn/McGraw-Hill, 1998.
- Roberts, Si., P. Heller, M. Ernest e R. et al. *The Complete Java 2 Certification Study Guide*. Alameda, CA: SYBEX, 2000.

- Rodrigues, L., "On JavaBeans Customization", *Java Developer's Journal*, May 1999, pp.21.
- Rumbaugh, J., I. Jacobson e G. Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.
- Rumbaugh, J., I. Jacobson e G. Booch. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.
- Rumbaugh, J., I. Jacobson e G. Booch. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.
- Scott, K. *UML Explained*. Reading, MA: Addison-Wesley, 2001.
- Shirazi, J. *Java Performance Tuning*. Sebastopol, CA: O'Reilly and Associates, 2000.
- Stevens, P. e R.J. Pooley. *Using UML: Software Engineering with Objects and Components Revised Edition*. Reading, MA: Addison-Wesley, 2000.
- Sun Microsystems Inc. *Java™ Look and Feel Design Guidelines, Second Edition*. Reading, MA: Addison-Wesley, 2001.
- Tan, E. M., "Java-The Software Design", *Java Developer's Journal*, January 1999, pp. 58-59.
- Topley, K. *Core Swing: Advanced Programming*. Upper Saddle River, NJ: Prentice Hall, 2000.
- Venners, B., *Inside the Java Virtual Machine*, Nova York, NY: McGraw-Hill, 1998.
- Vlissides, J. *Pattern Hatching: Design Patterns Applied*. Reading, MA: Addison-Wesley, 1998.
- Walsh, A., and Fronckowiak, J., *Java Bible*, Foster City, CA; IDG Books Worldwide, Inc. 1998.

Índice

Símbolos

!, NÃO lógico 236, 239-240
!=, não igual a 114-116, 236
%, módulo 111-113
&&, E lógico 236, 238-239
&, E lógico booleano 236
(), parênteses 112-113
* 94-95
*, multiplicação 111-113
+++, pré-incremento/pós-incremento 199-200
+, adição 111-113, 377-378, 528-529
+=, operador de atribuição de adição 198-199, 528-529
--, pré-decremento/pós-decremento 199-201
-, subtração 111-113
.java, extensão de nome de arquivo 144
/* */ comentário com múltiplas linhas 94-95
/** */ comentário de documentação de Java 94-95
/, divisão 111-113

@author 1234-1235
@files 1237-1238
@link 1241
@since 1241
@version 1237-1238
[], subscrito de array 352-353
\, caractere separador 880-881
\", sequência de escape aspas 100-101
\, caractere de escape 100-101
\n, sequência de escape nova linha 100-101, 149-150, 224-226
\r, sequência de escape retorno do carro 100-101
\t, sequência de escape tabulação 100-101, 224-226
\, OU exclusivo lógico booleano 236
\, chave à esquerda 96-98
\, OU inclusivo lógico booleano 236
\|, OU lógico 236-238
\, chave à direita 96-98

Numéricos

Jogo do 15 986-987
efeito 3D 815-816

A

aberto, código-fonte 59-60
abertura ({}, chave à esquerda de 283-284
abordagem de blocos de construção para a criação de programas 59-60, 68-69
abortar em uma situação excepcional 741-742
abreviado, nome 151-152
abreviaturas semelhantes às da língua inglesa 56-57
abrir um arquivo 820-821
abs, método de Math 262-263
abstração 66-67, 432-433
Abstract Factory, padrão de projeto 494-495, 956-957
Abstract Windowing Toolkit (AWT) 605-606
Abstract Windowing Toolkit 700
Abstract Windowing Toolkit Event Package 271
abstract, método 457-458, 465-466, 490-494, 611-612, 1105-1106, 1259
abstract, palavra-chave 175-176, 455, 1035, 1045-1046
AbstractButton 617-619, 690-691, 695-696
ação 50-51, 97-98, 173, 246-247, 418-419, 549-550, 551-552
accept 900-901, 903-904, 906, 919-920
AccountRecord.java 827-828
acessar dados compartilhados 782-783, 815
acesso a novas tecnologias de Java 160
acesso aleatório, arquivo de 818, 822-823, 846, 855-856
acesso aleatório, memória de (RAM) 55
acesso direto, aplicativo de 822-823
acesso direto, arquivos 846
acesso protected vs. acesso private em super classes 504-505
acesso protegido, modificador de 415-416
acesso simultâneo a uma Collection por múltiplas threads 1104-1105
acesso, método de 376-377, 381-382, 391
ActionEvent 278-279, 281, 311-312, 613-618, 644-645, 670-671, 687-688, 973
ActionListener 278-280, 473-474, 600-601, 613—618, 664-665, 687-688, 712-713, 747-748
ACTIVATED 897-898, 900-901

- acumulador 366
 Ada Lovelace 61
 Ada, linguagem de programação 61, 769
Adapter, padrão de projeto 494-495, 497-498, 726-727
add, método de *Container* 263-264, 280-281, 606-609
add, método de *JMenuBar* 695-696
add, método de *Vector* 893-894
addActionListener 281, 394-395, 480-483, 614-615, 617-618, 630-631, 644-645
addComponent 716-717, 719-720
addElement 1041-1042
addHyperlinkListener 897-898
addItemListener 620-621
Addition.java 105-106
AdditionApplet.html 151-152
AdditionApplet.java 149-151
addKeyListener 641-642
addListSelectionListener 627-628
addMenu 704-705
addMouseListener 633-634, 675-678, 698-699, 977-978
addMouseMotionListener 633-634, 676-680, 977-978
addPoint 585-587
addSeparator 693-694
addTarget 1142-1143
addWindowListener 684-685
 aderir às marcas divisórias 680
 adição 111-112
 adicionando componentes a um painel de conteúdo 606-607
 “adivinhe o número”, jogo 313
AdjustmentEvent 610
 adormecida, estado 771-772, 776-777, 779-780
 adormecida, *thread* 771-774
 adormecimento expira, intervalo de 772-773
 ADTs 418-419
 adulterados ou decorados, nomes 296-297
 Advanced Java 2 Platform How to Program 53-54, 61-62, 84-85, 668-669, 931
 Advantage Hiring, Inc. 1320-1321
 afundamento, classificação por 332-333
 afundar 332-333
 agregação 162-163, 380-381, 420-421, 725-726
 .aif ou .aiff, extensão 980-981, 1113, 1148
 aleatoriamente, formas dimensionadas 665-666
 aleatoriamente, triângulos gerados 600-601
 aleatório 414, 591-592
 aleatório, número 1035
 aleatórios para criar sentenças, geração de números 558
 aleatórios, geração de números 544-545
 aleatórios, gerador de números 1057-1058
 aleatórios, *limericks* 558
 aleatórios, processamento de números 271
Algorithms1.java 1097-1098
 algoritmo 173, 182-183, 292-293, 730-731, 1082, 1093
 algoritmo da estrutura de coleções 1093
 algoritmo de pesquisa binária 1098-1099
 almoxarifado do computador, seção de 55
 alternância, botões de 617
 alto-falante 979-980
 alto-falante 55
 altura 576-579
 altura de um *applet* em pixels 146-147, 263-264
 altura de um retângulo em pixels 155-156
ALU 55
 America's Job Bank 1318-1319
 American National Standards Committee on Computers and Information Processing (X3) 57-58
 American National Standards Institute (ANSI) 52-53, 419
 American Society for Female Entrepreneurs 1320-1321
 amigável para o navegador 973
 amostragem, taxa de 1132-1133
 amostras de cor 572
 ampliando 985-986
 ampulheta 899-900
 análise 68-69, 121, 123
Analysis.java 196-197
and 1070-1071
 Andrew Koenig 740
 anexar um *string* 109
 ângulo de um arco 581-582
 animação 51-54, 83-84, 140-141, 298, 566-567, 601-602, 967-968, 973-974, 985-986, 1186-1187, 1190-1191
 animando uma série de imagens 970-971
AnimatedPanel 1181-1183, 1186-1187, 1189-1191, 1306-1312
AnimatedPanel.java 1186-1187
Animator, applet 137-138
 aninhada, estrutura **for** 234-325, 342-345
 aninhada, estrutura **if/else** 179-181, 337-338, 432-433
 aninhada, estrutura **switch** 229-230
 aninhadas, estruturas 244, 246
 aninhadas, estruturas de controle 193-194, 229-230, 274-275
 aninhados, blocos 283-284
 aninhados, blocos de construção 244, 246
 aninhados, parênteses 111-112
 aninhamento, regra de 243-244
 aninhando 178, 217-218, 246-247
 aninhar 194-195
 anotação na UML 722-723
 ANSI 52-53, 57-58, 419
 antecipadamente, carregando 1122-1123
 anti-aliasing 140-141
 aparência e comportamento 605-608, 642-643, 700
 aparência e comportamento, observações de 59-60
 apenas de leitura, arquivo 833-834
 apenas de leitura, texto 607-608
 apenas de leitura, variável 279-280
 apenas de leitura, variável 323-324
 API 61-62
 aplicativo 62-64, 94, 96-97, 120-121, 136, 157-159
 aplicativo de linha de comando 742-743
 aplicativo para desenhar 505
 aplicativos colaboradores 814, 931
 aplicativos comerciais 60-61, 818
 aplicativos de console 742-743
 apóstrofes 509-510
append, método de *JTextArea* 224-226
append, método de *String-Buffer* 531-532
 Apple Computer, Inc. 55-56, 1337-1338
Applet 140-143, 270-271, 642-643, 969-970, 1192-1193
applet 62-64, 66, 94, 136, 154-155, 159-160, 263-264, 283-284, 606-607, 892-893, 969-970, 973-974
applet que também pode ser executado como aplicativo 685-686
applet vindo da World Wide Web 818
AppletContext 892-893
applets de domínio público 892-893, 973-974
applets intensivos em cálculos 66
appletviewer 136, 138-139, 140-141, 143-144, 145-146, 216, 218-219, 278-279, 298, 685-686
Aquent.com 1324-1326
arc, método 582-583
Arc2D 564
Arc2D.CHORD 590
Arc2D.Double 586-589, 601-602
Arc2D.OPEN 590
Arc2D.PIE 588-589
arcHeight 579-581
arco 137-138, 580-581
ArcTest, applet 137-138
arcWidth 579-581
 área de desenho dedicada 671-672
 área de exibição de um *applet* 146-147
 área de um círculo 314-315
 áreas de desenho como subclasses de *JPanel* 671-672
 argumento 98-99, 261-262
 argumento de linha de comando 138-139, 140-141
 argumento para uma chamada de método 263-264
ArithmeticeException 746-747
 arquivo .class 62-64, 99, 378, 384-385, 405-406, 487-488, 980-981

- arquivo **.class** de um *applet* 146-147
 arquivo 818-819
 arquivo de código-fonte contendo comentários de documentação, 1230-1231
 arquivo de contas a receber 887-888
 arquivo de lote 104-105
 arquivo de pastas suspensas 140-141
 arquivo, apontador de posição no 840, 846, 850-851
 arquivo, escopo de 477-478
 arquivo, modo de abertura de 850-851
 arquivo, nome de 96-97, 144
 arquivo, pasta de 140-141
 arquivo, posição no 840
 arquivo-mestre 886-887
 arquivos, processamento de 51-52, 754-755, 818, 820-821, 931
 arquivos, programa de comparação de 886-887
 arquivos, servidor de 55-56
 arrastando o mouse 671-672
 arrastar a caixa de rolagem 624-625
 arrastar o mouse 140, 632-633, 665-666
 arrastar, evento de 680
 arrastar, operação de 680
 array 293-294, 317, 818, 893-894, 1106-1107
array alocado com **new** 329
array bidimensional 342-343, 345-346, 354-355
arraycopy 1083-1084
ArrayIndexOutOfBoundsException 329, 586-587, 744-745, 751
ArrayList 1087-1089, 1093, 1098-1101
Arrays 1082-1083
arrays de arrays 342
arrays são objetos de Java 373
 arredondado, retângulo 247-248, 579-580, 589-590
 arredondando 110-111, 309
 arredondando erros 270-271
 arredondando um valor para o inteiro mais próximo 309
 arredondar 193-194, 226, 262-263
 arredondar um valor para uma quantidade específica de casas decimais 309
 árvore 140-141, 1007-1008, 1100-1101
 árvore binária 989, 1012-1013
 árvore de pesquisa binária 1007-1008, 1011-1012
ASCII 371, 1337-1338
asList 1085-1086, 1087, 1093-1094, 1101, 1109-1110
 aspas francesas (« ») 725
 aspas, « 97-98, 100-101
 assinados, *applets* 931
 assinatura 296-297, 386-387, 438-440
 associação 68-69, 82-83, 162-164, 420-422
 associam da direita para a esquerda 111-112, 119-120, 193-194, 201
 associam da esquerda para a direita 119-120, 201
 associatividade de operadores 111-112, 119-120, 201, 318-319
 assunto, objeto 729-730
 asterisco (*) 151-152
 ativa, área 977
 ativar um campo de texto 107-108
 atividade 82-83, 124-125, 246-247, 250
 ativo, ponta 138-139
 ator 652, 652-653
 atribuindo objetos de classe 376-377
 atribuindo referências para subclasse a referências para superclasse 436-437
 atribuir um valor a uma variável 107-108
 atributo (dados) 143-144, 161-162, 165-166, 373-376, 446-447
 atributo 67-68, 82-83, 202-206, 246-247, 249-250, 420-422, 488-493, 593-594, 807-809
 atributo **archive** 399-400
 .au 1113, 1148
 .au, arquivo 1190-1191
 .au, extensão de arquivo *Sun Audio* 1113, 1148
 áudio 51-54, 121, 123, 270-271, 601-602, 1147-1148, 1181-1182, 1190-1191
AudioClip 979-984, 1190-1193, 1306-1310
AudioFormat 1151-1152
AudioFormat.Encoding 1151-1152
AudioInputStream 1150-1151
AudioSystem 1150-1151
 áurea, média 288-289
 áurea, relação 288-289
 Australian National Botanic Gardens, *sítio* da Web 982, 1181
Author, anotação 1234-1235
-author, argumento 1237-1238
 autodocumentando 106-107
 auto-referencial interligados, objetos de classe 990-991
 auto-referencial, classe 989-991
 auxiliar, método 375-376, 1011-1012
 avaliando expressões 1017-1018
Average1.java 182-183
Average2.java 190-191
 .avi (extensão Microsoft Audio/Video Interleave) 1113
 .avi 1113
AWTEvent 610
AWTException 754-755
- B**
- B 57-58
 Babbage, Charles 61
 baixa prioridade, *thread* de 445-446, 771-774, 814
 balanceada, árvore binária 1013-1014
Balking, padrão de projeto 494-495, 809-810
 banco de dados 53-54, 892-893
BankUI.java 825-826
BarChart, *applet* 137-138
 barra de asteriscos 324-325
 barra invertida, \ 100-101
 base 50-51, 56-58, 538-539, 1058-1059
 Basic 60-61, 989
Basic Latin 1341-1342
BasicStroke 564, 587-588, 590
 bate-papo cliente/servidor 901-902
BCPL 57-58
Bell 161-166, 204-207, 298-301, 350-351, 420-422, 491, 493, 551-552, 594-595, 807-809, 1257, 1258, 1273-1275, 1289, 1306-1307
 Bell Laboratories 57-58
BellEvent 549-550, 1242-1243, 1249, 1281-1282, 1311-1312
BellListener 594-596, 1245-1246, 1249, 1257-1258, 1270-1271, 1273-1274, 1281-1282
 biblioteca, classes de 59-60, 741-742
 biblioteca-padrão de gabaritos (STL) 1082-1083
 bibliotecas de classes 59-60, 226, 400-401, 419, 432-433, 446-447
Bid4Geeks.com 1323-1324
 bidimensionais, demonstração de gráficos 140-141
 bidimensionais, formas 564
 bidimensionais, gráficos 586-587, 967
 bidimensional, *array* 342-343
 bidimensional, desenho 140-141
 bidimensional, estrutura de dados 1007-1008
BigDecimal 288-289
BigInteger 288-289
BilingualJobs.com 1320-1321
 binário 254-255
binarySearch 1082-1086, 1098-1101
BinarySearch.java 337-338
BinarySearchTest.java 1099-1100
BindException 912
 bit (unidade de tamanho) 1337-1338
 bit de vai-um (*carry*) 1223
BitSet 1071-1072, 1077-1078
BitSetTest.java 1072, 1074
BitShift.java 1066-1067
 Bjarne Stroustrup 740
black jack, jogo de cartas 964-965
Blackvoices.com 1319-1320
_blank, *frame* de destino 896
Blink, *applet* 137-138
 bloco 180-181, 192, 266, 900-901, 914-915
 bloco **try** que o envolve 751, 760-761
 blocos aninhados 283-284

- blocos de construção 173, 216, 243-244
 bloqueado para entrada/saída 773-774
 bloqueando um objeto 777-778
 bloqueia até que a conexão seja recebida 906
 bloqueio 784-785
 bloqueio fatal 771-772, 777-778, 800-801, 809-810, 815, 947
Bluetooth 1323-1324
 boas práticas de programação 59-60
Bohm, C. 174-175, 244, 246
 bola que salta 1204-1205
BOLD 573-574, 620-621
Booch, Grady 69-70
Boolean 488-489, 1005-1006, 1042
boolean, promoções 269-270
boolean, tipo primitivo de dados 175-177, 202-203
boolean, variáveis inicializadas com **false** 282-283
 borbulhar 1256-1257, 1264-1265, 1271, 1281-1282
 borda 684-685
BorderLayout 633-634, 641-643, 646, 671-672, 689-690, 696-697, 701-702, 707-711
BorderLayout.CENTER 646-649, 674-675
BorderLayout.EAST 646-647
BorderLayout.NORTH 646-647, 687, 708-709, 711-712
BorderLayout.SOUTH 633-634, 646-647, 651-652, 682-683
BorderLayout.WEST 646-647
BorderLayoutDemo.java 646
Boss.java 459-460
 botão 102-105
 botão de comando 617
BOTH 718-720, 736-737
bottom 564-565
Box, container 670-671, 707-708
BoxLayout 670-671, 707-709
BoxLayout.X_AXIS 710-711
BoxLayout.Y_AXIS 710-711
BoxLayoutDemo.java 707-708
 branco, caracteres de espaço em 95-98, 119-120, 524-525, 541-542
 branco, espaço em 176-177
Brassringcampus.com 1326-1327
break 175-176, 228-230, 233, 256-257, 283-284
BreakLabelTest.java 235-236
BreakTest.java 233
Bridge, padrão de projeto 494-495, 497-498, 727-728
 brilho 572-573
bubble sort 332-333, 355-356
BubbleSort.java 333-334
bucket sort 362-363
buffer 778-779, 822-823, 906, 1135-1136
buffer circular 787-788, 793-794
buffer vazio 778-779
BufferedImage 587-590
BufferedInputStream 823
BufferedOutputStream 823, 957
BufferedReader 823-824, 880-881, 937
BufferedWriter 823-824
Builder, padrão de projeto 494-495
Button 488-490, 548, 594-595, 617, 805-808, 1257-1261, 1264, 1264-1265, 1270-1271, 1271, 1274-1275, 1281-1282, 1287-1289
ButtonEvent 549-550, 1243-1244, 1249, 1264-1265, 1271, 1281-1282, 1310-1311
ButtonGroup 622, 690-694, 698-700
ButtonListener 594-596, 1245-1247, 1249, 1257-1258, 1264-1265, 1270-1271, 1271, 1281-1282
ButtonTest.java 617
ButtonText.java 617
Byte 1042
byte 202-203
Byte 488-489
byte, array 939-940
byte, promoções 269-270
byte, tipo primitivo de dado 175-176, 226, 957, 1058-1059
ByteArrayInputStream 823-824
ByteArrayOutputStream 823-824
bytecode 61-63, 66, 98-99, 145-146, 989
- C**
- C 50-51, 52-53, 55-58, 60-61, 373-374, 769
C How to Program 52-54
 C, linguagem de programação 57-58
 C++ 52-53, 55-59, 769
 C++ Como programar 53-54
 cabeça 989
 cabeça de uma fila 1004-1005
 cadeia 901
 cadeia de chamadas 758
 caixa de combinação 140-141, 624-625
 caixa de diálogo 100-102, 104-105, 695-696
 caixa de marcação não está marcada 140-141
 caixa preta 149-150
 caixas de marcação 140-141, 617, 621-622
 cálculo aritmético 110-111
 camada cliente 959-960
 “caminhar” além do fim de um *array* 329
 caminho absoluto 876, 880-881
 caminho, informações sobre o 876
 campo 818-819
CampusCareerCenter.com 1326-1327
CANCEL_OPTION 830-831, 833
CannotRealizeException 1133-1134
canRead 876
 canto superior esquerdo (0, 0) de um *applet*, coordenadas do 141-143, 145-147, 149-151
 canto superior esquerdo de um componente GUI 564
canWrite 876
CAP_ROUND 588-589
 capacidade 1035-1036
 capacidade de reutilização 989
 capacidade de um *StringBuffer* 527-528
 capacidade *default* 1047-1048
 capacidade *default*, incremento da 1035-1036
capacity 529-530, 1042
Caps Lock 640-641
 capturar erros relacionados 756-757
 capturar tipos de superclasse 745-746, 756-757
 capturar todas as exceções 744-746, 762
CaptureDeviceInfo 1133
CaptureDeviceManager 1133, 1195-1196
CapturePlayer 1132-1133
CapturePlayer.java 1124-1125, 1132
 caractere 271, 818-819, 1338-1339, 1343-1344
CardDeck.java 711-712
CardLayout 707, 710-711, 713-714
Cards.java 1094-1095
CardTest, applet 137-138
Career.com 1318-1319
CareerLeader.com 1322
CareerPath.com 1318-1319
CareerWeb 1318-1319
 carga, fator de 1046-1047
 carregador de classes 62-64, 953
 carregando 62-64
 carregando e reproduzindo um *AudioClip* 980-981
 carregando um documento de um URL para um navegador 894-895
 carregar 369-370
 carregar outra página da Web em um navegador 977, 979-980
 carregar um *applet* 147-148
 carregar/armazenar, operações 366-367
case, palavra-chave 175-176
case, rótulo 228-231
cases em uma instrução *switch* 229-230
 caso básico 285-288, 291-292
 caso *default* 274-275
 cassino 272, 275-276
catch, bloco 742-745, 750-751, 758
catch, tratador 175-176, 703-704, 743-746, 750-751, 757-758

- cauda de uma fila 1004-1005
 cavalo fechado: teste do passeio do 361-362
 cavalo, condutor do passeio do 1205
 cavalo, exercício do condutor do passeio do 1205
 cavalo, passeio do 359, 601-602
 cavalo: abordagem com força bruta, passeio do 361-362
`cd` para mudar de diretório 137-138, 140-141
`ceil`, método de `Math` 262-263
 Celsius 664
 Celsius, equivalente de uma temperatura em Fahrenheit 311-312
CENTER 644-645, 671-672, 689-690, 715
Chain-of-Responsibility, padrão de projeto 494-495, 497-498, 729
 chamada 330
 chamada bloqueante 937
 chamada por referência 330
 chamadas de método concatenadas 409
 chamadas de métodos em cascata 409
 chamador 260-261
 chamando polimorficamente um método de desenhar formas, 665-666
 charnando um método 217-218, 260-261
ChangeEvent 683-684
ChangeListener 683-684
char, array 511-512
char, palavra-chave 175-176, 202-203
char, promoções 269-270
char, tipo primitivo de dados 226
char, variável 106-107
Character 488-489, 509, 531-532, 536, 1005-1006, 1042, 1103-1104
Character, métodos de conversão da classe `static` 538-539
CharArrayReader 823-824
CharArrayWriter 823-824
charAt, método 530-531, 1103-1104
charValue, método 540-541
 chave 893-894, 1045-1046, 1052-1053
 chave à direita de fechamento `{}` 283-284
 chave duplicada 1102-1103
 chave, valor da 335, 1013-1014
 chave/valor, par 1045-1047, 893-894
 chaves `{ e }` 152-153, 180-181, 192, 218-220, 231-232, 234-235, 265-266
 chaves não-obrigatórias 229-230
 chaves que delimitam uma instrução composta 180-181
 chaves, `{ e }` 321-322
CheckBoxTest.java 619-620
ChiefMonster 1325-1326
 ciclo de execução de uma instrução 369-370
 ciclo de vida de uma *thread* 772-773
 científicos e de engenharia, aplicativos 60-61
 cifrão, sinal de (\$) 95-96
- Circle.java** 440-441, 444-445, 467-468, 474-475
 círculo 580-581, 599-600
 círculo de tamanho aleatório 737-738
 círculos concêntricos 599-600
 circunferência 132, 169-170, 737-738
CJK Unified Ideographs 1341-1342
 clareza 50, 64-65
Class 953
class 818-819
class, palavra-chave 95-96, 175-176
ClassCastException 436-437
 classe 59-60, 67-68, 82-83, 206-207, 250-251, 298-302, 347-348, 373-374, 420-421
 classe `AbstractList` 1105-1106
 classe `AbstractMap` 1105-1106
 classe `AbstractSequentialList` 1105-1106
 classe abstrata 455-457, 465-466, 471-472, 489-494, 499-500, 969-970, 1258-1259
 classe adaptadora 487-488, 635
 classe *applet* compilada 146-147
 classe ativa 802-804, 806-807, 1263-1264, 1272-1273
 classe base 143-144, 375-376
 classe `CaptureDeviceInfo` 1132-1133
 classe `Complex` 425-426
 classe concreta 455, 505
 classe declarada `final` 455
 classe derivada 143-144, 375-376
 classe interna anônima 477-478, 483-484, 486-487, 626-627, 636-637, 656-657, 670-671, 680, 683-684
 classe não pode herdar de uma classe final 455
 classes 260
 classes para implementar tipos de dados abstratos 419
 classificação em árvore binária 1012-1013
 classificação, ordem de 1100-1103
 classificação, técnicas de 138-139
ClassNotFoundException 837-838, 904-905, 909-910
CLASSPATH 386
`-classpath`, argumento de linha de comando para `java` 386, 824-825
`-classpath`, argumento de linha de comando para `javac` 386, 824-825
clear 1071-1072
clearRect, método 579
 clicando com o mouse 610
 clicando na caixa de fechamento 737-738
 clicando num botão 610
 clicar com o mouse 104-105, 107-108, 138-139, 154, 618-619
 clicar duas vezes em um elemento 604-605
 clicar em itens de menu gera eventos de ação 690-691
 clicar nas setas de rolagem 624-625
 clicar num botão 617
 clicar numa aba 142
Client.java 906-907, 915-916
 cliente 298-299, 349-350, 376-377
 cliente de uma biblioteca 741-742
 cliente de uma classe 418-419
 cliente do objeto 143-144
 cliente/servidor 915-916
ClientGUI.java 947-948
Clip 1150-1151
Clip, método start de 1151-1152
Clip LOOP_CONTINUOSLY 1151-1152, 1197
 clipe de áudio 979-982, 1273-1274
 clipes de áudio 769-770, 1181
ClipPlayer.java 1148
ClipPlayerTest.java 1152-1153
 clique com o botão do mouse 639-640
Clock, applet 137-138
close 833-834, 842-843, 853-854
close, método da classe `MulticastSocket` 947-948
closePath 590-591
COBOL (Common Business Oriented Language) 60-61
code, atributo da marca <applet> 146-147
 codificação 1337
 código ativo (*live-code™*), abordagem de 50
 código em C ou C++ compilado 66
 código-fonte, 136-137
 coerção 752
 coerção de argumentos 269-270
 coerção perigosa 442
 colaboração 82-83, 202-203, 347-351, 550-552, 1260-1261
 colateral, efeito 238-239, 292, 330
 colchetes, [e] 318-319
 colchetes, [] 328-329, 352-353
 coleção 456
 coleção de suporte 1104-1105
 coleções 87-88, 1035
 coleta de lixo automática 757-758, 769-770
 colisões 1046-1047
Collection 1041-1042, 1087, 1088-1089, 1101
collection 1093
Collection, implementação de 1104-1105
Collections 1095-1096, 1097-1098
CollectionTest.java 1087-1088
Collegegrads.com 1326-1327
Color 566-568, 590-591
Color, constante 566-570
 coluna 342
ComboBoxTest.java 624-625
 comentário de classe 1234-1235
 comentário, (/) 94, 106-107

comentários de uma só linha ao estilo de C++ 94-95
 comissão 210-211, 354-355
Command Prompt 62-64, 97-98
Command, padrão de projeto 494-495, 497-498
CommissionWorker.java 460
Common Birds, link 1181
 comparando objetos **String** 513-514
Comparator, objeto 1093-1095, 1097-1098, 1100-1103
Comparator, objeto em **sort** 1093-1094
compareTo 514, 515-516, 556-557
Comparison.java 114-116
 compartilhada, memória 784-785
 compartilhada, região de memória 778-779
 compartilhados, dados 778-779
 compartilhar 55-56
 compartimento de atributo 205-206
 compatibilidade com versões anteriores 1237
 compilador 56-57, 62-64
 compilar 98-99, 989
 compilar um programa 61-62
 complemento, ~ 1058-1059
 completa 771-772
 completamente qualificados, nomes de classe 384-385
 completo, passeio 601-602
 completos, nome de pacote e nome de classe 151-152
Complex 425-426
 complexidade 335
 complexidade computacional 335
Component 565-566, 572, 604-605, 632-633, 677-678, 684-685, 718-719, 726-729, 970-971, 1121
Component, método **paint** de 566-567
Component, método **repaint** de 566-567
ComponentAdapter 635
 componente 51-52, 67-69, 83-84, 124-125, 270-271, 631-632, 722-726
 componentes “peso-pesado” 605-606, 690-691
 componentes 58-59
 componentes AWT 605-606
ComponentEvent 610
ComponentListener 635
 comportamento (método) 143-144, 373-376
 comportamento 67-68, 82-83, 298-299, 446-447
 comportamento do sistema 247-248, 250, 349-350, 652
 composição 380-381, 402-403, 432-433, 447-448
Composite, padrão de projeto 494-495, 497-498, 728-729
 compras, lista de 181-182
 comprimento 529-530, 686-687

computação cliente/servidor 55-56
 computação cliente/servidor sem conexão com datagramas 912
 computação colaboradora 53-54
 computação crítica para o negócio 750-751
 computação distribuída 55-56
 computador 54
 computadores na educação 312-313
concat 521-52
 concatenação 109, 118-119, 528-529
 concatenação, operador de, + 155-156, 377-378
 concatenar 264-265, 377-378
 concorrência 769
ConcurrentModificationException 1263-1264
 condição 113-114
 condição de guarda 248-249
 condição dependente 238-239
 condicional, operador, ?: 178-179, 201
 conectar-se a um servidor em uma porta 900-901
 conectar-se ao servidor 901
 conexão 891-892, 910-911
 conexão a um servidor 924
 conexão de um cliente 924
 conexão entre o cliente e o servidor termina 901-902
 conexões de clientes 900-901
 confiável, fonte 931
 configurando uma referência para objeto como **null** 769-770
configureComplete, método 1142
ConfigureCompleteEvent 1142
Configuring 1142
 confundindo o operador de igualdade == com o operador de atribuição = 114-116
 conjunto de caracteres 133-134, 138-139, 818-819, 1337-1338
 conjunto de caracteres ASCII, Apêndice 1215
 conjuntos, interseção de 427-428
 conjuntos, união de 427-428
connect, método de **DataSource** 1133
ConnectException 901-902
 conservar memória 769-770
 constante caractere 509-510
 Construindo seu próprio compilador 989, 1024-1025
 Construindo seu próprio computador 366
 construtor 349-350, 386, 444-445
 construtor de classe 420-421
 construtor de cópia 509-510
 construtor *default* 386-387, 402-403, 437-438, 442, 509-510
 construtor *default* de uma classe interna anônima 486-487
 consulta, método de 391
ConsumeInteger.java 780, 783-784
 consumidor 781-782, 784-785
 consumindo dados 780
 consumir memória 292-293
 contador 182-183, 188-189
 contagem de cliques 637-638
Container 263-265, 278-281, 564-565, 606-609, 642-643, 650-651, 728-729
ContainerAdapter 635
ContainerListener 635
contains 1052-1053
 contêiner 604-605
 contêiner de *applets* 136, 144-145
 contêiner para menus 690-691
 contenção para o processador 777-778
Content Descriptor.RAW_RTP 1142
ContentDescriptor 1142
 contexto gráfico de um *applet* 565-566
continue 175-176, 233-234, 236-237, 256-257, 283-284, 937, 947
ContinueLabelTest.java 236-237
ContinueTest.java 233-234
 controlador (na arquitetura MVC) 957-958, 958-959
 controlador 83-84, 120-121, 123-124, 721-725
 controle de concorrência 815
 controles 604-605
Controller 1113-1114, 1121
ControllerAdapter 1121-1122
ControllerEvent 1122-1124
ControllerListener 1121-1122
 convergir para um caso básico 285-286
 conversão 442
 conversão automática 109
 conversão entre classes e tipos primitivos 376-377
 conversões entre sistemas de numeração 538-539
 convertendo uma referência a superclasse para uma referência a subclasse 504-505
 converter um *applet* em um aplicativo baseado em GUI 685-686
 converter um número binário para decimal 1221-1222
 converter um número hexadecimal para decimal 1221-1222
 converter um número octal para decimal 1221-1222
 converter um valor integral para um valor em ponto flutuante 270-271
 converter uma referência a superclasse para uma referência a subclasse 436-437
Cooljobs.com 1326-1327
 coordenadas (0, 0) 564
 coordenadas 144-145
 coordenadas cartesianas 426-427
 coordenadas nas quais o usuário libera o botão do mouse 665-666
 cópia de um objeto grande 373-374
 copiar texto de uma área de texto para outra 668-669
copy, algoritmo 1096-1098, 1108-1109
 cor 140, 564

- cor de desenho 568-570
 cor de fundo 570, 572, 664-665, 678-679
 cor *default* 140
 corpo 96-97
 corpo de um laço 221-222
 corpo de uma definição de classe 375-376
 corpo de uma definição de método 97-98
 corpo de uma estrutura *if* 113-114
cos, método de *Math* 262-263
 co-seno 262-263
countTokens 543
Courier, fonte 341-342, 693-694
 Courtois, P. J. 815
 CPU 55
craps (jogo de cassino) 272, 275-276, 279-280, 314-315, 357-358, 1205
Craps.java 275-276
createDataSink 1195-1196
createDataSink, método de *Manager* 1134-1135
createDataSource, método de *Manager* 1133
createGlue 708-709, 710-711
createHorizontalBox 670-671, 707-708
createHorizontalGlue 708-710
createHorizontalStrut 709-710
createPlayer, método de *Manager* 1121-1122
createProcessor, método de *Manager* 1133-1134
CreateRandomFile.java 848-849
createRealizedProcessor 1195-1196
createRigidArea 708-709
CreateSequentialFile.java 828-829
createVerticalBox 707-708
createVerticalGlue 710-711
createVerticalStrut 709-710
CreditInquiry.java 840
 criando novos tipos de dados 418-419
 criando um pacote 382-383
 criando uma classe reutilizável 382-383
 criando uma instância 192-193
 criar novas classes a partir de definições de classes existentes 375-376
 criar novos tipos 419
 criar um objeto de classe interna 488-489
 criar um pacote 382-383
 criar um *Socket* 901
 criptografar 214
 Crivo de Eratóstenes 361-362, 1072, 1074
 Cruel World 1322
Ctrl, tecla 629-630, 642-643
 cultura corporativa 1317, 1319-1320
currentThread 771-772, 780-781, 785
 cursor 97-98, 100
Cursor 899-900
Cursor.DEFAULT_CURSOR 899-900
Cursor.WAIT_CURSOR 899-900
 curto-círcuito, avaliação em 238-239
 curva 138-139, 590
 curva complexa 590
CustomPanel.java 673
CustomPanelTest.java 674
 Cyber Classroom 50-51
Cylinder.java 451-452, 468-469, 475-476
- D**
- d argumento 1237-1238
 - d linha de comando 1237-1238
 - dados 54, 67-68
 - dados de um tipo abstrato, representação de 418-419
 - dados no suporte de ações 418-419
 - dados, abstração de 418-419, 453-454
 - dados, entrada de 102-103
 - dados, estrutura de 376-377, 989, 1106-1107
 - dados, hierarquia de 818-820
 - dados, integridade de 391-392
 - dados, manipulação de 967
 - dados, membro de 373-374
 - dados, programa de lançamento de 601-602
 - Dartmouth College 60-61
 - data 271
 - datagrama 911-912
 - DatagramPacket* 912-913, 931, 937-940, 947
 - DatagramSocket* 912-913, 915-916
 - DataInput*, interface 823-824
 - DataInputStream* 820-821, 823, 840, 846, 921-922, 924-926
 - DataLine.Info* 1151-1152
 - DataOutput* 823-824
 - DataOutputStream* 820-821, 823, 840, 846, 921-922, 924-926
 - DataSink* 1124-1125, 1132-1137
 - DataSinkEvent* 1134-1135, 1195-1196
 - DataSinkListener* 1134-1135, 1195-1196
 - dataSinkUpdate*, método 1134-1135, 1195-1196
 - DataSource* 1123-1124, 1142
 - DataSource*, classe 1132-1133
 - Date.java* 402-403
 - DBCS (*double byte caractere set*) 1339-1340
 - DEC PDP-11, computador 57-58
 - decifrar 214
 - decimal (base 10), sistema de numeração 1217
 - DecimalFormat*, classe 192, 340-341
 - decisão 50-51, 113-114, 177, 246-247
 - decisão, símbolo de 175-177
 - DeckOfCards.java* 544-545
 - declaração 106-107, 181-182, 266
 - declaração de *array* 321-322
 - declaração e inicialização de *array combinadas* 319-320
 - Decorator*, padrão de projeto 494-495, 497-498, 957
 - decrementar 216
 - decrementar, operador de, - 199-201
 - default**, caso 228-230
 - default**, palavra-chave 175-176
 - DEFAULT_CURSOR** 898-899
 - definição de classe 95-97, 105-106, 143-144, 152-153, 297, 375-376, 436-437
 - definição de classe de aplicativo 184-185
 - definidas pelo programador, classes 95-96
 - definido pelo programador, método 260-261
 - definido pelo programador, tipo 373-374
 - definir um pacote de classes para reutilização 378
 - Deitel & Associates, Inc. 52-53
 - Deitel, H. M. 815
 - deitel@deitel.com* 54, 85-86
 - DeitelMessenger.java* 954-955
 - DeitelMessengerServer.java* 931-932
 - delete* 533-534
 - deleteCharAt* 533-534
 - delimitadores 541-542
 - demo**, diretório 140-141
 - demonstração, *applets* de 159-160
 - Department of Defense (DOD) 61
 - dependência 722-724
 - Deprecated*, link 1238-1240
 - Deprecated*, nota 1237
 - deprecated-list.html* 1241
 - depuração 454-455
 - depurando aplicativos com múltiplas *threads* 771-772
 - dequeue* 1004-1005
 - descarregar o *buffer* de saída 911-912
 - descarregar o fluxo de saída 906
 - descida 576-579
 - descobrindo um *applet* 566-567
 - DESELECTED** 621-622
 - desempenho da classificação e pesquisa em árvore binária 1032-1033
 - desempenho, dicas de 59-60
 - desenfileirar, operação 420
 - desenhando 565-566
 - desenhando em aplicativos 565-566
 - desenhando retângulos, *strings* e elipses 217-218
 - desenhar arco 137-138
 - desenhar curva complexa 138-139
 - desenhar formas 564
 - desenhar gráficos 141-145
 - desenhar linhas 217-218
 - desenhar linhas e pontos 138-139
 - desenhar linhas e pontos em cores diferentes 140
 - desenhar retângulo 154-155, 168-169

- desenhar `String` 140-141
 desenhar uma forma com o mouse 664-665
 desenvolvimento de *software* orientado a componentes 419
 desenvolvimento rápido de aplicações (RAD), 400-401
Design Patterns, Elements of Reusable Object-Oriented Software 70-71
`DesktopTest.java` 704-705
 deslocados e escalonados, inteiros aleatórios 272-273
 deslocamento 847
 deslocamento em *bytes* 847
 deslocamento, valor de 275-276
 deslocar 272
 desnecessários, parênteses 113-114
 despachando um evento 616
 despachar 772-773
 destino, *frame* de 896
`destroy`, método de *JApplet* 298
 destruir recursos alocados para um *applet* 298
 desvio 248-250, 728-729
`developer.java.sun.com/devel_oper` 160
 devolver 260-261
 devolvida, mensagem 805-806
 devolvido, tipo 299-302
 devolvido, tipo do valor 266
 diacrítico 1338-1339
 diagrama de atividades 82-83, 123-124, 247-250, 1287-1288
 diagrama de classes 82-83, 123-124, 161-164, 202-203, 205-206, 299-301, 420-422, 490-491, 490-493, 594-596, 722-725, 806-807, 118-119, 319-320, 1182-1183, 1256-1257, 1264-1265, 1274-1275, 1282
 diagrama de colaborações 83, 123-124, 349-351, 550-552, 803-804
 diagrama de componentes 83-84, 123-124, 722-725, 1248, 1288, 1311-1312
 diagrama de instalação 123-124
`Dialog` 574-575
`DialogInput` 574-575
 diálogo para seleção de cor 572
 diâmetro 132, 169-170, 737-738
`Dice.com` 1322-1323
`Dictionary`, classe 1035
`digit` 537-538
 Digital Equipment Corporation, PDP-7 57-58
 digitando em um campo de texto 610
 dígito 106-107, 536-539, 1217
 dígitos invertidos 311-312
`Dimension` 677-678, 681, 705-706, 708-709, 929-930, 973-974
 dimensionando uma imagem 967-970, 984-985
 dinâmica de memória, alocação 192-193, 989-990
 dinâmica de métodos, vinculação 454-455, 464-465
 dinâmica, vinculação 441-442, 445-446, 471-472
 dinamicamente redimensionável, *array* 893-894, 1042
 dinamicamente vinculado, método 464-465
 dinâmicas, estruturas de dados 989
 dinâmico, *array* 754-755
 dinâmico, conteúdo 58-59
 dinheiro, cálculos em 226
`dir`, comando no Windows 137-138
Direct Sound 1123-1124
`DIRECTORIES_ONLY` 828-829
 direita 564
 direita com extensão de sinal, operador de deslocamento à, >> 1059-1060, 1070-1071
 direita com extensão de zero, operador de deslocamento à, >>> 1066-1067, 1070-1071
 direita, }, chave à 143-144, 185-186, 192
 direita, alinhado à 643-644
 direita, filho à 1007-1008
 direita, subárvore 1007-1008, 1011-1012, 1020-1021
 direta e indireta, super classe 468-469
 direta, superclasse 431-432
 diretório 824-825, 875-877, 1237-1238
 diretório *applets* 137-138
 diretório no arquivo JAR, estrutura do 399-400
 diretório, nome de 876
 diretórios, árvore de 140-141
 disco 54-56, 64-65, 818
 disco, acionador de 136-137
 disco, espaço em 989-990
 disco, finalização de E/S em 740-741
 discussão, fóruns de 160
 disparada novamente, exceção 758
 disparar exceção verificada 752
 disparar novamente uma exceção 746, 766-767
 disparar objetos de subclasses 751
 disparar um evento 604-605
 disparar uma exceção 742-744, 750-751
 disparo, ponto de 743-744
`dispose` 684-685
`dispose`, método de *RTPManager* 1142-1143
`DISPOSE_ON_CLOSE` 684-685
 dispositivo 54
 dispositivo de captura 1114-1115, 1123-1124
 dispositivos eletrônicos de consumo 58-59
 distribuição de cartas 544-545
 distribuidos, aplicativos cliente/servidor 55-56
`DitherTest`, *applet* 137-138
 diversidade 1319-1320
- `DivideByZeroException.java` 746-747, 750-751
`DivideByZeroTest.java` 747-748
 dividir para conquistar, abordagem 260, 262-263
 divisão 111-112
 divisão por zero 64-65, 188-189, 419, 746-747
 divisórias, marcas 680
`do/while`, estrutura de repetição 175-176, 229-232, 242-243, 246-247
`do/while`, fluxograma 232
`DO NOTHING ON CLOSE` 684-685
 documentação, comentário de 94-95, 1229-1231, 1234
 documentar o código Java 1229-1231, 1234
 documento 136, 668-669, 703-704
 documento do ANSI C padrão 64-65
 DOD 61
`Dogfriendly.com` 1326-1327
 dois maiores valores 210-211
 dois, complemento de 1223
`Door` 489-490, 594-595, 807-808, 1257-1258, 1260-1261, 1263-1265, 1270-1271, 1281-1282, 1287-1289, 1306-1307
`DoorEvent` 549-550, 1243-1244, 1249, 1263-1264, 1271, 1281-1282, 1310-1311
`DoorListener` 592-596, 1245-1247, 1249, 1257-1258, 1263-1264, 1270-1271, 1281-1282
`Double` 154-155, 488-489, 1042
`double` 106-107, 149-155
`double`, promoções 269-270
`double`, tipo primitivo de dado 175-176, 189-190, 192-193, 202-203
`DoubleArray.java` 345-346
`double-byte character set (DBCS)` 1339-1340
`DoWhileTest.java` 231-232
`draw` 587-590
`draw`, método 665-666
`draw3DRect` 579-580
`drawArc`, método 581-582, 599-600
`drawArcs.java` 582-583
`drawImage`, método de *Graphics* 967-968
`drawLine`, método de *Graphics* 149-151, 217-220, 376-377, 578-579, 599-600, 687-688
`drawOval`, método de *Graphics* 170-171, 231-232, 376-377, 579-580, 598-599, 676-677, 688-689
`drawPolygon`, método 584-587
`DrawPolygons.java` 585
`drawPolyline`, método 584-586, 601-602
`drawRect`, método de *Graphics* 154-156, 170-171, 376-377, 578-580, 583-584, 588-589, 590, 600-601, 688-689, 929-930

- drawRoundRect**, método de *Graphics* 579-581
DrawShapes.java 686-687
drawString, método de *Graphics* 144-145, 147-149, 153-156, 212-213, 376-377, 528-529, 568-569, 601-602, 929-930
DrawTest, *applet* 138-139, 140
driver de dispositivo, objeto 457-458
Driveway.com 1322
dump 369-370
dump de memória 369-370
dumpStack 771-772
duplicata de datagrama 912
duplicatas 1104-1105
duplo igual, == 114-116
duração 282-283
duração automática 282-283
durante a execução, erro de lógica 107-108
durante a execução, exceções 751
- E**
- “é um”, relacionamento 432-433, 441-442
E lógico booleano, & 238-239
E sobre *bits*, & 1058-1059
E/S termina 773-774
E/S, melhora de desempenho de 823
E/S, pedido de 772-773
E/S, término de 772-773
EAST 646-647, 715
ecoar um pacote de volta para o cliente 912
Edit 604
Edit, comando do DOS 61-62
editar 102-103
editar um programa 61-62
editável, *JTextField* 281
editor 62-63
editor, programa 61-62
eficiente (base de projeto do Unicode) 1337-1338, 1343-1344
“elaborar classes valiosas” 68-69
eLance.com 1324-1326
“elástico”, efeito de 665-666
elemento de um *array* 293-294, 317-318
elementos 1087
elements 1042, 1045-1046, 1050-1051
elevador 120-121
elevador, simulação de 120-121, 160, 246-247, 300-301, 652-653, 722-723
elevador, sistema de 123-124, 160-162, 652-653
elevados, retângulos 580-581
Elevator 161-166, 204-207, 247-250, 298-300, 347-351, 420-422, 489-493, 548, 551-552, 594-595, 657-658, 802-809, 1257-1259, 1263-1265, 1271, 1274-1275, 1281-1282, 1287-1289, 1306-1307
ElevatorButton 161-166, 204-207, 247-248, 298-300, 348-351, 420-422, 488-490
ElevatorConstants 654-655
ElevatorController 653-655, 722-726
ElevatorDoor 161-166, 204-207, 298-301, 348-351, 420-422, 489-490, 548
ElevatorModel 161-166, 203-207, 298-301, 348-349, 420-421, 491, 493, 594-595, 722-726, 807-808, 807-809, 1251, 1256-1258, 1270-1271, 1287-1289, 1309-1312
ElevatorModelEvent 548-552, 1242, 1249
ElevatorModelListener 722-726, 1245-1246, 1248-1249, 1256-1258, 1309-1310
ElevatorMoveEvent 549-550, 553, 1244-1245, 1249, 1310
ElevatorMoveListener 553, 594-596, 1245-1247, 1249, 1257-1258, 1264-1265, 1270-1271, 1273-1275
ElevatorMusic 1181-1183, 1190-1191, 1193-1194, 1306-1307, 1312
ElevatorMusic.java 1192-1193
ElevatorShaft 161-166, 204-207, 298-301, 347-351, 420-421, 489-491, 493, 551-552, 594-595, 803-804, 807-809, 1251, 1256-1258, 1260-1261, 1264-1265, 1270-1275, 1289, 1306-1307
ElevatorSimulation 722-726
ElevatorView 722-726, 1181-1183, 1186-1187, 1190-1191, 1249, 1270-1271, 1273-1274, 1287-1288, 1290, 1305-1312
elidido, diagrama 162-163, 592-595
eliminação de duplicatas 1012-1013
eliminar perda de recursos 757-758
ellipse 141-145, 169-170, 175-176, 578-579, 581-582, 664
ellipse delimitada por um retângulo 581-582
ellipse preenchida com cores que mudam gradualmente 589-590
ellipse, caixa delimitadora de uma 232
ellipse, símbolo 175-176
Ellipse2D 564
Ellipse2D.Double 586-587, 599-600
Ellipse2D.Float 586-587
else oscilante, problema do 179-180, 211-212
else, palavra-chave 175-176
em execução 771-772
em execução, estado 771-772, 776-777
em execução, *thread* 771-772
emacs 61-62
e-mail 905-906
e-mail, programa de 62-64
embaralhamento, algoritmo de 1094-1095
embaralhando 544-545
embaralhar um maço de cartas 1094-1095
empacotada, classe 1104-1105
empacotadores, métodos 1087
empilhados, blocos de construção 244, 246
empilhamento, regra de 243-244
empilhando 178, 246-247
Employee, hierarquia de classes 458-459
Employee.java 404-405, 415, 458-459
EmployeeTest.java 404-405, 415-416
empregado, número de identificação do 818-819
empregados 445-446
empregos de seis dígitos 1325-1326
EmptyListException.java 991-992, 995
EmptyStackException 1042-1043, 1045-1046
encadeada, lista 376-377, 989-991, 1085-1086, 1106-1107
encadeamento de objetos de fluxo 833-834
encadeando referências à superclasse 443-444
encapsulamento 67-68, 373-374, 380-382, 418-419, 455
End, tecla 640-641
EndOfFileException 839-840
endOfMedia, método de *String* 516-517
engenharia progressiva 420-423, 491, 493-494, 593-594
enqueue 420, 1004-1005
ensureCapacity, método de *StringBuffer* 529-530
Enter (ou Return), tecla 97-98, 138-141, 288-289, 335-336, 369-370, 611-612
ENTERED 900-901
entrada de uma estrutura de controle, ponto de 243-244
entrada única/saída única, estruturas de controle com 176-177, 242-244
entrada, campo de texto de diálogo de 154
entrada, cursor de 911-912
entrada, diálogo de 105, 107-110, 154, 184-185, 228-229, 278-279
entrada, dispositivo de 55
entrada, ponto de 176-177
entrada, unidade de 55
entrada/saída 754-755, 820-821
entrada/saída, operação de 175-176, 366-367
entrada/saída, pacote de 271
entrelinha 576-577, 578-579
Enumeration 1050-1053, 1057-1058, 1087
Enumeration, interface 1035, 1045-1046, 1078-1079
enviando uma mensagem para um objeto 217-218
enviar dados para o servidor 910-911

enviar mensagem 144-145, 153-154, 154
 enviou, objeto que 548
EOF (fim do arquivo) 901-902
EOFException 903-904, 908-909
equals 641, 1072, 1074, 1082-1085
equalsIgnoreCase, método de
String 514-516, 526
 erro de compilação 98-99
 erro detectado em um construtor 756-757
 erro durante a compilação 98-99
 erro por um 218-220, 318-319
 erro, código de tratamento de 740, 741-742
Error 743-744, 752
Error, hierarquia 752-753
 erros comuns de programação 59-60
 escala, fator de 272, 275-276
 escalar 330-331
 escalares, quantidades 330-331
 escalonado, inteiro aleatório 272-273, 275-
 276
 escalonador 773-774, 778-779
 escalonamento 772-773
 escalonar 272
 escape, caractere de 100-101
 escape, sequência de 100-101, 106-107,
 880-881, 1341-1342
 escopo 219-220, 282-283
 escopo de bloco 283-284, 380-381
 escopo de classe 283-284, 380-381
 escopo de um identificador 283-284
 escopo, exemplo de 284-285
 escrevendo o equivalente em palavras ao
 valor de um cheque 560-561
 escritor 815
 escritoras, *threads* 815
 esfera 306-307
 esgotada, memória 769-770
 esgotando a memória 288-289
 esgotar a memória livre disponível 769-770
 espaço, caractere 95-96
 espaço/tempo, compromisso entre 1047-
 1048, 860-861
 especial, caractere 100, 106-107, 509-510
 especial, símbolo 818-819
 especificação MIDI 1.0 completa detalhada
 1154-1155
 espera, cursor de 899-900
 espera, estado de 784-785, 800-801
 espera, fila de 419, 989
 espera, *thread* em 784-786
 esperando 772-773
 esperando, consumidor que está 778-779
 esperar por eventos 281, 614-615
 esperar uma nova conexão 901-902
 espiral 288-289, 601-602
 esquerda 564-565
 esquerda para a direita, avaliação 113-114
 esquerda, {, chave à 96-98, 105-106
 esquerda, alinhado à 609-610, 643-644
 esquerda, chave à, { 96-98, 105-106, 143-
 144

esquerda, filho à 1007-1008
 esquerda, operador de deslocamento à, <<
 1059-1060, 1070-1071
 esquerda, seta para a, tecla 681
 esquerda, subárvore 1007-1008, 1011-
 1012, 1020-1021
 esquerdo do mouse, botão 139
 esquerdo do mouse, clique no botão 638-
 640
 estação de trabalho 55-56
 estado 82-83, 124-125, 202-203, 246-247,
 818
 estado bloqueado 771-772
 estado consistente 377-380, 387-388, 391-
 392
 estado de nascimento 771-772
 estado morta 771-772, 776-777
 estado, barra de 141-143, 281-282, 664
 estado, objeto 730-731
 estado, transição de 82-83
 estados, diagrama de 246-247
 estados, diagrama de mapa de 82-83, 123-
 124, 246-249
 estática, duração 282-283
 estável, classificação 1093
 estendendo uma classe 473-474
 estouro 370, 740-741
 estouro aritmético 740-741
 estratégia 360-361
 estratégia, objeto 730-731
 estrela 590-591
 estrela de cinco pontas 590
 estrutura de coleções 1082, 1087
 estrutura de controle 174-175, 176-177,
 216, 246-247, 292-293
 estrutura do sistema 161-162, 164-165,
 205-206
 estrutura, padrões de projeto de 70-71, 494,
 496-497, 726-727, 957
 estruturada, programação 50-51, 61, 66-67,
 120-121, 157-159, 174-175, 216, 233-
 234, 246-247, 418-419
 estruturadas, técnicas 234-235
 estruturas de controle, aninhamento de
 176-177
 estruturas de controle, empilhamento de
 176-177, 243-244
 estudantes, programa de análise de
 pesquisa entre 327-328
 Euler 359
EventListenerList 615-616
 evento 83-84, 278-281, 548, 550-552, 566-
 567, 610, 1242, 1244-1245, 1248, 1256-
 1257, 1270-1273, 1287-1288
 evento de ação 281, 604-605, 668-669
 evento, classes de eventos 610
 evento, identificador de 616
 evento, objeto 611
 evento, *thread* de despacho de 787-788
EventObject 624-625, 713-714
 eventos assíncronos 740-741

eventos, classe adaptadora de *ouvinte* de
 635
 eventos, interface *listener* de 473-474, 553,
 611-612, 615-616, 631-632, 635, 640-
 641
 eventos, interfaces de tratamento de 487-
 488
 eventos, mecanismo de tratamento de 487-
 488
 eventos, método de tratamento de 611
 eventos, métodos de tratamento de 281
 eventos, movida a (a natureza de desenhar
 gráficos) 566-567
 eventos, movida a 368-369, 610
 eventos, origem de 611
 eventos, *ouvinte* de 550-553, 611-612, 635,
 685-686, 740-741
 eventos, processamento de 745-746
 eventos, processo movido a 566-567
 eventos, programação movida a 281
 eventos, registro de 614-615
 eventos, *thread* de tratamento de 787-788
 eventos, tratador de 279-280, 611
 eventos, tratamento de 83, 278-280, 471-
 472, 550-552, 610, 615-616, 1242, 1309-
 1310
EventType 900-901
 eWork® Exchange 1324-1326
 exame, problema dos resultados de 195-
 196
 exceção 329, 740-741
 exceção disparada e tratada por um **catch**
 local 760-761
 exceção não-capturada em um aplicativo
 de linha de comando 742-743
 exceção não-capturada em um *applet* de
 uma aplicação baseada em GUI 742-743
 exceção que ocorre em um tratador de
 exceção 746
 exceção verificada 752-753
 exceção, o próprio tratador de exceção
 dispara a mesma 766-767
 exceção, objeto 743-744
 exceção, subclasse 740-741
 exceção, superclasse 740-741
 exceção, superclasse de exceção e várias
 subclasses de 766-767
 exceção, tratador de 740-743, 750-751,
 756-757
 exceção, tratamento de 703-704, 740-741
Exception 740-741, 743-745, 757-759
 excluindo de uma pilha 418-419
 excluindo um item de uma árvore binária
 1013-1014
 exclusão de árvore binária 1020-1021
 execução, erro durante a 64-65
 execução, pilha de 745-746, 1000-1001
 executando cálculos 120-121, 157-159
 executando um *applet* como aplicativo
 689-690
 executar 98-99

- executar como *applet* ou como aplicativo 685-686
 executável, estado 771-772
 executável, *thread* 769-770
Exemplos
AccountRecord.java 827-828
Addition.java 105, 105-106
AdditionApplet.html 151-152
AdditionApplet.java 149-151
Algorithms1.java 1097-1098
Analysis.java 196-197
AnimatedPanel.java 1186-1187
 Atendendo mídia de fluxo com gerenciadores de sessão RTP 1136-1137
Average1.java 182-183, 190-191
BankUI.java 825-826
BinarySearch.java 337-338
BinarySearchTest.java 1099-1100
BitSetTest.java 1072, 1074
BorderLayoutDemo.java 646
Boss.java 459-460
BoxLayoutDemo.java 707-708
BreakLabelTest.java 235-236
BreakTest.java 233
BubbleSort.java 333-334
ButtonTest.java 617
ButtonText.java 617
CapturePlayer.java 1124-1125, 1132
CardDeck.java 711-712
Cards.java 1094-1095
CheckBoxTest.java 619-620
Circle.java 440-441, 444-468, 474-475
Client.java 906-907, 915-916
ClientGUI.java 947-948
 ClipPlayer reproduz arquivos de áudio gravados 1148
ClipPlayer.java 1148
ClipPlayerTest.java 1152-1153
CollectionTest.java 1087-1088
ComboBoxTest.java 624-625
CommissionWorker.java 460
Comparison.java 114-116
ConsumeInteger.java 780, 783-784
ContinueLabelTest.java 236-237
ContinueTest.java 233-234
Craps.java 275-276
CreateRandomFile.java 848-849
CreateSequentialFile.java 828-829
CreditInquiry.java 840
CustomPanel.java 673
CustomPanelTest.java 674
Cylinder.java 451-452, 468-469, 475-476
Date.java 402-403
DeckOfCards.java 544-545
DeitelMessenger.java 954-955
DeitelMessengerServer.java 931-932
DesktopTest.java 704-705
DivideByZeroException.java 746-747, 750-751
DivideByZeroTest.java 747-748
DoubleArray.java 345-346
DoWhileTest.java 231-232
DrawArcs.java 582-583
DrawPolygons.java 585
DrawShapes.java 686-687
ElevatorMusic.java 1192-1193
Employee.java 404-405, 415, 458-459
EmployeeTest.java 404-405, 415-416
EmptyListException.java 991-992, 995
FibonacciTest.java 289-290
FileTest.java 876-877
FlowLayoutDemo.java 643-644
Fonts.java 574-575
ForCounter.java 218-219
 Formatando e salvando mídia a partir de dispositivos de captura 1124-1125, 1132
GridBagDemo.java 715-716, 719
GridBagDemo2.java 719
GridLayoutDemo.java 648-649
HashtableTest.java 1047-1048
HoldIntegerSynchronized.java 785, 790-791
HoldIntegerUnsynchronized.java 780-781
HourlyWorker.java 461-462
ImageMap.java 977
ImagePanel.java 1183-1184
Increment.java 401
InitArray.java 320-323, 342-343
Interest.java 222-224
KeyDemo.java 640-641
LabelTest.java 607-608
LinearSearch.java 335-336
LinesRectsOvals.java 579-580
List.java 992-993
ListTest.java 995-996, 1089-1090
LoadAudioAndPlay.java 980-981
LoadImageAndScale.java 967-968
LogicalOperators.java 239-241
LogoAnimator.java 970-971
LogoAnimator2.java 974-975
LogoApplet.java 975-976
LookAndFeelDemo.java 700-701
MapTest.java 1102-1103
MenuTest.java 691-692, 694
MessageListener.java 934-935
MessageManager.java 940-941
MethodOverload.java 295-297
Metrics.java 577-578
 MidiData carrega arquivos MIDI para reprodução 1155
MidiData.java 1155
MidiDemo.java 1167
 MidiRecord permite que um programa grave uma sequência MIDI 1160-1161
MidiRecord.java 1160-1161
 MidiSynthesizer pode gerar notas e enviá-las para um outro dispositivo MIDI 1162-1163
MidiSynthesizer.java 1162-1163
MiscBitOps.java 1061-1062
MouseDetails.java 637-638
MouseTracker.java 633-634
MovingPanel.java 1185-1186
MulticastSendingThread.java 938-939
MultipleSelection.java 629-630
OtherCharMethods.java 540-541
OvalPanel.java 681-682
PackageDataTest.java 406-407
PacketReceivingThread.java 944-945
Painter.java 635-636
PanelDemo.java 622, 650-651
PassArray.java 330-331
PieceWorker.java 460-461
Point.java 437, 442-443, 447-448, 466-467, 472-473
PopupTest.java 697-698
PrintBits.java 1059-1060
ProduceInteger.java 779-780, 782-783, 788-790
PropertiesTest.java 1053-1054
QueueInheritance.java 1004-1005
QueueInheritanceTest.java 1005-1006
RadioButtonTest.java 622
RandomAccessAccountRecord.java 847

R
RandomCharacters.java 797
RandomIntegers.java 272-273
ReadRandomFile.java 855-856
ReadSequentialFile.java 834-835
ReadServerFile.java 896-897
ReceivingThread.java 935-936
RTPServer.java 1136-1137
RTPServerTest.java 1142-1143
Scoping.java 284-285
SelfContainedPanel.java 675-676
SelfContainedPanelTest.java 678-679
SendingThread.java 943-944
Server.java 902-903, 912
SetTest.java 1101
Shape.java 465-466, 472-473
Shapes.java 587-588
Shapes2.java 590-591
SharedCell.java 781-782, 786-787, 794-795
ShowColors.java 568-569
ShowColors2.java 570
SimplePlayer.java 1114-1115
SiteSelector.html 892-893
SiteSelector.java 894-895
SliderDemo.java 682-683
SocketMessageManager.java 940-941
SocketMessengerCons-tants.java 933-934
Sort1.java 1093
Sort2.java 1093-1094
SortedSetTest.java 1101-1102
SoundEffects.java 1191-1192
StackComposition.java 1003-1004
StackInheritance.java 1001-1002
StackInheritanceTest.java 1002-1003
StaticCharMethods.java 536
StaticCharMethods2.java 538-539
StringBufferAppend.java 532-533
StringBufferCapLen.java 529-530
StringBufferChars.java 530-531
StringBufferConstruc-tors.java 528-529
StringBufferInsert.java 534-535
StringConcat.java 521-52
StringConstructors.java 509-510
StringHashCode.java 518-519
StringIntern.java 526
StringMisc.java 512-513
StringStartEnd.java 516-517
StringValueOf.java 524-525
StudentPoll.java 327-328
SubString.java 521
Sum.java 222-223
SumArray.java 323-324
SwitchTest.java 227
Test.java 440-441, 444-445, 448-450, 452-453, 462-463
TestFieldTest.java 611-612
TextAreaDemo.java 668-669
TextFieldTest.java 611-612
ThisTest.java 407-408
ThreadTester.java 775
TicTacToeClient.java 924-925
TicTacToeServer.java 918-919
Time.java 477-478
Time1.java 374-375, 383-384
Time2.java 387-388
Time3.java 391-392
Time4.java 409-410
TimeTest.java 378
TimeTest2.java 381-382
TimeTest3.java 385
TimeTest4.java 389-390
TimeTest5.java 395-396
TimeTest6.java 412-413
TimeTestWindow.java 479-480, 484
TokenTest.java 541-542
Tree.java 1007-1009
TreeTest.java 1010-1011
UpdateThread.java 788-789
UsingArrays.java 1082-1083
UsingAsList.java 1085-1086
UsingExceptions.java 758-760, 762
UsingToArray.java 1091-1092
VectorTest.java 1035-1036
Welcome1.java 94-95
Welcome2.java 100
Welcome3.java 100-103
WelcomeApplet.html 146-147
WelcomeApplet.java 142
WelcomeApplet2.html 148-149
WelcomeApplet2.java 148-149
WelcomeLines.htm 149-150
WelcomeLines.java 149-150
WhileCounter.java 216-217
WriteRandomFile.java 851-852
 exibição, área de 146-147
 exibindo dados na tela 157-159
 exibindo múltiplos *strings* 148-149
 exibir 1181-1182, 1190-1191
 exibir a saída 100-102, 120-121, 123-124
e *exists* 876
 exit, método de *System* 104-105, 183-184
E *EXITED* 900-901
exp, método de *Math* 262-263
 expandido, submenu 695
 expedição do computador, seção de 55
E*xperience.com* 1325-1326
 explícita, conversão 192-193
 explícito da referência *this*, uso 411-412
 exponenciação 110-111, 371
 exponenciação, operador de 224-226
 exponencial, “explosão” de chamadas 292-293
 exponencial, método 262-263
 expressão booleana 178-179
 expressão condicional 178-179, 376-377
 expressão de controle 228-229
 expressão integral constante 229-231
 expressões de atribuição abreviada 198-199
e *extends* 431-432, 435-436
e *extends JApplet* 143-144, 152-153
e *extends*, palavra-chave 143-144, 175-176, 278-279
 extensão, pacote de 100-102
 extensibilidade 456-457
 extensibilidade de Java 740
 extensível 453-454
 extensível, linguagem 288-289, 377-378
 externa, referência *this* da classe 488-489
 externa, referência *this* para classe 800-801
 externo, bloco 283-284
 externo, bloco *try* 759-760
 externo, evento 631-632, 668-671
 extremidades de uma linha 149-151

F

fábrica 956-957
 fábrica, método 726-727
 fabricação do computador, seção de 55
F*acade*, padrão de projeto 494-495, 497-498, 957
 fachada, objeto de 957
factoria, método 286-287
Factoria**T**est.java 286-287
Factory Method, padrão de projeto 494-496, 726-727
 Fahrenheit 664
 Fahrenheit de uma temperatura em Celsius, equivalente 311-312
 falando para computadores 55
 falha no construtor 766-767
 falhas, tolerante a 107-108, 740
false, palavra-chave 175-176, 177
 falsidade 113-114
 fatal, erro 64-65, 180-181, 370
 fator de carga *default* 1047-1048
 factorial 214, 253-254, 285-286
 factorial, método 286-287

- faxina 413-414
 fazendo referência a um objeto de subclasse com uma referência para subclasse 445-446
 fazendo referência a um objeto de subclasse com uma referência para superclasse 445-446
 fazendo referência a um objeto de superclasse com uma referência para superclasse 445-446
 fazer seu ponto 275-276
 fechando uma janela 610
 fechar um arquivo 840, 854-855
 fechar um diálogo 104-105, 110
 fechar uma janela 482-483
 ferramenta, dicas sobre a 607-610
 ferramentas, barras de 619-620
 Fibonacci definida recursivamente, série de 288-289
fibonacci, método 292
 Fibonacci, série de 288-289, 292, 371
FibonacciTest.java 289-290
FIFO 419
FIFO, ordem 420
 fila 376-377, 419, 989, 1004-1005
 fila para o servidor 901-902
File 820-821, 824-825, 833-834, 839-840, 875-876, 878-879
File OutputStream 840
FileInputStream 820-824, 833-835, 839-840, 842-843, 875-876, 1055-1058
FileOutputStream 820-824, 831-834, 875-876, 957, 1055-1057
FileReader 820-821, 824-825, 880-881
FILES_AND_DIRECTORIES 828-829
FILES_ONLY 828-829, 830-831
FileTest.java 876-877
FileTypeDescriptor 1133-1134
FileWriter 820-821, 824-825
 filha 1007-1008
 fill 587-593, 601-602, 1082-1084, 1093, 1096-1098
fill3DRect 579-580
fillArc 581-584
fillOval 579-582
fillOval, método 636-637
fillPolygon 584-587
fillRect, método de **Graphics** 310-311, 568-570, 579-580, 588-590
fillRoundRect, método de **Graphics** 579-581
FilterInputStream 822-823
FilterOutputStream 822-823
FilterReader 823-824
 filtragem, fluxo de 822-823
 fim da entrada de dados 187-188
 fim de arquivo 901-902
- fim de arquivo, marcador de 839-840, 859-860
 fim de fluxo 901-902
 fim de um laço 232
 finais, caracteres de espaçamento 524-525
 final de variável de controle, valor 216, 218-222
final, classe 455
final, palavra-chave 175-176, 229-231, 401, 455
 final, valor 217-218
final, variável 279-280
 final, variável 323-324
final, variável local 483
finalize, método 379-380, 413-416, 442-445, 756-757, 766-767
finally, bloco 175-176, 742-744, 757-758, 760-761
first 1101-1102
 física: bola que pula, demonstração de 1204-1205
 física: cinética, demonstração de 1204-1205
 física: cinética, exercício de demonstração de 1204-1205
 físicas, operações de entrada 823
 físicas, operações de saída 823
 fita magnética 818
 fixa, fonte de largura 341-342
 fixo, registro de tamanho 846
flipDog.com 1314-1315
 fliperama, exercício da máquina de 1205
 fliperama, máquina de 1205
Float 488-489, 1042
float, promoções 269-270
float, tipo primitivo de dado 106-107, 149-151, 175-176, 202-203
floor 161-166, 203-207, 298-300, 420-421, 489-491, 493, 653-654, 656-658, 805-809, 1251, 1256-1261, 1274-1275, 1281-1282, 1287-1289, 1306-1307
floor, método de **Math** 262-263, 306-307
FloorButton 161-166, 204-207, 247-248, 298-301, 347-351, 420-421, 488-489
FloorDoor 161-166, 204-207, 298-301, 347-351, 420-421, 489-490
FlowLayout 278-279, 288-289, 619-620, 634-635, 642-643, 645-646, 707
FlowLayout.CENTER 644-646
FlowLayout.LEFT 644-646
FlowLayout.RIGHT 645-646
FlowLayoutDemo.java 643-644
flush 823, 833-834, 904-906, 908-910
 flutuante, divisão em ponto 192-193
 flutuante, número em 149-151, 153-154, 191-192
 flutuante, constante em ponto 224-225
 fluxo de bytes 819-820
- fluxo de controle 120-121, 157-159, 182-183, 191-192
 fluxo de entrada 187-188
 fluxo, cabeçalho de 906
 fluxo, linha de 174-175
 fluxo, mídia de 1135-1136
 fluxo, objeto 820-821
 fluxo, processamento de 818
 fluxo, soquete de 891-892, 901-902, 918
 fluxograma 174-175, 177, 248-249
 fluxograma da estrutura de controle **for** 220-222
 fluxograma da estrutura de repetição **do/while** 232
 fluxograma da estrutura de repetição **while** 182-183
 fluxograma da estrutura de seleção dupla **if/else** 178
 fluxograma da estrutura de seleção simples **if** 177
 fluxograma que pode ser reduzido para o fluxograma mais simples 244, 246
 fluxograma, símbolos de 242-243
 fluxos 891-892
 fluxos, transmissão baseada em 911-912
Flyweight, padrão de projeto 497-498
 foco 611-612, 681, 700
FocusAdapter 635
FocusListener 635
 folha 728-729
 folha em uma árvore de pesquisa binária, nodo 1011-1012
 folha, nodo 1007-1008
Font 564, 574-575, 620-621
Font, classe 338-339, 341-342
Font.BOLD 341-342, 573-575, 577-578, 620-621, 695
Font.ITALIC 341-342, 573-576, 620-621, 695
Font.PLAIN 341-342, 345-346, 573-576, 620-621, 694
 fonte 341-342, 564, 572-573, 577-578
 fonte *default* 574-575
 fonte, estilo de 573-574, 619-620
 fonte, nome de 573-574
 fonte, tamanho de 573-574
 fontes, controle de 572-573
 fontes, manipulação de 341-342, 565-566
 fontes, métricas de 576-577
FontMetrics 564, 576-578
Fonts.java 574-575
for, estrutura de repetição 175-176, 218-226, 228-229, 242-243, 246-247
 fora dos limites, subscrito de *array* 740-741, 751
 força bruta 361-362
 força bruta, versão do passeio do cavalo 361-362
ForCounter.java 218-219
forDigit 538-539
 forma 140, 586-587

forma animada 140-141
 forma *default* 140
 forma, classes de 427-428
 forma, ponto, círculo e cilindro, hierarquia de 465-466
 formando programas estruturados 243-244
 formas 138-139
Format 1132-1133, 1141-1142
format, método de **DecimalFormat** 224-226
 formatado, número em ponto flutuante 192
FormatControl 1132-1133
 formato 376-377
 formato, indicador de 192
 formato, *string* de controle de 377-378
 fornecedor de serviço de aplicativo (ASP) 1320-1321
 Forté for Java Community Edition 61-62
 fortemente tipadas, linguagens 202
 Fowler, Martin 70
 fracionário, resultado 192-193
 fractal 138-139
Fractal, *applet* 138-139
frame 1150-1151
Frame 680, 684-685
frames 1150-1151, 1196-1197
frames, sequência de 1189-1190
 FreeAgent 1324-1326
 freeware 59-60
 função 67-68
 função, tecla de 640-641
 funcional, decomposição 54
 fundo da janela do aplicativo 678-679
Futurestep.com 1320-1321

G

garabito de conversão 157-159
 Gamma, Erich 70-71
 "gangue dos quatro" 70-71, 494, 496-498, 809-810
 Gates, Bill 60-61
 gaussiana, distribuição 1058-1059
gc, método de **System** 417-418, 445-446
 generalização 83, 489-490
 generalizações, diagrama de 489-490, 592-593
GeneralPath 564, 590, 600-601
 genérico, caminho 590
 gerador de jogos de palavras cruzadas 562
 gerando labirintos aleatoriamente 365
 gerenciador 736-737
get 893-894, 1045-1046, 1071-1072, 1087, 1088-1089, 1103-1104
get, método 381-382, 391, 394-395
getAbsolutePath 876-878
getActionCommand 613-614, 617-620
getActionCommand, método de **ActionEvent** 339-340
getAddress 913-915

getAppletContext, método de **Applet** 896
getAscent, método 576-577
getAudioClip, método 980-981
getAudioInputStream, método 1150-1151
getAvailableInstruments, método 1165-1166
getBlue, método 567-570
getByName 908-909
getChannels, método 1165-1166
getClass, método de **Object** 953
getClassName, método de **UIManager.LookAndFeelInfo** 703-704
getClickCount 638-639
getCodeBase, método 980-981
getColor, método 567-570
getCommand, método 1159-1160
getComponent 698-699
getContentPane, método 617-618, 694, 698-699
getContentPane, método de **JApplet** 263-264, 280-281, 331-332, 345-346
getControl 1134-1135
getControlComponent, método de **Control** 1134-1135
getCurrencyInstance, método de **NumberFormat** 224-225
getCurrentTimeMillis, método de **System** 292-293
getData 913-914, 914-915
getData1, método 1159-1160
getData2, método 1159-1160
getDescent, método 576-577
getDeviceList, método de **CaptureDeviceManager** 1133
getDocumentBase, método 969-970, 979-980
getEventType, método de **HyperlinkEvent** 900-901
getFamily, método 573-576
getFieldValue 854-855
getFont, método 573-577, 695
getFontMetrics, método 576-578
getFormatControl, método 1133
getFrameSize, método 1151-1152
getGreen, método 567-569, 570
getHeight, método 576-578, 983-984
getHostName, método de **InetAddress** 904-906, 908-909
getIconHeight 705-706
getIconWidth 705-706
getImage, método 983-984
getInetAddress 904-906, 908-909
getInputStream 900-902, 904-905, 908-909, 921-922, 925-926
getInsets 564-565
getInstalledLookAndFeel 701-702
getInstalledLookAndFeels, método de **UIManager** 701-704
getKeyChar 641-643
getKeyCode 641-642
getKeyModifiersText 641-643
getKeyText 641-642
getLeading, método 576-578
getLength 913-915
getLine, método 1151-1152
getLocalHost 911-912, 915-917
getLocator, método de **CaptureDeviceInfo** 1133
getMaxPriority 802
getMessage 761-762
getMessage, método 1159-1160
getMinimumSize 681-682, 929-930, 973-974
getModifiers 641-643
getName 575-576, 695, 771-772, 779-780, 783-784, 802, 876-877
getName, método 573-574
getOutputStream 900-901, 904-905, 908-909, 921-922, 925-926
getParameter 892-893, 895-896, 973-974
getParent 802, 876-877
getPassword 613-614
getPath 876-877
getPort 913-915
getPredefinedCursor 898-900
getPreferredSize 676-678, 681-682, 929-930, 973-974
getPriority 773-774
getProperty 1054-1057
getRed, método 567-570
getResource, método de **Class** 953
getSelectedFile 833-834, 836-837, 842-843, 849-850, 853-854, 857-858
getSelectedIndex 625-626, 628-629
getSelectedText 671-672
getSelectedValues 630-632, 894-895
getSequencer, método de **MidiSystem** 1158-1159
getSize 573-576
getSource 311-312, 394-395, 615-616, 620-622, 646-647, 697-698, 712-713
getStateChange 620-621, 625-626
getStyle, método 573-574
getTargetFormats, método 1151-1152
getText 697-698, 917-918, 927-928, 1054-1055
getTick, método 1159-1160
getTracks, método 1159-1160
getURL, método de **HyperlinkEvent** 900-901
getValue 683-684
getX 633-634, 635, 675-676, 679-680, 698-699

- getY** 633-635, 676-677, 679-680, 698-699
GIF 609-610, 969-970
.gif 609-610, 969-970
Gosling, James 58-59
goto, comando 174-175, 244, 246
goto, eliminação de 174-175
grade 648-649, 714
gradiente 587-588
GradientPaint 564, 587-588, 600-601
gráfica de um algoritmo, representação 174-175
gráfica de uma lista encadeada, representação 992-993
gráfico de barras 137-138, 254-255, 324-325, 601-602
gráfico, contexto 565-566
gráfico, informações em um 324-325
gráficos 51-54, 137-141, 1182-1183, 1312
gráficos de uma maneira independente de plataforma 565-566
gráficos, demonstração de 140-141
grafo 138-139, 254-255
Grand, Mark 809-810
Graphics 505-506, 565-566, 589-590, 967-970
Graphics Interchange Format (GIF) 609-610, 969-970
Graphics, classe 141-145, 149-154, 217-219, 232, 298, 376-377, 427-428, 586-587, 665-666
Graphics2D 586-587, 589-590, 592-593, 599-601
GraphicsEnvironment classe 341-342
GraphicsTest, applet 138-139
GraphLayout, applet 138-139
grau 581-582
gravando dados aleatoriamente em um arquivo de acesso aleatório 851-852
gravável 876
GridBagConstraints 714-716, 718-719
GridBagConstraints, variável de instância 714
GridBagConstraints.BOTH 715, 716-717
GridBagConstraints.HORIZONTAL 715
GridBagConstraints.RELATIVE 719-720
GridBagConstraints.REMAINDER 719-720
GridBagDemo.java 715-716
GridBagDemo2.java 719
GridBagLayout 707, 714-716, 718-719
gridheight 714, 716-719
GridLayout 642-643, 648-649, 674-675, 686-687, 700-701, 707, 712-713
GridLayoutDemo.java 648-649
gridwidth 714, 716-719
gridx 714, 716-717
gridy 714, 716-717
grossas, linhas 587-588
grossas, linhas brancas 590
grupo de *threads*-filhas 801
.gsm (extensão de arquivo GSM) 1113
Guarded Suspension, padrão de projeto 494-495, 809-810
GUI 51-52, 120-121, 653-654
GUI que gerou um evento, componente de 394-395
GUI, aplicativo baseado em 685-686
GUI, componente de 137-138, 224-226, 604
GUI, ferramentas para projeto de 280-281
GUI, tratamento de eventos de 391-392, 471-474, 483
- ## H
- habilitar** 445-446
handle 380-381
hardware 50-51, 54, 56-57
hardware, plataforma de 57-58
hash, código de 518-519
hash, colisões em uma tabela de 1046-1048
hash, fazendo 1035, 1046-1047
hash, recipiente de 1047-1048
hash, tabela de 518-519, 541-542, 1046-1047, 1100-1101
hashCode, método 518-519
HashMap, classe 1102-1104
HashSet, classe 1100-1101
Hashtable 893-895, 1035, 1047-1048, 1052-1053, 1082, 1106-1107
HashtableTest.java 1047-1048
hasMoreElements 1044-1045, 1050-1051, 1055-1056
hasMoreTokens, método de
 StringTokenizer 543-544, 556-557
hasNext 1088-1092, 1101-1102
Headhunter.net 1322
headSet 1101-1102
Helm, Richard 70-71
Help, link 1238-1240
helpdoc.html 1238-1240
Helvetica, fonte 341-342
herança 67-68, 83, 143-144, 163-164, 278-279, 373-374, 375-376, 380-381, 431, 434-435, 446-447, 453-454, 488-493, 1259
herança com exceções 756-757
herança de interface *versus* herança de implementação 504-505
herança e tratamento de exceções 745-746
herança *versus* composição 503-504
herança, exemplos de 433-434
herança, hierarquia de 434-435, 456-457, 502-503
herdada, interface que pode ser 465
herdar implementação 504-505
herdar interface 455, 504-505
herdar interface e implementação 465
herdar variáveis de instância 436-437
heurística 360-361
heurística de acessibilidade 360-361
hexadecimal (base 16), sistema de numeração 254-255, 371, 1217
Heymans, F. 815
HIDE_ON_CLOSE 684-685
hierarquia de classes 456-457
hierarquia de coleções 1087
hierarquia de formas 454-455, 465
hierarquia profunda 504-505
hierarquias de classes encabeçadas por superclasses abstratas 454-455
hierarquias projetadas para herança de implementação 465-466
hierárquica, estrutura 260-261
hierárquico entre método trabalhador e o método patrão, relacionamento 260-261
hierárquico, diagrama 434-435
hieróglifo 1338-1339
hipotenusa de um triângulo retângulo 309-310
Hire.com 1320-1321
HireAbility.com 1323-1324
Hirediversity.com 1319-1320
Histogram.java 324-325
histograma 254-255, 324-325
histórias, contador de 1204-1205
histórias, exercício do contador de 1204-1205
Hoare, C. A. R. 777-778, 815
HoldIntegerSynchronized.java 785, 790-791
HoldIntegerUnsynchronized.java 780-781
Home, tecla 640-641
horista, trabalhador 738
HORIZONTAL 681-682, 716-717
horizontal, cola 708-709
horizontal, componente **JSlider** 680
horizontal, coordenada 564
horizontal, espaçamento 647-648
horizontal, orientação 681
horizontal, regras para barra de rolagem 671-672
horizontal, tabulação 100-101
HORIZONTAL_SCROLLBAR_ALWAYS 671-672
HORIZONTAL_SCROLLBAR_AS_NEEDED 671-672
HORIZONTAL_SCROLLBAR_NEVER 671-672
host de um servidor, nome de 911-912
HotDispatch.com 1323-1324
HotJobs.com 1316-1318, 1322

- HotSpot**, compilador 66
HourlyWorker.java 461-462
HTML (Hyper Text Markup Language) 1229-1231, 1234
HTML (Hypertext Markup Language) 135-136, 145-147
HTML Converter 156-157
HTML, arquivo 138-139, 147-148, 1237-1238
HTML, documentação 1229-1231, 1234
HTML, documento 140-141, 216, 218-219, 298, 685-686, 891-893, 1237-1240
HTML, frame 896
HTML, marca de 146-147
HTTP (HyperText Transfer Protocol) 892-893
HugeInteger 426-427
Hyperext Transfer Protocol 892-893
hyperlink 159-160, 896-897, 899-900, 1235-1238
hyperlink Applets 159-160
HyperlinkEvent 896-897, 899-900
HyperlinkListener 897-900
Hypertext Markup Language (HTML) 62-64
-
- I**
- IBM** 55-56, 60-61
IBM Corporation 1337-1338
IBM Personal Computer 55-56
Icon 608-609, 625-626
ícone 109
identificador 95-96, 176-177
identificador, duração de um 282-283
identificador, escopo de um 282-283
identificar as classes 160
IDEs 66
IEEE 696-697, ponto flutuante 202-203
if, estrutura de seleção simples 113-114, 117-118, 175-178, 226, 242-243, 246-247
if/else, estrutura de seleção dupla 175-176, 178, 189-190, 195-196, 226, 242-243, 246-247
ignorando o elemento zero 329
igual probabilidade 273-274
igualdade, operador == para comparar objetos **String** 514
IllegalArgumentException 773-774, 827-828
IllegalMonitorStateException 778-779
IllegalThreadStateException 771-772, 795-796
Image 967-970
ImageIcon 608-609, 617-618, 625-626, 967-968, 969-970, 973-974, 977, 1189-1190
imagem 51-54, 298, 601-602, 967-968, 982, 1181
ímagem com pontos ativos 138-139
íagem, mapa de 967-968, 977
ImageMap, applet 138-139
ImageMap.java 977
ImageObserver 969-970
ImagePanel 1181-1183, 1306-1310, 1312
ImagePanel.java 1183-1184
imaginária, parte 425-426
implementa **ActionListener** 281
implementa uma interface 473-474
implementação 380-381, 1290
implementação de coleção 1082-1083
implementação de uma classe ocultada de seus clientes 379-380
implementação *default* de cada método em uma classe adaptadora 635
implementação, classes de 1082
implementação, código dependente de 380-381
implementação, detalhes da 373-374
implementação, fase de 491, 493-494, 1194-1195
implementação, herança de 465-466
implementação, processo de 81-85, 299-300, 420, 594-595
implementações abstratas 1105-1106
implementar 553, 593-594, 594-596
implementar muitas interfaces 473-474
implements 175-176, 473-474
implícita de objeto de subclasse para objeto de superclasse, conversão 445-446
implícita, conversão 192-193
import, instrução 100-102, 104-105, 141-143, 151-152, 175-176, 264-265, 270-271, 378, 382-383, 1234-1235
importar o pacote inteiro 151-152
impressão, *spool* de 1004-1005
impressora 64-65
imprimindo árvores 1021
imprimindo datas em vários formatos 559-560
imprimindo uma árvore binária em um formato de árvore bidimensional 1013-1014
imprimir em múltiplas linhas 99, 100
imprimir um *array* 363-364
imprimir um *array* recursivamente 363-364
imprimir um histograma 324-325
imprimir uma linha de texto 97-98
imutável 415-416, 511-512
inanição 773-774
inconsistente, estado 742-743
Increment.java 200-201, 401
incrementar 216, 219-220
incrementar uma variável de controle 217-218, 221-222
incremento da capacidade 1035-1036, 1041-1042
incremento de uma estrutura **for** 220-221
incremento de uma variável de controle 218-220
incremento e decremento, operadores de 199-200
incremento, expressão de 233-234
incremento, operador de, ++ 199-200
indefinida, repetição 187-188
indefinidamente, adiar um escritor 815
indefinido, adiamento 771-774, 815
independente, fornecedor de *software* 59-60
“independentes”, unidades 55-56
index.html 1238-1240
index-all.html 1241
indexOf 1041-1042
IndexOutOfBoundsException 1096-1097
indicador, valor de 187-188
índice 317-318
indireta, classe base 566-567
indireta, superclasse 434-435, 468-469
InetAddress 908-909, 915-916, 925-926, 939-940, 944-945, 957-958, 1142-1143
InetAddress 911-912, 914-915
infinita, recursão 288-289, 292-293
infinita, série 254-255
infinito, laço 181-182, 192, 219-220, 231-232, 288-289, 914-915, 918
infixa para pós-fixa, algoritmo de conversão de 1017-1018
infixa, notação 1017-1018
informações com “escopo de classe” 414
informações, camada de 959-960
informações, ocultação de 67-68, 373-374, 381-382, 407-408, 418-419, 432-433, 455
INFORMATION_MESSAGE 405-406
inicial de um atributo, valor 205-206
inicial de variável de controle, valor 216, 218-222
inicial, ângulo 581-582
inicial, conjunto de classes 373-374
inicial, estado 247-248
inicial, valor 216-217
initialização 216-217, 219-220
initialização da estrutura **for**, seção de 222-224
initialização de um *applet* 298
initialização no início de cada repetição 198-199
initialização, fase de 189-190
inicializadores 386-387
inicializadores, lista de 321-322, 342
inicializadores, sublista de 342
inicializando *arrays* bidimensionais em declarações 342-343
inicializando *arrays* multidimensionais 342-343
inicializando objetos de classe 376-377

- inicializando um *array* com uma declaração 322-323
- initializar contadores e totais 184-185
- initializar implicitamente com valores *default* 386-387
- initializar variáveis de instância 379-380
- initializar variáveis de instância do *applet* 153-154
- iniciar uma ação 690-691
- init**, método de *JApplet* 144-146, 153-156, 263-264, 288-289, 298, 333-334, 343-344, 689-690
- InitArray.java** 320-323, 342-343
- initialize**, método de *RTPManager* 1142-1143
- inline*, colocar 455
- inorderTraversal** 1010-1011
- InputEvent** 632-633, 639-641
- InputStream** 822-824, 900-902, 937, 1057-1058
- inserção, ponto de 991-992, 1085-1086, 1098-1099
- inserindo em uma pilha 418-419
- inserir um item em um objeto contêiner 376-377
- insert**, método de *StringBuffer* 533-534
- insertElementAt** 1041-1042
- Insets** 564-565
- instanceof**, operador 175-176, 442
- instância 143-144
- instância de um tipo definido pelo usuário 373-374
- instância de um tipo primitivo 373-374
- instância, variável de 152-153, 282-284, 375-376, 391, 409, 436-437, 823-824
- instanciando um objeto 192-193
- instanciar (ou criar) objetos 143-144, 373-374
- instanciar 67-68, 349-350, 1256-1257
- instantâneo, aplicativos de acesso 846
- instrução 98-99, 265-266
- instrução auxiliada pelo computador (CAI) 312-313
- instrução de atribuição 107-108, 154
- instrução, terminador de, (;) 98-99
- Instrument** 1162-1163
- instrumento, número de programa de 1166
- int** 106-107
- int**, promoções 269-270
- int**, tipo primitivo de dado 175-176, 189-190, 199-200, 226
- Integer** 108-109, 154-155, 185-186, 484, 1005-1006, 1042, 1103-1104
- integerPower**, método 309-310
- integrados, ambientes de desenvolvimento (IDEs) 61-62
- integrais, valores 226
- integral, expressão 229-231
- inteira, divisão 110-111, 192-193
- inteiro 105
- inteiro binário 213-214
- inteiro, quociente 110-111
- inteiro, valor 106-107
- inteiros grandes 288-289
- inteiros, *array* de 321-322
- inteiros, matemática com 418-419
- Intel 1113
- inteligente, agente 1314-1315
- inteligentes, dispositivo de consumo eletrônicos 58-59
- interação 203-205
- interações entre objetos 347-351, 418-419
- interações, diagrama de 349-350, 805-806
- intercalação, classificação por 1093
- Interest.java** 222-224
- interface 67-68, 83, 278-279, 381-382, 431-432, 471-474, 553, 592-596, 1242, 1245-1246, 1248, 1256-1257, 1273-1275, 1281-1282
- interface** 451-452, 471-474
- interface *ActionListener* 614-615
- interface *CaptureDevice* 1133
- interface comum para os membros de uma hierarquia de classes 456-457
- interface de programação de aplicativo (API) 59-62
- interface gráfica com o usuário (GUI) 51-52, 102-103, 105, 120-121, 123-124, 270-271, 394-395, 604, 653-654
- interface gráfica com o usuário (GUI), componente da 102-103
- interface *ItemListener* 621-622
- interface *LayoutManager* 643-644
- interface *MouseListener* 665-666
- interface, herança de 465-466
- interface, palavra-chave 175-176
- interfaces da estrutura de coleções 1082
- interfaces de coleção 1082-1083
- intern**, método de *String* 526-527
- interna de dados, representação 420
- interna para tratamento de eventos, classe 477-478, 535-536
- interna que pode ser fechada, *frame* 707
- interna que pode ser maximizada, *frame* 707
- interna que pode ser minimizada, *frame* 707
- interna que pode ser redimensionada, *frame* 707
- interna, classe 477-478, 535-536, 572, 621-622, 626-627, 636-639, 680, 683-684, 696-697, 703-704, 900-901
- interna, definição de classe 477-478
- interna, objeto de classe 483
- internacionalização 271
- International Standards Organization (ISO) 52-53, 57-58, 419
- Internet 50-52, 61, 136-137, 159-160, 892-893, 969-970
- Internet and World Wide Web How to Program, Second Edition* 62-64
- Internet do servidor, endereço na 911-912
- Internet e na Web, recursos na 982, 1181
- Internet em ordem inversa, nome de domínio na 384-385
- Internet, endereço na 914-915
- interno, bloco 283-284
- internos, colchetes mais 328-329
- internos, parênteses mais 111-112
- Internshipprograms.com** 1326-1327
- interpretador 62-66, 94, 97-98, 186-187, 222-223
- Interpreter**, padrão de projeto 497-498
- interrupção 772-773
- interrupt** 771-772, 801, 806-807
- InterruptedException** 752, 776-777, 779-780, 783-785, 789-791, 920-921
- InterruptedIOException** 937, 947
- interseção de dois conjuntos 427-428
- intervalo, verificação de 381-382
- InterviewSmart** 1326-1327
- inválida, conversão 752
- inválida, referência para *array* 329
- InvalidMidiDataException** 1158-1159, 1166
- InvalidMidiException** 1158-1159
- inválido, subscrito de *array* 744-745
- InvalidSessionAddressException** 1142-1143
- invocando um método 144-145
- invocar 261-262
- invokeLater**, método de *SwingUtilities* 787-788, 800-801, 953-954
- IOException** 752, 754-755, 833-834, 837-838, 850-851, 908-909
- IOExceptions** do pacote *java.net* 755-756
- irmão 1007-1008
- isAbsolute** 876-878
- isActionKey** 641
- isAlive** 771-772
- isAltDown** 638-641
- isBold** 573-574, 576-577
- isControlDown** 642-643
- isDaemon** 795-796
- isDefined**, método de *Character* 536-537
- isDigit**, método de *Character* 536-538
- isDirectory** 876
- isEmpty**, método 376-377, 1041-1042, 1045-1046, 1052-1053, 1103-1104
- isFile** 876-878
- isFull** 376-377
- isInterrupted** 771-772
- isItalic**, método 573-574, 576-577
- isJavaIdentifierPart**, método de *Character* 536-538

- isJavaIdentifierStart**, método de *Character* 536-538
isLetter, método de *Character* 536-538
isLetterOrDigit 536-537
isLetterOrDigit, método de *Character* 537-538
isLowerCase 536-538
isMetaDown 638-641
ISO 52-53, 57-58, 419
isPlain, método 573-574, 576-577
isPopupTriggered 698-700
isRunning 973
isSelected 694, 697-698
isSelectable, método de *AbstractButton* 697-698
isShiftDown 642-643
isUpperCase 536-537
isUpperCase 537-538
ITALIC 573-574
ItemEvent 620-625
ItemHandler 697-698
ItemListener 620-627, 664-665, 696-697
ItemStateChanged 621-623, 626-627, 697-698
iteração 292-293
iteração de um laço 216, 219-220, 233-234
iterador 1082
iterativo 288-289
Iterator 457-458, 1101-1103
Iterator, padrão de projeto 494-495, 497-498, 1106-1107
- J**
- J2RE 156-157
J2SDK (Java 2 Software Development Kit) 50-51
Jacobson, Ivar 69-70
Jacopini, G. 174-175, 244, 246
janela 683-684
janela de comando 97-98, 100, 107-108, 136-139, 145-146, 776-777, 779-780, 784-785
janela de saída de linha de comando 444-445
janela do aplicativo 680
janela do cliente 911-912
janela primária do aplicativo 689-690
janela, eventos de 684-685
janela, métodos de tratamento de eventos de 635
janela, *ouvintes* de eventos de 685-686
janela-filha 668-669, 703-704, 707, 736-737
janelas, sistema de 605-606
JApplet 142, 141-145, 151-152, 270-271, 278-279, 283-284, 297-298, 439-440, 566-567, 672-673, 690-691, 969-970
- JAR (Java archive), arquivo 73-74
jar, utilitário para Java archive 398-399
JARS, *sítio* 160
Java 2.50, 120-121, 341-342
Java 2 Multimedia Cyber Classroom Fourth Edition 50-51
Java 2 Platform, Enterprise Edition 53-54
Java 2 Platform, Micro Edition 53-54
Java 2 Platform, Standard Edition 50-51
Java 2 Runtime Environment (J2RE) 156-157
Java 2 Software Development Kit (J2SDK) 50-51, 136-137, 159-160, 260-261
Java 2 Software Development Kit (J2SDK), diretório **demo** do 140-141
Java 2 Software Development Kit (J2SDK), diretório **install** 136-137
Java 2 Software Development Kit (J2SDK), Standard Edition 50-51
Java 2 Software Development Kit (J2SDK), versão do 136-137
Java 2 Software Development Kit 94-95
Java 2, ambiente de desenvolvimento 61-62
Java 2, plataforma 53-54, 156-157, 796
Java Abstract Windowing Toolkit Event, pacote 271
Java Abstract Windowing Toolkit Package (AWT) 270-271
Java API, documentação da 65, 100-102, 1229-1231, 1234
Java Applet Rating Service 160
Java Applet, pacote 270-271
Java archive (JAR), arquivo 73-74
Java archive, utilitário para, **jar** 398-399
Java Developer Connection 160
Java Input/Output, pacote 271
Java Language, pacote 271
Java Look-and-Feel Graphics Repository 982-983
Java Media Framework (JMF 1.1), versão 1.1 do 980-981, 1181
Java Media Framework (JMF) 1113
Java Media Framework, pacote doméstico 1181
Java Networking, pacote 271
Java Plug-in 1.3 HTML Converter 156-159
Java Plug-in HTML Converter 156-157
Java Security API 931
Java Sound 1123-1124
Java Sound API 1113-1114, 1148
Java Sound Engine 1148
Java Swing Event, pacote 271
Java Swing, pacote de componentes GUI 271
Java Text, pacote 271
Java Utilities, pacote 271
Java, ambiente 62-63
Java, Apêndice de demonstrações de 1206
Java, *applet* 143-144
Java, arquivo-fonte 1237-1238
Java, biblioteca de classes 100-102
- Java, bibliotecas de classes 59-60
Java, conjunto de caracteres 138-139
Java, ferramenta para desenvolvimento 136-137
Java, interface de programação de aplicativos (API) 59-60, 100-102, 259, 270-271
Java, interpretador 62-63, 147-148, 186-187, 222-223
java, interpretador 94, 97-99, 685-686
Java, *kit* para desenvolvimento 50-51
Java, navegador da Web habilitado para 155-156
java, pacote do núcleo de 99
Java, *plug-in* 136, 146-148, 156-157, 896-897
Java, programa *applet* em 136-137
java.applet, pacote 141-143, 270-271, 1190-1191
java.awt, exceções do pacote 755-756
java.awt, pacote 142, 141-143, 148-152, 263-264, 270-271, 278-279, 341-342, 471-472, 564-565, 565-566, 584-587, 605-606, 672-673, 677-678, 700, 752-753, 967-968
java.awt.color, pacote 586-587
java.awt.event, pacote 271, 278-279, 471-472, 487-488, 610-611, 635, 642-643
java.awt.font, pacote 586-587
java.awt.geom, pacote 586-587
java.awt.image, pacote 586-587
java.awt.image.renderable, pacote 586-587
java.awt.peer, pacote 605-606
java.awt.print, pacote 586-587
.java, extensão de nome de arquivo 96-97
.java, extensão de nome de arquivo 96-97, 375-376, 471-472
java.io, exceções do pacote 755-756
java.io, pacote 271, 752-753, 820-821, 821-822
java.lang, pacote 104-107, 108-109, 154-155, 185-186, 261-262, 271, 375-376, 417-418, 436-438, 509, 535-536, 746-747, 752-753, 769-770, 1106-1107
java.math, pacote 288-289
java.net, exceções do pacote 756-757
java.net, pacote 271, 755-756, 894-895
java.sun.com 50-53, 159-160, 271
java.sun.com/applets 159-160
java.sun.com/docs/books/tutorial/java/java/java00/final.html 455
java.sun.com/j2se 50-51
java.sun.com/j2se/1.3/docs.htm 65, 100-102
java.sun.com/j2se/1.3/docs/api 270-271

- J**
java.sun.com/j2se/1.3/docs/api/index.html 65, 100-102
java.sun.com/j2se/1.3/docs/api/overviewsummary.html 271
java.sun.com/products/hotspot/ 66
java.sun.com/products/java-media/jmf 1113-1114, 1136-1137
java.sun.com/products/plugin 156-157
java.text, pacote 192, 222-226, 271
java.util, exceções do pacote 754-755
java.util, pacote 224-226, 271, 509, 541-542, 752-753, 894-895, 990-991, 1035, 1042
Java2D API 140-141, 564-565, 586-587
Java2D, applet 140-141
Java2D, diretório 140-141
Java2D, formas de 586-588
javac 61-62
javac, compilador 98-99, 186-187
javadoc 1227
javadoc, comentários ao estilo de 94-95
javadoc, homepage 1229-1231, 1234
javadoc, marcas de 1237
javadoc, programa utilitário 94-95, 1229-1231, 1234
javasound 1123-1124
javax, pacotes de extensão 102-103
javax.media, pacote 1113-1114, 1121
javax.media.control, pacote 1134-1135, 1142
javax.media.format, pacote 1133-1134
javax.media.protocol, pacote 1123-1124, 1132-1133
javax.media.rtp.event, pacote 1142-1143
javax.sound.midi, pacote 1148, 1154-1155, 1159-1160, 1190-1191, 1193-1194
javax.sound.midi.spi, pacote 1148
javax.sound.sampled, pacote 1148
javax.sound.sampled.spi, pacote 1148
javax.swing, pacote 100-103, 141-143, 151-152, 224-226, 270-271, 278-279, 482-483, 566-567, 570, 600-601, 606-607, 609-610, 615-616, 670-671, 684-685, 703-704, 707, 967-968
javax.swing.event, pacote 271, 610, 615-616, 629-630, 683-684
JBuilder da Borland 61-62
JButton 278-281, 601-602, 604-605, 617-619, 629-632, 643-644, 648-649, 651-654, 656-657
JButton, eventos 394-395
JCheckBox 604-605, 617, 619-622, 664-666
JCheckBoxMenuItem 690-691, 696-697
JColorChooser 570, 572, 665-666
JComboBox 604-605, 624-627, 664-666, 700-701, 715, 719-720
JComponent 566-567, 606-610, 615-617, 624-627, 650-651, 672-673, 677-678, 728-729
JDesktopPane 703-705, 738, 860-861
JDialog 695-696
JEditorPane 891-892, 896-897
jfc, diretório 137-138, 140-141
JFileChooser 828-829, 836-837, 839-842, 848-849, 852-853, 857-858
JFileChooser.CANCEL_OPTION 830-831, 836-837, 842-843, 849-850, 853-854
JFileChooser.FILES_ONLY 830-831, 836-837, 842-843, 848-849, 852-853, 857-858
JFrame 479-480, 482-483, 505-507, 601-602, 605-607, 648-649, 672-674, 683-687, 722-723, 725-726
JFrame.EXIT_ON_CLOSE 570
JInternalFrame 703-704, 707, 860-861
JLabel 277-281, 604-605, 607-610, 624-625, 633-635, 660-661, 664-665, 694, 696-697
JList 604-605, 626-630, 719-720, 894-895
JMenu 690-692, 696-697, 703-705
JMenuBar 690-691, 695-696, 703-705
JMenuItem 690-691, 703-705
JMF (Java Media FrameWork) 1113
JMF 980-981, 1113-1114
JMF, API 1113
jobfind.com 1316-1318
Jobs.com 1318-1319
JobsOnline.com 1322
jogar 272
jogar, programa para 272
jogo de adivinhar o número 664-665
jogo de craps 281-282
jogo de dados 275-276
jogo-da-velha 426-427, 918
jogos de cartas 544-545
jogos, cassino de 272
Johnson, Ralph 70-71
join 771-772
JOIN_ROUND 588-589
joinGroup, método de
MulticastSocket 947
Joint Photographic Experts Group (JPEG) 609-610, 969-970
JOptionPane 100-106, 151-152, 183-184, 733-734
JOptionPane para diálogo de
mensagem, constantes de 110
JOptionPane.ERROR_MESSAGE 110
JOptionPane.INFORMATION_MESSAGE 110, 116, 186-187, 196-197
JOptionPane.PLAIN_MESSAGE 105, 109, 110, 132-133
JOptionPane.QUESTION_MESSAGE 110
JOptionPane.WARNING_MESSAGE 110
JPanel 604-605, 642-643, 650-651, 653-654, 671-678, 681, 686-687, 728-729, 970-971, 1181-1183, 1305
JPanel opacos por *default*, objetos de 678-679
JPasswordField 611-613, 615-616
.jpeg 609-610, 969-970
JPEG 609-610, 982-983, 984-985
.jpg 609-610, 969-970
JPopupMenu 697-700
JRadioButton 617, 619-624, 663-665, 700-701
JRadioButtonMenuItem 690-693, 696-700
JScrollPane 627-629, 631-632, 670-672
JScrollPane, classe 239-241, 242
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS 671-672
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED 671-672
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER 671-672
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS 671-672
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED 671-672
JScrollPane.VERTICAL_SCROLLBAR_NEVER 671-672
JSlider 680-682
JTextArea 640-642, 646, 668-671, 715, 718-719, 953
JTextArea, classe 224-226, 253-254, 264-268, 320-322, 324-325, 342-343
JTextComponent 611-612, 614-615, 668-669, 671-672
JTextField 600-601, 604-605, 611-615, 618-622, 664-665, 668-669, 719
JTextField, classe 278-281, 288-289, 323-324, 335-336
JToggleButton 619-620
JumpingBox 138-139
junções de linha 589-590
juros compostos 222-224, 253-255
juros, taxa de 222-224
JustCJobs.com 1323-1324
JustComputerJobs.com 1323-1324
just-in-time (JIT), compilador 66
JustJavaJobs.com 1314, 1323-1324

K
Kelvin, escala de temperaturas 664
Kernighan e Ritchie 57-58
KeyAdapter 635

KeyDemo 640-641
KeyDemo.java 640-641
KeyEvent 616, 640-641, 660-661
KeyListener 616, 635, 640-642, 664-665
KeyListener, métodos 640-641
keyPressed 640-642
keyReleased 640-641
keys 1045-1046
keyTyped 640-641

L

LabelTest.java 607-608
 labirinto, percorrer um 601-602
 labirinto, percurso 364-365
 laço 187-188, 985-986, 1151-1152, 1197
 laço, condição de continuação do 217-220, 222-223, 228-234
 laço, condição de término do 329
 laço, contador do 216
 laço, corpo do 219-220, 229-231
Lady Ada Lovelace 61
 lançamento de moeda 272, 312-313
 lançar dois dados 356-357
 lançar um dado de seis faces 274-275, 325-326
 lançar um dado de seis faces 6000 vezes 273-274
LANs 55-56
 largura 578-579
 largura de banda 967
 largura de um retângulo em *pixels* 154-155
 largura e altura de um *applet* 685-686
 largura e altura de um *applet* em *pixels* 146-147, 263-264
 largura e altura de uma janela de aplicativo 685-686
last 1101-1102
lastElement 1041-1042
lastIndexOf 519-520
lastModified 876-877
Latin World 1320-1321
Layers, padrão de arquitetura 494-495, 959-960
LayoutManager 646
LayoutManager, interface 643-644, 646
LayoutManager2, interface 710-711
Lea, Doug 809-810
leaveGroup, método da classe *MulticastSocket* 947-948
 legado em C, código 51-52
 legibilidade 94, 95-96
 nível 876
 leiaute de componentes GUI 137-138
 leiaute *default* do painel de conteúdo 671-672
 leiaute, gerenciador de 280-281, 642-643, 650-651
 leis de De Morgan 255-256
 leitor 815

leitor de tela *braille* 607-608
 leitoras, *threads* 815
 leitores e escritores 815
 leitores e escritores em controle de simultaneidade, o problema dos 815
 lendo dados do teclado 120-121, 157-159
 lendo um arquivo de acesso aleatório seqüencialmente 855-856
 lendo um arquivo em um servidor Web 896-897
 lendo um arquivo seqüencial 835-836
length 523-524, 529-530, 649-650, 876-879
length, variável de um *array* 318-319
 ler dados do teclado 604-605
 letra 536-537, 818-819
 leve, componente GUI 684-685, 696-697
 leves, componentes 605-606
 lexicográfica, comparação 515-516, 1094-1095
 liberar o botão do mouse 665-666
 liberar recursos 757-758
 liberar um bloqueio 784-785
LIFO 418-419
Light 161-166, 204-207, 247-248, 298-301, 347-348, 350-351, 420-421, 491, 493, 551-552, 594-595, 807-809, 1257-1258, 1264-1265, 1270-1273, 422-423, 1306-1307
LightEvent 549-550, 1244-1245, 1249, 1311-1312
LightListener 594-596, 1245-1247, 1249, 1257-1258, 1270-1271
Limericks 558
 limite de crédito em uma conta de débitos 209-210
 limites de *array* 329
Line, método *close* de 1151-1152
Line, método *start* de 1151-1152
Line.Info 1151-1152
Line2D 564
Line2D.Double 586-589, 599-600
 linear, coleção 990-991
 linear, estrutura de dados 1007-1008
 linear, pesquisa 335, 336-338, 363-364
LinearSearch.java 335-336
LineEvent 1150-1152
LineEvent.Type 1151-1152
LineEvent.Type.STOP 1151-1152
LineListener 1150-1152
LineNumberReader 823-824
LinesRectsOvals.java 579-580
lineTo 590-593
LineUnavailableException 1151-1152
 linguagem de alto nível 56-57, 61
 linguagem híbrida 58-59
 linguagem sensível a maiúsculas e minúsculas 181-182
 linguagem simbólica 56-57
 linguagem, pacote de 271
 linha 138-139, 141-144, 564, 578-579, 585-587
 linha de base da fonte 576-577
 linha de comando 1238-1240
 linha de comando, argumento 685-686, 687
 linha em branco 95-96, 188-189
 linhas conectadas 584-585
 linhas de comprimento aleatório em cores aleatórias 599-600
 linhas tracejadas 587-588
link 989-991, 1007-1008, 1230-1231
link salientado 1239
-link, argumento de linha de comando 1237-1240
LinkedList, classe 990-991, 1098-1099, 1110-1111
links para pacotes 1230-1231
Linux 56-57
List 1085-1091, 1094-1097, 1100-1101, 1189-1190
list 876-877, 1056-1057, 1098-1099
list, método de **File** 880-881
List.java 992-993
 lista 624-625
 lista de argumentos separados por vírgulas 261-262
 lista de parâmetros separados por vírgulas 266
 lista escamoteável 140-141, 604-605, 624-625
 lista separada por vírgulas 106-107, 117-220, 266
 listas indexadas 1032-1033
listener, interface 83-84
listener, objeto 1271, 1282
listenerList 615-616
ListIterator 1089-1090
ListSelectionEvent 627-629, 894-895
ListSelectionListener 627-628, 664-665, 893-895
ListSelectionModel.SINGLE_INTERVAL_SELECTION 630-631
ListSelectionModel.SINGLE_SELECTION 627-628, 630-631
ListTest.java 995-996, 1089-1090
 literal 109
 lixo, coletor de 379-380, 413-417, 442, 444-446, 511-512, 751, 757-758, 769-770, 795-796, 979-980
 lixo, *thread* coletora de 769-770
load 1055-1056
LoadAudioAndPlay.java 980-981
LoadImageAndScale.java 967-968
 locais criadas no *try*, variáveis 751
 locais, redes (LANs) 55-56
 locais, variáveis 152-153, 185-186, 261-262, 283-285, 332-333, 380-381, 409
locale 224-226, 271
Locale, classe 224-225

- L**
- `Locale.US` 224-225
 - `localhost` 906
 - localização 607-608, 1337
 - localização de uma imagem na Internet 969-970
 - `Location` 489-494, 548-552, 654-655, 804-805, 807-808, 1242, 1258, 1259, 1263-1264, 1271, 1282, 1289
 - `log`, método de `Math` 261-262
 - logaritmo 262-263
 - lógica, decisão 54
 - lógica, erro de 107-108, 180-181, 187-189, 218-220
 - lógica, negação, ! 236, 239-240
 - `LogicalOperators.java` 239-241
 - lógicas, operações de entrada 823
 - lógicas, operações de saída 823
 - lógicas, unidades 54
 - lógico, E, && 236, 238-239
 - lógico, OU, || 238-239
 - lógicos, operadores 236, 239-240
 - Logo, linguagem 358
 - `LogoAnimator.java` 970-971
 - `LogoAnimator2.java` 974-975
 - `LogoApplet.java` 975-976
 - `Long` 488-489
 - `long`, palavra-chave 175-176, 202-203
 - `long`, promoções 269-270
 - `LookAndFeelDemo.java` 700-701
 - `LookAndFeelInfo` 700-701, 703-704
 - Lord Byron 61
 - losango 175-176, 182-183
 - lotes 55
 - `ls`, comando no UNIX 137-138
 - `lvalue ("left value")` 201, 317-318
- M**
- `m por n, array` 342
 - Macintosh 97-98, 565-566, 684-685, 700
 - MacOS 56-57
 - Macromedia Flash 2, filmes 1113
 - mãe especificada como `null`, janela 695-696
 - mãe para uma caixa de diálogo, janela 695-696
 - mãe, janela 668-669, 703-704
 - `main`, método 98-99, 105-106, 117, 184-187
 - maior e menor inteiros em um grupo 169-170
 - maior que, >, sinal de 146-147
 - maior valor 210-211
 - mais (+), indicando visibilidade pública, sinal de 420
 - maiúscula, letra 95-96, 106-107
 - `MalformedURLException` 893-896
 - `Manager` 1121-1122
 - `Manager`, método
 - `createRealizedProcessor`, de 1133-1134
 - manipulação de *bits* 1035
 - manipulação de cores 565-566
 - manipulando polimorficamente objetos de classes empacotadoras de tipos, 488-489
 - manutenção 454-455
 - `Map` 1082-1083, 1102-1105
 - `Map`, interface 1102-1103
 - `MapTest.java` 1102-1103
 - Máquina Analítica 61
 - máquina caça-níqueis 272
 - máquina de caixa automática 652
 - máquina, dependente de 56-57
 - máquina, linguagem de 56-57
 - marca 146-147
 - marcado para coleta de lixo 413-414
 - marcador 242, 680-682
 - marcador, posição do 683-684
 - marcar um objeto para coleta de lixo 415-416
 - máscara 1059-1060
 - `Math` 261-262, 437-438
 - `Math.E` 261-262
 - `Math.PI` 132, 169-170, 261-262, 306-307, 437-438
 - `max`, algoritmo 1093, 1096-1097
 - `max`, método de `Math` 262-263, 269-270
 - maximizada, *frame* interna 706
 - maximizar uma janela 482-483, 706
 - máximo 210-211
 - máximo divisor comum (MDC) 312-313, 314
 - `maximum`, método 267-268
 - `Maximum.java` 267-268
 - `MBAFreeAgent.com` 1324-1326
 - MBCS (*multi-byte caracter set*) 1339-1340
 - MDI 668-669, 703-704
 - média (média aritmética) 112-113
 - média 182-183, 186-188
 - média aritmética 112-113
 - média da turma 182-183
 - `MediaLocator` 1123-1124
 - `MediaLocator`, classe 1121-1122
 - `Mediator`, padrão de projeto 498-499
 - meia palavra 371
 - meio do *array*, elemento do 337-338
 - meio do mouse, botão do 640-641
 - meio, camada do 959-960
 - melhorar o desempenho 355-356
 - melhorar o desempenho do *bubble sort* 355-356
 - memento, objeto 498-499
 - Memento*, padrão de projeto 494, 498-499
 - memória 54, 55
 - memória, *buffer* na 823
 - memória, consumo de 1082
 - memória, esgotamento da 740-741
 - memória, perda de 413-414, 757-758, 769-770
 - memória, unidade de 55
 - memória, utilização da 1047-1048
 - `MemoryImageSource` 985-986
 - menor dentre vários inteiros 253-254
 - menor inteiro em um grupo 169-170
 - menores, marcas divisórias 680
 - menos indicando visibilidade privada, (-), sinal de 420
 - menos infinito 746-747
 - menos ou mais infinito 746-747
 - mensagem 67-68, 97-98, 138-139, 144-145, 153-154, 246-247, 347-352, 548, 550-552, 802-806, 1256-1257
 - mensagem, diálogo de 100-102, 105, 109, 184-185, 224-226, 242, 253-254, 278-279, 320-321
 - mensagem, estrutura do nome de 548
 - mensagem, passagem de 805-806
 - mensagem, tipos de diálogos de 109
 - menu 102-104, 604-605, 668-669, 690-691, 695
 - menu `Applet` de `appletviewer` 139
 - menu `popup` sensível ao contexto 697-698
 - menu, item de 690-691, 695-696
 - menus, barra de 103-104, 604-605, 690-691, 695-696
 - `MenuTest.java` 691-692, 694
 - mesma assinatura 439-440
 - mesma prioridade, *threads* de 813-814
 - mesmo objeto 526-527
 - `MessageListener.java` 934-935
 - `MessageManager.java` 940-941
 - `Meta`, chave 639-641
 - metal, aparência e comportamento de 668-669, 700-701
 - `MetaMessage` 1193-1194
 - `MetaMessage`, classe 1159-1160
 - `MethodOverload.java` 295-297
 - método 59-60, 68-69, 97-98, 246-247, 260, 420, 490-491
 - método `actionPerformed`
 - `ActionListener` 279-281, 289-290, 394-395, 471-472, 480-481, 484-485, 600-601, 611-612, 615-618, 630-631, 644-645, 648-649, 670-671, 687-688, 710-711, 746, 751
 - método chamado 260-261
 - método concreto 490-491, 1260-1261
 - método é declarado `final` 456-457
 - método `paint` de um *applet* 690-691
 - método que chamou 330
 - método sobrecarregado, chamada de 405-406
 - método, chamada de 144-145, 260-261, 266
 - método, corpo de 266
 - método, definição 97-98, 266
 - método, encadeamento de chamadas 411-412
 - método, nome de 299-300
 - método, operador de chamada de 263-264
 - método, sobrecarga de 294-295, 302-303, 376-377

- métodos chamados automaticamente durante a execução de um *applet* 298
- métodos da classe **Character** para conversão de/para maiúsculas/minúsculas 536
- métodos implicitamente **final** 455, 488-489
- métodos, monitoramento da pilha de chamadas de 771-772
- métodos, pilha de chamadas de 761-762
- métrica, programa de conversão de 560-561
- Metrics.java** 577-578
- Microsoft 60-61, 1337-1338
- Microsoft Audio/Video Interleave, arquivos 1113
- Microsoft Internet Explorer 100-102, 136, 931
- Microsoft Windows 565-566, 684-685, 700
- Microsoft Windows, aparência e comportamento no estilo do 700-701
- .mid 1113
- .mid, arquivo 1190-1191
- .mid (extensão de arquivo MIDI) 980-981, 1113, 1154-1155
- MIDI 980-981, 1148, 1154-1155
- MIDI 1190-1192, 1308-1309
- MIDI, canal 1165-1166
- MIDI, dispositivo 1158-1160
- MIDI, evento 1155, 1159-1160
- MIDI, geração de nota em 1165-1166
- MIDI, gravação em 1159-1160, 1162-1163
- MIDI, instrumentos 1162-1163
- MIDI, mensagem 1158-1159, 1165-1166
- MIDI, notas 1159-1160
- MIDI, resolução 1179-1181
- MIDI, salvando 1162-1163
- MIDI, sequência 1159-1160
- MIDI, sincronização de eventos 1155
- MIDI, sintetizador 1154-1155
- MIDI, tipo 1162-1163, 1197
- MIDI, tipos de arquivos suportados por 1162-1163
- MIDI, trilha 1158-1159
- MIDI, volume 1179-1180
- mídia capturada 1123-1124
- MidiChannel** 1162-1163
- MidiData** 1154-1155, 1158-1159
- MidiData.java** 1155
- MidiDemo** 1155, 1166
- MidiDemo.GUI** 1166
- MidiDemo.java** 1167
- MidiDevice** 1158-1160
- MidiEvent** 1159-1160
- MidiMessage** 1159-1160, 1166
- MidiRecord** 1154-1155, 1159-1163, 1179-1180
- MidiRecord.java** 1160-1161
- MidiSynthesizer** 1154-1155, 1162-1163
- MidiSynthesizer.java** 1162-1163
- MidiSystem** 1158-1159
- MidiUnavailableException** 1158-1159, 1165-1166
- min, algoritmo 1093, 1096-1097
- min, método de **Math** 262-263
- minimizar uma *frame* interna 706
- minimizar uma janela 482-483, 685-686, 706
- mínimo em um *array*, valor 363-364
- mínimo, navegador 147-148
- minúsculas, letras 95-96, 106-107, 181-182, 818-819
- MiscBitOps.java** 1061-1062
- missão, aplicativo crítico para a 741-742
- missão, computação crítica para a 750-751
- MissingResourceException** 147-148
- mistas, expressão com tipos 269-270
- mnemônico 607-608, 690-691, 695, 696-697
- modal, diálogo 572, 695-696, 833
- modelar uma interação 349-350
- modelo (na arquitetura MVC) 957-959
- modelo 83-84, 120-121, 123-124, 160-165, 203-206, 490-491, 594-597, 721-725, 1248-1249, 1251, 1256-1257, 1260-1261, 1264, 1272-1273, 1282, 1289
- modelo de computação ação/decisão 177-178
- modelo de delegação de evento 611-612
- Model-View-Controller (MVC)** 83-84, 494-495, 695-696, 721-722, 729-730, 957-958
- modificador de acesso 489-490
- modificador de acesso a membro 420
- modificador de acesso a membro **private** 375-376
- modificador de acesso a membro **public** 375-376
- modificador de chave 642-643
- modificador, método 381-382, 391
- modificadores de acesso a membro 375-376
- modificáveis, strings 528-529
- modularizando um programa com métodos 262-263
- módulo 260
- módulo, operador, % 110-112, 127-128, 213-214, 254-255
- módulos em Java 260
- MoleculeViewer**, *applet* 138-139
- monitor 777-778, 804-806
- monitor de vídeo 564-565
- monitor, objeto 777-778
- monitor, variável de condição de um 782-783
- MonitorControl**, interface 1134-1135
- monolítico, programa 294-295
- Monospaced** 341-342
- Monospaced** 573-575, 719-720
- Monster.com** 1314, 1316-1318, 1322, 1324-1326
- montador (*assembler*) 56-57
- MorganWorks.com** 1320-1321
- Morse, código 560-561, 965, 1205
- morta 772-773
- Motif (UNIX), aparência e comportamento no estilo 668-669, 700-701
- mouse 55, 138-139, 604-605
- mouse com um, dois ou três botões 639-640
- mouse da janela do aplicativo, eventos do 700
- mouse pressionado, evento 680
- mouse, apontador do 104-105, 107-108, 154
- mouse, botão do 139, 632-633
- mouse, clique do 637-638
- mouse, clique do botão do 639-640
- mouse, cursor do 632-633, 899-900
- mouse, evento do 506
- mouse, eventos de movimentação do 680
- mouse, eventos do 616, 631-632, 671-672, 675-676
- mouse, operação de arrastar o 680
- mouse, tratamento de eventos do 632-633
- MouseAdapter** 635, 675-678, 698-700, 977-978
- mouseClicked** 632-633, 637-639
- mouseClickHandler** 635
- MouseDetails.java** 637-638
- mouseDragged** 632-637, 680
- mouseEntered** 632-633, 634
- MouseEvent** 616, 632-633, 635-636, 675-676, 679-680, 700, 977-978
- mouseExited** 634, 977-978
- MouseListener** 616, 631-632, 698-699
- MouseMotionAdapter** 635-636, 977-978
- MouseMotionListener** 616, 631-632, 635, 679-680
- mouseMoved** 632-636, 680, 977
- mousePressed** 632-633, 662-663, 675-678, 698-700, 815-816
- mouseReleased** 632-633, 661-662, 676-678, 698-700, 928-929
- MouseTracker** 633-634
- MouseTracker.java** 633-634
- .mov 1113, 1194-1195
- .mov (extensão de arquivo QuickTime) 1113
- movendo o mouse 140, 610
- moveTo** 590-591
- MovingPanel** 1181-1183, 1185-1187, 1190-1191, 1309-1310, 1312
- MovingPanel.java** 1185-1186
- .mp3 1113
- .mp3 (extensão MPEG Layer 3) 1113
- MPEG-1, vídeos 1113

- .mpeg (extensão de arquivo MPEG-1) 1113
- MPEG Layer 3*, áudio 1113
- .mpg 1113
- .mpg (extensão de arquivo MPEG-1) 1113
- MS-DOS, *prompt* do 62-64, 137-138
- mudando a aparência e o comportamento de uma GUI baseada em Swing 703-704
- mudar de diretório 137-138, 140-141
- multi-byte character set* (MBCS) 1339-1340
- multicast* 931, 933-934, 937-938
- multicast*, endereço de 937-938, 939-940
- multicast*, grupo de 937-938
- multicasting* 891-892
- MulticastSendingThread.java** 938-939
- MulticastSocket** 944-945, 947
- multidimensional, array* 342-343
- Multimedia Authoring System 1204-1205
- Multimedia Authoring System, exercício de projeto com o 1204-1205
- Multimedia Gallery 1181
- mídia 51-52, 967
- mídia digital 1113
- múltipla, estrutura de seleção 229-230, 246-247
- múltipla, herança 67-68, 431-432, 502-503
- múltipla, lista de seleção 626-632
- múltipla, seleção 175-176
- múltiplas linhas, comentário com 94-95
- múltiplas *threads*, aplicativo de servidor com 965
- múltiplas *threads*, aplicativos colaboradores em rede, com 814
- múltiplas *threads*, linguagem de programação para 802
- múltiplas *threads*, programa do jogo-davelha com 964-965
- múltiplas *threads*, seguro quanto a 995
- múltiplas *threads*, servidores com 964-965
- MULTIPLE_INTERVAL_SELECTION** 628-632
- MultipleSelection.java** 629-630
- multiplicação, * 110-112
- multiplicativos, operadores: *, / e % 193-194
- multiplicidade 162-164
- múltiplos botões, mouse com 639-640
- múltiplos documentos, interface com (MDI) 668-669, 703-704
- múltiplos níveis, prioridade com 773-774
- multiprogramação 55-56
- multitarefas 61
- multithreading* 51-54, 61, 83-84, 154-155, 769-770, 802-804, 809-810
- multithreading* prédefinido 769-770
- mundial, padronização 419
- Musical Instrument Digital Interface (MIDI), formato de arquivo 980-981
- Musical Instrument Digital Interface 1154-1155
- Musical Instrument Digital Interface, arquivos 1113
- Musical Instrumental Data Interface (MIDI) 1148, 1196-1197
- mutuamente exclusivas, opções 621-622
- MVC 83-84, 120-121, 721-723, 957-959
- N**
- não-ambígua (base de projeto do Unicode) 1337-1338, 1343-1344
- não-capturada, exceção 740-741
- não-editável 281, 668-669
- não-editável, **JTextField** 281
- não-editável, texto ou ícones 604-605
- não-estruturado, fluxograma 244, 246
- não-fatal, erro de lógica 180-181
- não-fatal, erro durante a execução 64-65
- não-lineares, estruturas de dados 990-991
- não-modificáveis, empacotadores 1104-1105
- não-public, métodos 376-377
- não-recursiva, chamada de método 293-294
- não-static, classe interna 488-489
- não-static, membros de classe 417-418
- não-verificada 140-141
- não-verificados, **RuntimeException** e **Error** 752
- NASA multimedia gallery* 982, 1181
- native**, palavra-chave 175-176
- nativo, código 52-53
- nativo, código de máquina 66
- natural de um computador particular, linguagem 56-57
- natural, logarítmico 262-263
- navegação, barra de 1239
- navegador 62-64, 100-102, 145-146, 216, 298, 685-686
- navegador no qual um *applet* é executado 144
- navegando 892-893, 973-974
- navegando pela Internet 147-148
- navegar por uma estrutura de diretórios 386
- negativo, grau 581-582
- negativos, ângulos de arcos 582-583
- negativos, números binários 1216
- NervousText**, *applet* 138-139
- NetBeans 61-62
- Netscape Navigator 100-103, 136, 157-159, 604, 931
- new**, operador 175-176, 192, 319-321, 377-378, 386-387, 413-414, 990-991, 1082
- newInstance**, método de **RTPManager** 1142
- next** 1088-1091, 1101-1102
- nextDouble** 1058-1059
- nextElement** 1044-1045, 1050-1051, 1055-1056
- nextFloat** 1058-1059
- nextGaussian** 1058-1059
- nextInt** 1057-1058
- nextToken** 543-544
- níveis de aninhamento 217-218
- níveis de refinamento 189-190
- nível, percorrer uma árvore binária em ordem de 1013-1014, 1021
- NoDataSourceException**, classe 1134-1135
- NoPlayerException** 1121-1122
- NoProcessorException** 1133-1134
- NORTH 646-647, 715
- NORTHEAST 715
- NORTHWEST 715
- NoSuchElementException** 1041-1042
- notação algébrica 111-112
- noteOff**, método 1165-1166
- noteOn**, método 1165-1166
- Notepad 61-62
- notify** 772-774, 784-785, 800-801, 809-810
- notifyAll** 772-774, 778-779, 801
- nova linha, seqüência de escape, \n 100-101, 106-107, 149-150, 224-226, 371, 509-510
- null 103-104, 175-176, 282-283, 319-320, 444-445, 769-770, 989-990, 1006-1007, 1103-1104
- NullPointerException** 609-610, 752, 1052-1053
- Num Lock**, tecla 640-641
- NumberFormat**, classe 224-225
- NumberFormatException** 746, 748-751
- numeração, sistemas de 538-539
- numéricas, classes 488-489
- número complexo 426
- números de 905-906
- O**
- Object** 375-376, 436-437, 488-489, 1242
- Object Management Group* (OMG) 69-70

- Object**, classe 375-376, 502-503, 606-607, 839-840
ObjectInput 823-824, 827-828
ObjectInputStream 820-821, 823-824, 834-835, 839-840, 842-843, 875-876, 901-902, 906-907
ObjectOutput 823-824
ObjectOutputStream 820-821, 823-824, 827-829, 875-876, 901-909, 957, 1056-1057
 objeto (ou instância) 51-52, 58-59, 143-144, 373-374
 objeto 66-69, 82-83, 144, 347-350
 objeto a ser disparado 743-744
 objeto anônimo **String** 509-510, 515-516
 objeto cliente 548, 957-958
 objeto de classe para objeto de super-classe, conversão de 445-446
 objeto de fluxo encadeado 833-834
 objeto de subclasse é um objeto de super-classe, relacionamento 431-432, 445-446
 objeto de uma subclasse 436-437
 objeto de uma subclasse é instanciado 442
 objeto de uma superclasse 436-437
 objeto zelador 498-499
 objetos (OBP), programação baseada em 373-374
 objetos (OOAD), processo de análise e projeto orientados a 68-69
 objetos (OOD), projeto orientado a 66-68, 81-82, 84-85, 120-121, 123, 298-299, 420, 1193-1194
 objetos (OOP), programação orientada a 136-137, 431
 objetos (OOP), programação orientada a 50-51, 54, 58-59, 120-121, 373-374
 objetos 373
 objetos construídos “de dentro para fora” 444-445
 objetos devolvidos por referência 330
 objetos, diagrama de 82-83, 123-124, 164-165, 1308-1309
 objetos, fase de análise orientada a 652
 objetos, linguagem orientada a 67-68
 objetos, orientação a 66-67, 373
 objetos, orientada a 373-374
 objetos, serialização de 906, 1052-1053
 objetos, sistema operacional orientado a 457-458
 objetos, técnica para tratamento de exceções relacionadas orientada a 766-767
Observable, classe 730-731
 observador, objeto 729-730
Observer, interface 730-731
Observer, padrão de projeto 494-495, 498-499, 729-730
 octal (base 8) 254-255
 octal (base 8), sistema de numeração 1217
 oculta 684-685
 ocultar a representação interna dos dados 420
 ocultar os detalhes da implementação 260-261, 380-381, 418-419
 ocultar um diálogo 104-105, 110
 ocultar uma variável de instância 409
 “ocultas”, variáveis de instância 283-284
 Oito Rainhas 361-364
 Oito Rainhas: abordagens com força bruta 361-362
OK, botão 104-105, 154
 OMG 69-70
on-line, cursos de treinamento 160
on-line, documentação 606-607, 752-753
on-line, recrutamento 1315-1316
on-line, serviços de empreitada 1323-1324
 OOAD 68-70
 OOD 66-68, 160, 202-203, 205-206, 1311-1312
 OOP 69-70, 120-121, 373-374, 431
 opaco 672-673
 opacos, componentes de GUI Swing 678-679
 opção 140, 140-141
open, método de **Clip** 1151-1152
open, método de **DataSink** 1134-1135
 operação 82-83, 161-162, 298-302, 347-348, 368-369, 420-422, 488-493, 593-594, 807-809
 operação de atacado 1087
 operação, código de 366
 operacionais, sistemas 55
 operacional 769-770
 operacional, primitiva para *multithreading* do sistema 769-770
 operações de um tipo abstrato de dado 419
 operador == 526
 operador 110-111, 189-190
 operador binário 107-108, 110-111, 239-240
 operador de atribuição, = 107-109, 114-116, 154-155, 192-193, 199-200
 operador de coerção 189-190, 192-193, 270-271, 436-437
 operador de complemento sobre *bits*, ~ 1066-1067
 operador OU exclusivo sobre *bits*, ^ 1066-1067
 operadores aritméticos 110-111, 366-367
 operadores aritméticos binários 192-193
 operadores de atribuição 198-199
 operadores de atribuição aritmética: +=, -=, *=, /= e %= 207
 operadores de atribuição sobre *bits* 1070-1071
 operadores de deslocamento sobre *bits* 1066-1067
 operadores sobre *bits* 1058-1059
 operadores, Apêndice com a tabela de precedência de 1213
 operadores, precedência de 111-112, 292
 operadores, tabela de precedência de 193-194
 operando 107-108, 192-193, 366
 óptico, disco 818
or 1072-1073
 Oracle Corporation 1337-1338
 ordem 173, 173-174
 ordem ascendente 335, 1084-1085, 1100-1101
 ordem crescente 332-333
 ordem de promoção 192-193
 ordem de tratadores de exceção 766-767
 ordem decrescente 333-334
 ordem descendente 335, 1084-1085
 ordem na qual as instruções são executadas 157-159
 ordem na qual construtores e finalizadores são chamados 442
 ordem, percorrer em 1010
 ordenado, *array* 991-992
 ordenando 332-333, 860-861, 989
 ordenando *arrays* grandes 335
 ordenar alfabeticamente 513-514
 ordenar um *array* 335
 orientado a ações 373-374, 67-68
 origem de um evento 394-395
 origem, componente de 700
 originador, objeto 498-499
OtherCharMethods.java 540-541
 otimizado, código 1030-1031
 otimizador, compilador 226
 otimizando o compilador Simple 1030-1031
 OU exclusivo lógico booleano, ^ 239-240
 OU exclusivo sobre *bits*, ^ 1058-1059, 1066
 OU inclusivo lógico booleano, | 238-239
 OU inclusivo sobre *bits*, | 1058-1059
OutOfMemoryError 990-991
OutputStream 822-824, 900-902
OutputStreamWriter 824-825
ouuinte 594-596, 740-741, 1242, 1245-1246, 1248
OvalPanel.java 681-682
 oxímoro 323-324

P

- pack**, método 707
package, instrução 382-383, 1237-1238
PackageDataTest.java 406-407
package-list.txt 1238-1240
PacketReceiving-Thread.java 944-945
 pacote 100-102, 124-125, 270-271, 378, 382-383, 402-403, 722-724, 1237-1238, 1248, 1256-1257, 1311-1312
 pacote 891-892, 912
 pacote aos membros de uma classe, acesso de 406-407
 pacote de datagrama 912

- pacote de uma superclasse, membros com acesso de 435-436
 pacote *default* 378
 pacote do núcleo 100-102
 pacote é recebido 918
 pacote, acesso de 405-406, 432-433, 435-436
 pacote, métodos com acesso de 405-406
 pacote, nome de 151-152
 pacotes da API Java 270-271
 pacotes, dando nomes a 384-385
 pacotes, estrutura de diretórios de 382-383
 pacotes, nomes de diretórios de 384-385
 padrão 587-588
 padrão de Is e Os 818-819
 padrão de projeto 70-71, 84-85, 494-499, 721-722, 726-731, 809-811, 957, 1106-1107
 padrão de projeto de comando 729
 padrão de tabuleiro de damas 132-133, 170-171
 padrão, entrada 820-821
 padrão, formato de hora 378-379
 padrão, objeto fluxo de saída 820-821
 padrão, objeto fluxo para saída de erros 820-821
 padrão, objeto saída 97-98
 padrão, processo de desenvolvimento 420-421
 padrões comportamentais de projeto 70-71, 494, 497-498, 729, 1106-1107
 padrões de arquitetura 494-495, 957-960
 padrões de criação de projeto 70-71, 494, 726-727, 956-957, 1105-1106
 padrões *de facto* 419
 padrões de projeto para concorrência 809-810
 padrões de projeto simultâneos 494-495
 padrões de projeto xxxv
 "padronizadas, partes intercambiáveis" 68-69
 padronizado, tratamento de exceções 740-741
 padronizados, componentes reutilizáveis 433-434
 pagamento, arquivo de folha de 818-820
 pagamento, sistema de folha de 445-446, 457-458
Page Down, tecla 640-641
Page Up, tecla 640-641
 página, *software* de leiaute de 509
 pai, diretório 876-877
 pai, grupo de *threads*- 801
 pai, nodo 1007-1008, 1020-1021
 painel 650-651
 painel de conteúdo 263-264, 266, 280-281, 288-289, 330-331, 572, 606-607, 609-610, 696-697
 painel de conteúdo de um *applet* 266, 280-281, 288-289
paint 588-589, 606-607, 672-673, 689-690
paint, método da classe *JApplet* 216-220
paint, método de *JApplet* 142, 144-145, 151-156, 212-213, 298
paintComponent 566-567, 672-673, 677-678, 681-682, 690-691, 736-737, 929-930, 973, 1182-1183
Painter.java 635-636
paintIcon 705-706, 967-968, 970-971, 982-983
 palavra-chave 95-96, 176-177
 palavras de um valor de cheque, equivalente em 560-561
 palavras e frases que descrevem 203-206
 palavras, processador de 62-64, 509, 519-520
 palavras-chave reservadas mas não usadas por Java 176-177
 palíndromo 213-214, 363-364
 Palo Alto Research Center (PARC) 58-59
panel, classe 642-643
PanelDemo.java 650-651
 parada 1135-1136
 paralelas, atividades 61
 paralelogramo 433-434
param, marca 892-893, 973-974
parameters, nota 1235-1236
 parâmetro 261-262, 264-266
 parâmetro de um *applet* 892-893
 parâmetro, nome de 266
 parâmetro, tipo de 266
 parâmetros, lista de 144-145, 266
 PARC 58-59
 parênteses "no mesmo nível" 112-113
 parênteses forçam a ordem de avaliação 111-112
 parênteses, () 111-112
parentOf 802
 Parnas, D. L. 815
parseDouble, método de *Double* 150-151, 154-155, 262-263, 267-269
parseInt, método de *Integer* 105, 108-110, 117-118, 192, 196-197, 262-263, 266-267, 746
 parte cliente de uma conexão por fluxo com soquetes 906-907
 participação 364-365
 particionamento, etapa de 364-365
 Pascal 50-51, 52-53, 60-61
 Pascal, Blaise 61
 passagem por referência 330
 passagem por valor 330, 330-331, 373-374
 passar 332-333
 passar um *array* para um método 330-331, 352
 passar um elemento de *array* para um método 330-331
PassArray.java 330-331
 passeio fechado 361-362, 601-602
Patch, classe 1166
PATH, variável de ambiente 139
 PDP-11 57-58
 PDP-7 57-58
 pedido de proposta 1324-1326
peek 1045-1046
peer 605-606
 "pensar em objetos" 66-67, 373
Peoplescape.com 1320-1321
 pequeno círculo 229-230
 pequeno círculo, símbolo 174-177
 pequeno losango, símbolo 248-249
 pequenos, métodos 266-267
 percorrer a lista 999-1000
 percorrer uma árvore 1011-1012
 perfeito, número 311-312
 persistente, **Hashtable** 1052-1053
 persistentes, dados 818
Person 161-165, 203-207, 247-249, 298-301, 347-351, 420-421, 489-493, 548, 551-552, 592-593, 593-594, 78-79, 652-657, 802-809, 1251, 1256-1261, 1274-1275, 1282, 1287-1289, 1306-1307
 personalizando a classe **Jpanel** 668
PersonMoveEvent 549-550, 1244-1245, 1249, 1287-1288, 1310
personMoveListener 1287-1288
PersonMoveListener 594-596, 1245-1249, 1258, 1287-1288
 pesquisa 329
 pesquisa binária 335-340
 pesquisa em árvore binária 1021
 pesquisa, chave de 335, 1100-1101
 pesquisa, programa de análise de 327-328
 pesquisando 335, 989
 pessoa 803-804
 pessoal, computação 55-56
PI 169-170
 piano, tocador de 1155, 1180-1181
PIE 588-589
PieceWorker.java 460-461
Pig Latin 558
 pilha 313, 376-377, 418-419, 745-746, 759-760, 762, 989, 1000-1001
 pilha de chamadas 761-762
 pilha é desempilhada 762
 pilha, desempilhamento da 759-760
 pilha, mensagem de monitoramento da 762
 pilha, monitoramento da 771-772
 "pinçar" cada dígito 133-134
 pintando em componentes de GUI Swing 668
pipe 822-823
PipedInputStream 822-823
PipedOutputStream 822-823
PipedReader 824-825
PipedWriter 824-825
 Pitágoras, triplas de 255-256
pitch 1159-1160
pixel 145-146, 216, 564-565, 677-678
pixel, coordenadas de 141-143, 148-149

- PixelGrabber** 985-986
PLAIN 573-574, 620-621
PLAIN_MESSAGE 105, 109
 planilha de cálculo, programa de 62-64
 planta baixa 143-144
 plataforma 51-52
play 979-980, 980-981
play, método da interface **AudioClip** 979-980
play, método de **Applet** 979-980
Player 1121-1122
Player, interface 1113-1114, 1121-1123
Player, método 1121-1123
Player, método **close** de 1121-1122
Player, método **setMedia-Time** de 1123-1124
Player, método **start** de 1122-1123
Player, método **stop** de 1123-1124
 plugável (PLAF), aparência e comportamento 668-669
 plugável, pacote para aparência e comportamento 607-608
plug-in 147-148
.png 609-610
.png, arquivo 1190-1191
Point.java 437, 442-443, 447-448, 466-467, 472-473
Point2D 1182-1183
poker 557-558
poker, jogo de cartas 964-965
 polida, prática de programação 801
 polígono 585-586, 664
 polígonos fechados 584-585
 polilinhas 584-585
 polimórfica, programação 454-455
 polimórfico de erros relacionados, processamento 756-757
 polimórfico, comportamento 464-465
 polimórfico, de coleções, processamento 1087
 polimórfico, gerenciador de tela 456
 polimorfismo 229-231, 373-374, 431, 441-442, 445-446, 453-454, 456-457, 465
 polimorfismo como alternativa à lógica de **switch** 504-505
 polinômio 113-115
Polygon 564, 584-586
 ponta de seta 805-806
 pontilhamento 137-138
 ponto (.), operador 103-104, 144-145, 261-262, 380-381, 411-412, 414, 435-436, 439-440, 572, 586-587
 ponto 139, 573-574
 ponto de desvio 249-250
 ponto de *handshake* 900-901, 911-912
 ponto de venda, sistema de 846
 ponto decimal 192
 ponto, operador de seleção de membro 464-465
 ponto, tamanho do 341-342
 ponto-e-vírgula, ; 98-99, 106-107, 118-119, 181-182
pool de threads 901
pop 1035
pop 418-419
popup, evento de disparo de 697-700
popup, menu 698-699
PopupTest.java 697-698
port, número de 900-901
 porta 1136-1137
 porta abaixo de 929-930,
 porta de conexão 905-906
 porta, número de 901, 905-906, 911-912, 918
 portabilidade 64-65, 565-566, 1339-1340
 portabilidade, dicas de 59-60
 portáveis, GUIs 271
 portável 51-52, 57-58
 portável, linguagem de programação 770-771
 pós-decremento 199-201
 pósfixa, algoritmo de avaliação de expressão 1017-1018
 pósfixa, notação 1017-1018
 posição (0, 0) 141-143
 posição 1182-1183, 1185-1186
 posição dos dois 1218-1219
 posição na memória do computador 110
 posição zero, número de 317-318
 posição, número de 317-318
 posicionais no sistema de numeração decimal, valores 1217-1218
 posicional, notação 1217
 posicional, valor 1217-1218
 pós-incrementar 220-221
 pós-incremento 199-201
 positivos e negativos, ângulos de arco 582-583
 positivos, graus 581-582
 pós-ordem, percurso na 1010-1011
postorderTraversal 1010-1011
 potência 262-263, 311-312
 potência de 2 maior do que 1000 181-182
 poupança, conta de 222-224
pow, método de **Math** 224-226, 262-263, 306-307
power, método 313
 precedência 119-120, 201, 292
 precedência dos operadores aritméticos 112-113
 precedência e associatividade de operações 318-319
 precedência, Apêndice com a tabela de 1213
 precedência, tabela de 111-112, 193-194
 precisão dupla, número em ponto flutuante com 149-151
 pré-decrementar 199-201
 predefinida, constante 437-438
 predicado, método 376-377, 995
 pré-empacotadas, estruturas de dados 1082
 preempção 770-771
 preempção de uma *thread* 773-774
 preencher com cor 564
 preenchida, forma 590
 preenchido, retângulo 569-570
 preenchido, retângulo tridimensional 579
 preenchimento, textura de 590
 pré-fabricados, componentes de *software* 51-52
prefetchComplete, método 1122-1123
PrefetchCompleteEvent 1122-1123
 pré-incrementando 220-221
 pré-incrementar 199-201
 prematuro, término do programa 329
 pré-ordem, percorrer na 1009-1010
preorderTraversal 1010-1011
 pressionar e segurar o botão do mouse 140
 pressionar o botão do mouse 665-666
 primária, janela 689-690
 primária, memória 55
 primeira passagem 1024-1025
 primeiro a entrar, primeiro a sair (FIFO), estrutura de dados 419-420, 1004-1005
 primeiro refinamento 187-188, 194-195
 primitivo (ou *built-in*), tipo de dado 106-107, 154, 202, 269-270, 282-283, 288-289
 primitivo, tipo 205-206
 primitivos devolvidas por valor, variáveis de tipos de dado 330
 primitivos sempre passadas por valor, variáveis de tipos de dado 330
 primitivos, variáveis de tipos de dado 153-154
 primo 311-312, 1072-1073, 1110-1111
 primos, números 361-362
Princeton Review 1326-1327
 principais, marcas divisórias 680
 principal 222-224
 princípio do mínimo privilégio 282-283, 381-382, 401, 855-856
print, método de **System.out** 100, 107-108
PrintBits.java 1059-1060
println, método de **System.out** 97-98, 100, 107-108
printStackTrace 761-762
PrintStream 822-823, 1056-1057
PrintWriter 944-945
 prioridade mais alta, *thread* de 769-771
 prioridade máxima, *thread* de 771-772
 prioridade máxima, *thread* pronta de 773-774
 prioridades, escalonamento com 774-775
private 420
private de uma superclasse, membros 435-436
private static, membro de classe 414
private, dados 381-382, 391-392
private, membros de classe 380-381
private, método 375-376
private, palavra-chave 144, 175-176

- p**
private, variável de instância 391
private, visibilidade 420
 probabilidade 272
 probabilidade 273-274
 problema de concorrência 809-810
 problema, definição do 81-83, 120-121,
 203-206, 652-653, 1181-1182
 procedimento 173
 procedurais, linguagens de programação
 373-374
 procedural, linguagem de programação 67-
 68
 processamento de dados comercial 886-
 887
 processamento em lotes 55
 processamento, fase de 189-190
 processamento, unidade de 54
 processando polimorficamente, **Shapes**
 473-474
 processo de compilação 1027-1028
 processo de projetar 68-69, 81-84, 120-
 121, 123, 299-302, 420-421, 722-724,
 806-807, 1193-1194
Processor, interface 1123-1124, 1132-
 1133
Processor, método 1134-1135, 1142
Processor>, método 1142
ProcessorModel, classe 1133-1134,
 1195-1196
 produção, trabalhador por 460-461
ProduceInteger.java 779-780,
 782-783, 788-790
 produtividade 55-56
 produto de inteiros ímpares 252-253
 produtor 782-785
 produtor, método 778-779
 produtor/consumidor, relacionamento 778-
 779
 produtora, *thread* 778-779
 produtos 1209
 programa de computador 54
 programa de consulta de crédito 840
 programa de conversão de métrica 560-561
 programa de média da turma com repetição
 controlada por contador 182-183
 programa de média da turma com repetição
 controlada por sentinelha 190-191
 programa de pintar 665-666
 programa para desenhar 62-64, 427-428
 programa para jogar damas 964-965
 programa para jogar xadrez 964-965
 programa, controle do 94, 136-137, 173-
 174
 programa, pilha de execução do 1000-1001
 programa, processo de desenvolvimento de
 um 418-419
 programa, término de um 329
 programação em linguagem de máquina
 366
 programação simultânea 769-770
 programador de computador 54
 programando genericamente 504-505
 programas, desenvolvimento de 94, 136-
 137
 programas, ferramentas para
 desenvolvimento de 198-199
 programas, manutenção de 454-455
 programas, princípios de construção de 216
programChange, método 1166
 projeto 350-352, 725-726
 Projeto: Simulador de vôo 1204-1205
 Projeto: Sistema de composição em
 multimídia 1204-1205
 promoção 192-193
 promoção, regras de 269-270
 promoções para tipos de dados primitivos
 269-270
prompt 107-108, 192
 pronta, estado 771-772, 780-781, 784-785,
 797
Properties 1035, 1052-1053, 1056-
 1057, 1079
PropertiesTest.java 1053-1054
propertyNames 1055-1056
 proteção de cheque 559-560
protected de uma subclasse, membros
 435-436
protected de uma superclasse,
 membros 431-432, 435-436
protected, acesso 431-432
protected, membro 489-490
protected, palavra-chave 144, 175-176,
 380-381
protected, variável de instância 439-
 440, 442, 448-449
Prototype, padrão de projeto 494-496,
 1106-1107
Proxy, padrão de projeto 494, 497-498
 pseudo-aleatórios, gerador de números
 1057-1058
 pseudocódigo 68-69, 173-174, 176-178,
 181-183, 188-190, 195-196
public abstract, método 471-472
public de uma subclasse, membros 435-
 436
public encapsuladas em um objeto,
 operações 379-380
public final static, dados 471-
 472
public herdados para uma classe,
 métodos 465
public static, membros de classe
 414
public static, método 414
public, classe 96-97, 144, 374-375, 383-
 384
public, dados 375-376
public, método 375-377, 380-381
public, palavra-chave 96-97, 143-145,
 175-176, 375-376, 420-422
public, serviço 376-377
public, visibilidade 420
 pública, interface 376-377, 449-450, 457-
 458
 puros, componentes Java 605-606
push 1035
push 418-419
PushBackInputStream 823-824
PushbackReader 823-824
put 893-894, 1045-1046, 1103-1104
- Q**
- quadrada, raiz 262-263
 quadrado 433-434, 580-581
quantum 770-771, 773-774
quantum, expiração do 772-773
QueueInheritance.java 1004-
 1005
QueueInheritanceTest.java
 1005-1006
quicksort 363-364
 QuickTime, arquivos 1113, 1194-1195
Quit, do menu **Applet** do
appletviewer 140-141
- R**
- "r", modo de abertura de arquivo 850-
 851, 855-856
 radianos 262-263, 592-593, 598-599
 radio, botões de 140, 617, 621-622
RadioButtonHandler 623-624
RadioButtonTest.java 622
 raio 169-170
 raio de um círculo 314-315
 raiz 538-539
 raiz quadrada de um inteiro negativo é
 indefinida, 419
 raiz, diretório 876
 raiz, nodo 1007-1008
 RAM (*random access memory*) 55
Random 1035, 1057-1058
random 1057-1058
Random, classe 271
random, método de **Math** 272, 274-275,
 309-310, 355-356, 414, 779-780, 783-
 785, 789-791, 1010-1011
RandomAccessAccountRecord.java
 847
RandomAccessFile 822-823, 839-
 840, 846-851, 853-856, 875-876
RandomCharacters.java 797
RandomIntegers.java 272-273
Rational 426
Rational 426
 Rational Software Corporation 69-70, 121,
 123
 Rational Unified Process™ 121, 123
read 857-858
ReadWriteLock, padrão de projeto 494-
 495, 809-810
readChar 847-848, 926-927
readDouble 848-849
Reader 823-824
readInt 848-849, 923-924, 927-928

- readLine** 878-879, 937
readObject 823-824, 837-840, 843-844, 904-905, 909-910
ReadRandomFile.java 855-856
ReadSequentialFile.java 834-835
ReadServerFile.java 896-897
readUTF 926-927
real, número 106-107, 189-190
real, parte 425-426
realização 83, 592-594
realizações 83
realizações, diagrama de 593-595
realizeComplete, método 1122-1123, 1142
RealizedCompleteEvent 1122-1123
Real-Time Transport Protocol 1114-1115, 1136-1137
rebaixados, retângulos 580-581
recebendo, objeto que está 548
receber dados do servidor 910-911
receber uma conexão 901-902
receive 914-915
receive, método da classe *MulticastSocket* 947
Receiver, interface 1160-1163, 1165-1166
Receiver, método **send** de 1166
ReceivingThread.java 935-936
recepção do computador, seção de 55
recordEnable, método 1162-1163
Recruitsoft.com 1320-1321
Rectangle 426-427, 587-588
Rectangle2D 564
Rectangle2D.Double 586-589, 600-601
reculo 97-98, 157-159, 176-177
reculo, convenção de 178
reculo, tamanho de 96-97
reculo, técnicas de 120-121
recuperando memória alocada dinamicamente 769-770
recuperar memória 417-418
recuperar-se de um erro 329
recursão no texto, exemplos e exercícios de 293-294
recursão versus iteração 292-293
recursão, etapa de 285-286, 291-292
recursão, sobrecarga de 292-293
recursiva, avaliação 286-287
recursiva, chamada 285-286, 291-292
recursiva, etapa 363-364
recursivamente de trás para a frente, imprimindo uma lista 1020-1021
recursivamente, gerando números de Fibonacci 292
recursivamente, pesquisando uma *List* 1020-1021
recursivas, chamadas para o método **fibonacci** 292
recursivo factorial, método 286-287
recursivo power, método 313
recursivo, método 285-286
recursivo, programa 292
recursos para *arrays* predefinidos 1082
recursos, perda de 413-414, 742-743, 757-758
rede, aplicativos colaboradores em 814
rede, chegada de mensagem da 740-741
rede, código Morse em 965
redes 53-56, 818
redes, pacote para 271
redes, problemas em 755-756
redimensionável, *array* 893-894
redimensionável, implementação de uma *List* como um *array* 1087
redirecionar um fluxo 820-821
redundantes, parênteses 113-114
Refer.com 1320-1321
referência 153-154
referência como **null**, inicialização de uma 282-283
referência para um novo objeto 377-378
referência para um objeto 405-406
referência para um objeto de subclasse convertida implicitamente para referência a objeto de superclasse 445-446
referências para superclasses **abstract** 456-457
referências para um objeto 769-770
referências, contagem de 751
refinamento passo a passo de cima para baixo 54, 187-190, 193-194
regionMatches 514, 556-557
registra um ouvinte de eventos de janela 487-488
registrado, *listener* 615-616
registrador acumulador 368-369
registrando um tratador de eventos 281
registrar um ActionListener 695-696
registrar um ouvinte de eventos 611
registrar um WindowListener 637-638
registro 818-819
registro, chave de 819-820, 846, 886-887
registro, estrutura de 824-825
registro, tamanho de 846
regra geral 234-235
regras de precedência dos operadores 111-112, 292
“reinventando a roda” 100-102, 260-261, 270-271
relação entre números sucessivos de Fibonacci 288-289
relacionados, tipos de exceção 740-741
relacionamento “tem um” 432-433
RELATIVE 719-720
relativo, caminho 876
Reload do menu **Applet** do *appletviewer* 139-141
relógio 137-138, 986-987
relógio digital 985-986
REMAINDER 719, 719-720
remoto, computador 62-64
remove, método 1045-1046, 1052-1053, 1086, 1088-1089
removeAllElements 1041-1042
removeElement 1041-1042
removeElementAt 1041-1042
remover Strings duplicados 1100-1101
removeTargets, método 1142-1143
remuneração 210-211
repaint 566-567, 598-599, 606-607, 672-673, 677-678, 973-974
repaint, método de **JApplet** 298
repetição 182-183, 242-244, 246-247
repetição controlada por contador 182-183, 191-192, 194-197, 216-220, 292-293, 367-368
repetição definida 182-183
repetição, condição de 217-218
repetição, estrutura de 174-175, 181-182, 188-189, 292-293
repetindo um laço 194-195
replace 523-524
reprodução de áudio 139
reprodução instável 769-770
reproduzindo um AudioClip 980-981
requisito capturar ou declarar 752
requisitos 68-69, 652
reservadas, palavras 95-96
reservas, sistema de 357-358
resolução 564-565
respostas a uma pesquisa 327-328
restart, método de **Timer** 982-983
resto 110-111
resume 796
resume, método de **Thread** 800-801
reta, forma de linha 111-112
reta, linha 590
retângulo 138-139, 141-143, 168-169, 426-427, 433-434, 564, 568-569, 578-579, 664
retângulo com cantos arredondados 664
retângulo delimitador 232, 579, 580-582, 678-679, 681
retângulo delimitador de uma elipse 170-171
retângulo, símbolo 174-177, 182-183, 221-222, 229-230, 243-244
retângulo, triângulo 214
retomada, modelo de tratamento de exceção com 743-744
retomada, software de filtragem de 1319-1320
retomado 801
retomar 1314-1315, 1319-1320, 1322
retorno do carro 100-101
return, palavra-chave 175-176, 264-267, 285-286, 330
Returns, nota 1237
reutilização 100-102

- reutilização de código 1082
 reutilização de componentes 724
 “reutilizar, reutilizar, reutilizar” 68-69
 reutilizáveis, componentes 433-434
 reutilizáveis, componentes de *software* 51-52, 58-59, 270-271
 reversa, engenharia 420
reverse 530-531, 1093, 1096-1099
reverseOrder 1093-1094
 revistas 1209-1210
 RGB 572
 RGB, valores 567-568
 Richards, Martin 57-58
 Ritchie, Dennis 57-58
 RMF, arquivos 1158-1159
.rmf (extensão de arquivo Rich Music Format) 1154-1155
.rmi 980-981
 robustos, aplicativos 750-751
 robustos, programas 740
rodízio (round robin) 770-771, 773-774
 rolagem automática 629-630
 rolagem, barra de 242, 624-625, 628-629, 671-672
 rolagem, caixa de 242, 624-625
 rolagem, regras para barra de 671-672
 rolagem, seta de 624-625
 rolar 224-226, 253-255, 626-627, 629-630
 rolar através do texto 224-226
 roleta 1205
 roleta, exercício 1205
rollDice, método do programa de *craps* 281
RollDie.java 273-274, 325-326
rollover, Icon 618-619
 rosa 567-568
rotate 591-592
 rotulada, instrução 234-235, 236
 rotulada, instrução **break** 234-235
 rotulada, instrução **continue** 236
 rotulado, bloco 235-236
 rótulo 234-235, 604, 607-608
 rótulo de botão 617
 rótulo de caixa de marcação 621-622
 rótulos para marcas divisórias 680
RoundRectangle2D 564
RoundRectangle2D.Double 586-587, 590
RTP 1114-1115, 1136-1137, 1142
RTPManager 1136-1137, 1142
RTPServer.java 1136-1137
RTPServerTest.java 1142-1143
 Rumbaugh, James 69-70
run, método 770-771, 775, 778-780, 796, 924
Runnable 770-771, 796-797, 802, 924-925, 1281-1282
RuntimeException 751-755, 760-761, 995
“**rw**”, modo de abertura de arquivo 850-851, 862-863
“**rw**” para abrir o arquivo para leitura e escrita 850-851
- ## S
- saída 97-98, 100-101
 saída de uma estrutura de controle, ponto de 176-177, 243-244
 saída do bloco 282-283
 saída, *buffer* de 906
 saída, cursor de 97-98, 100
 saída, dispositivos de 55
 saída, unidade de 55
Salary.com 1326-1327
SansSerif, fonte 341-342, 573-576, 578-579
 saturação 572-573
Scoping.java 284-285
 Scott, Kendall 70
script 1342-1343
Scroll Lock, tecla 640-641
 seção administrativa do computador 55
Seção Especial:
 Construindo seu Próprio Compilador 989
 Construindo seu Próprio Computador 989
 Exercícios Avançados sobre Manipulação de *Strings* 558-559
 Seção Especial: Construindo seu Próprio Computador 366
 secundário, dispositivos de armazenamento 818
 secundário, unidade de armazenamento 55
SecurityException 1134-1135
See Also, nota 1234-1235
seek 840, 850-851, 854-855
 segunda passagem 1024-1025
 segundo grau, polinômio de 113-115
 segundo refinamento 187-188, 195-196
 segurança 64-65
 seleção 176-177, 242-244, 246-247
 seleção dupla 246-247
 seleção dupla, estrutura de 175-176, 194-195
 seleção, classificação por 363-364
 seleção, estrutura de 174-176, 292-293
 seleção, modo de 628-629
 selecionado, texto 668-669
 selecionando um item de um menu 610
SELECTED 620-621, 625-626
_self, frame de destino 896-897
SelfContainedPanel.java 675-676
SelfContainedPanelTest.java 678-679
 sem argumento, construtor 386-387, 389-390, 402-403, 442
 sem tipos, linguagem 57-58
send, método da classe **DatagramSocket** 939-940
send, método de **DatagramSocket** 914-915
SendingThread.java 943-944
SendStream, interface 1142-1143
 senha 611-612
 seno 262-263
 sensibilidade a maiúsculas e minúsculas 386
 sensível a maiúsculas e minúsculas 95-96
 sentinela, repetição controlada por 186-192, 254-255, 367-368
 sentinelas, valor de 187-188, 192
 separadora, barra 695
 separadora, linha 696-697
 separar uma instrução 104-105
separatorChar, variável de **File** 880-881
Sequence, classe 1158-1159
SequenceInputStream 823-824
Sequencer, interface 1158-1160
 seqüência 176-177, 242-247, 1007-1008, 1087
 seqüência de mensagens 350-351
 seqüência inicial de chamadas de métodos de um *applet* 689-690
 seqüência, diagrama de 83-84, 123-124, 349-350, 805-807
 seqüência, estrutura de 174-175, 187-188
 seqüenciador 1155, 1158-1159, 1193-1194
 seqüenciador, gravação com 1162-1163
 seqüencial, arquivo de acesso 818-820, 822-825, 828-829, 846, 901
 seqüencial, código 454-455
 seqüencial, execução 173-174
Serializable 823-824, 827-828, 1238-1240
Serif, fonte 341-342, 573-575, 577-578, 719-720
Server 913-914
Server.java 902-903, 912
ServerSocket 900-903, 918-919, 923-924, 931-933
 serviço 299-300, 380-381, 465
 serviço sem conexão 891-892
 serviços de uma classe 415-416
 servidor de uma computação cliente/servidor sem conexão usando datagramas, lado do 912
 servidor de uma conexão cliente/servidor com fluxo de soquete, parte 902-903
 servidor espera conexões de clientes 900-901
 servidor, endereço na Internet do 901
 servidor, número de porta 911-912
 servidor, objeto 548
Set 1082-1083, 1087, 1100-1103
set 1091-1092
set, método 381-382, 391, 394-396
 seta 169-170, 229-230
 seta para a direita, tecla 681
 seta para baixo, tecla 681

setAlignment, método 644-645
setBackground 570-572, 590-591, 628-629, 674, 682-683, 694, 698-699
setBounds, método 1166
 setCaretPosition 906, 911-912, 917-918, 927-928
setColor, método 567-570, 575-576, 579-580, 583-584, 587-588, 590-592
setConstraints 718-719
setContentDescriptor, método 1142
setCurrentChoice 687-688
setCursor 899-900
setDaemon 795-796
setDefaultClose_Operation 684-685
setEditable 612-613
setEditable, método de *JTextField* 281
setElementAt 1041-1042
setEnabled 641
setErr 820-821
setFieldValues 837-838
setFileSelectionMode 828-829, 836-837, 842-843, 848-849, 852-853, 857-858
setFixedCellHeight 630-631
setFixedCellWidth 630-631, 631-632
setFont 573-575, 577-578, 620-621, 694
setFont, método de *JTextArea* 341-342, 345-346, 352-353
setForeground 694
setHeight 687
setHorizontalAlignment 609-610
setHorizontalScrollBarPolicy 671-672
setHorizontalTextPosition 608-609
setIcon 608-610
setIn 820-821
setInverted 681
setJMenuBar 690-693, 695-696, 704-705, 733
setJMenuBar, método de *JFrame* 695-696
setLayout 606-607, 612-613, 643-644, 645-648, 650-651, 707-708, 710-711
setLayout, método de *Container* 280-281
setLayout, método de *JApplet* 280-281
setLength, método 847-848
setLineWrap 671-672
setLineWrap, método da classe *JTextArea* 953
setListData 630-631
 setLocation 684-685
setLookAndFeel 701-704
setMajorTickSpacing 682-684
setMaximumRowCount 625-626
setMaxPriority 802
setMediaTime, método 1123-1124
setMessage 1166
setMnemonic 691-696
setName 771-772
setOpaque 672-673
setor 582-583
setOut 820-821
 setPage, método de *JEditorPane* 899-900
setPaint 587-589
setPaintTicks 682-684
setPriority 773-774
setRolloverIcon 617-618
setSeed 1057-1058
 setSelected 692-693
 setSelected, método de *AbstractButton* 696-697
setSelectionMode 627-628, 630-631
setSequence, método 1158-1159
setSize 590-591, 608-609, 684-685, 687-688
setSoTimeout, método da classe *MulticastSocket* 947
setSoTimeout, método da classe *Socket* 937
setStroke 587-590
SetTest.java 1101
setText 545-546, 608-609, 671-672
setText, método de *JTextArea* 224-226, 331-332
setTitle 638-639, 679-680
setToolTipText 608-609
setVerticalAlignment 609-610
setVerticalScrollBarPolicy 671-672
setVerticalTextPosition 608-609
setVisible 646-647
setVisible, método de *Component* 684-685
setVisibleRowCount 627-628, 630-631
setWidth 687
setWrapStyleWord, método da classe *JTextArea* 953
Shape, hierarquia de classes 434-436, 504-505
Shape, objeto 589-590
Shape.java 465-466, 472-473
Shapes.java 587-588
Shapes2.java 590-591
SharedCell.java 781-782, 786-787, 794-795
shareware 59-60
shell 97-98
shell em UNIX, *prompt* do 62-64
shell script 104-105
shell, ferramenta do 97-98
Shift 642-643
Short 488-489, 1042
short, palavra-chave 175-176
short, promoções 269-270
short, tipo primitivo de dado 226
ShortMessage, classe 1159-1160, 1166
ShortMessage.NOTE_OFF 1181
ShortMessage.NOTE_ON 1166, 1181
ShortMessage.PROGRAM_CHANGE 1166
show 684-685, 700
ShowColors.java 568-569
ShowColors2.java 570
showDialog 570-571
showDocument 892-893, 896
showInputDialog, método de *JOptionPane* 105, 107-108, 117-118, 149-151, 154, 185-186, 192, 196-197, 267-268
showMessageDialog, método de *JOptionPane* 103-104, 108-109, 116, 148-149, 196-197
showOpenDialog 836-837, 839-840, 842-843
showSaveDialog 833, 848-849
shuffle, algoritmo 1093
Silicon Graphics 1113
símbolo 1337
símbolo conectador 175-176
símbolo de ação 175-176
símbolo de decisão 162-163, 169-170, 175-177, 182-183, 221-222, 229-230, 255-256
símbolos, tabela de 1024-1025
Simple, comandos do 1022
Simple, compilador de 1030-1031
Simple, interpretador do 1032
Simple, linguagem 1022
SimpleGraph, *applet* 138-139
SimplePlayer 1113-1114
SimplePlayer.java 1114-1115
simples, condição 236
simples, estrutura de seleção 175-176
simples, fluxograma mais 243-245
simples, herança 431-432
simples, lista de seleção 626-627
simples, número em ponto flutuante com precisão 149-151
simples, seleção 246-247
Simpletron Machine Language (SML) 366, 989
Simpletron, simulador do 368-370, 987, 989-990
simulação 272, 349-350
simulação de embaralhamento e distribuição de cartas 544-545, 1094-1095
simulação: a tartaruga e a lebre 365, 601-602, 1204-1205
simulado, teclado de piano 1155
simulador 123-124, 202-203, 366
simulador de computador 368-369

- simular lançamento de moeda 312-313
 simular um clique com o botão direito do mouse em um mouse com um botão 640-641
 simular um clique do botão do meio do mouse em um mouse com um ou dois botões 640-641
sin, método de **Math** 262-263
 sinal de menor (<) 146-147
 sinalização, valor de 187-188
Since, nota 1237-1238
 síncrona, chamada 349-350, 802-804
 sincronização 777-778, 787-788, 800-804, 1180-1181
 sincronização, empacotadoras de 1104-1105
 síncrono, erro 740-741
SINGLE_INTERVAL_SELECTION 628-632
SINGLE_SELECTION 628-629
Single-Threaded Execution, padrão de projeto 494-495, 809-810
Singleton, padrão de projeto 494-496
 sintaxe, erro de 98-99, 145-146, 181-182
 sintetizador 1154-1155, 1162-1163
 sistema 81-85, 123-124, 247-248, 298-299, 347-348, 420-421
 sistema de coordenadas 564-566
 sistema de gerenciamento de banco de dados (DBMS) 819-820
 sistema de numeração binário (base 2) 1217
 sistema de reserva de linhas aéreas 357-358, 815, 822-823
 sistema de *software* em camadas 456-457
 sistema de *software* para comando e controle 61
 sistema, caixa do 652-653
 sistema, comportamento de um 123-124
 sistema, estrutura do 123-124
 sistema, requisitos do 652
 sistema, responsabilidades do 123-124
 sistema, serviço do 905-906
 sistemas de numeração, Apêndice 1216
 sistemas para comando e controle 769
site do cliente 66
sites de empregos abrangentes 1314
SiteSelector.html 892-893
SiteSelector.java 894-895
size 1042, 1045-1046, 1071-1072, 1087-1089, 1103-1104
SkillsVillage.com 1320-1321
sleep 771-772, 774-777, 779-780, 783-785, 789-790
SliderDemo.java 682-683
 Smalltalk 58-59
SML 989
 sobrecarga 376-377, 439-440
 sobrecarregados, construtores 386-387, 437-438
 sobrecarregados, métodos 294-295, 302-303, 386-387
 sobrepostos, blocos de construção 244, 246
 sobrescrever (substituir ou redefinir) comportamento 144
 sobrescrever 438-439, 489-494, 593-594
 sobrescrever o método **toString** 439-440
 sobrescrever um método de superclasse 377-378, 438-439, 439-440
 sobrescrever uma definição de método 297
Socket 900-901, 906-907, 911-912, 921-922, 924-925, 942-943, 956-957
SocketException 912-913
SocketImpl 956-957
SocketMessageManager.java 940-941
SocketMessengerConstants.java 933-934
software 50-51, 54
software, modelo em 368-369
software, patrimônio em 68-69
software, reuso de 59-60, 262-263, 380-383, 401, 431, 446-447
software, simulação baseada em 366
software, simulador em 120-121
Solaris 770-771
 solicitar que o usuário digite um valor 154
 sólido com uma ponta de seta anexada, círculo 247-248
 sólido, arco 582-583
 sólido, círculo 248-249
 sólido, polígono 584-585
 sólido, retângulo 579
 som 83-84, 298, 967, 1312
 som, banco de 1165-1166
 som, mecanismo de 980-981
 som, placa de 979-980
 somar os elementos de um *array* 323-324
 sons de muitos animais 1181
 soquete 891-892
 soquete de datagrama 891-892
 soquetes, comunicação baseada em 891-892
sort 1082-1085, 1093-1094, 1099-1100
Sort1.java 1093
Sort2.java 1093-1094
SortDemo, applet 138-139
 sorte 272
SortedMap 1102-1103
SortedSet 1100-1103
SortedSetTest.java 1101-1102
SoundBank 1162-1163
SoundEffects 1181-1183, 1190-1193, 1306-1310, 1312
SoundEffects.java 1191-1192
SOUTH 633-634, 715
South 646
SOUTHEAST 715
SOUTHWEST 715
spooling 1004-1005
SpreadSheet, applet 138-139
sqrt, método de **Math** 261-263, 269-270
square, método 264-265
SquareInt.java 264-265
Stack 1000-1001, 1035, 1045-1046, 1082
StackComposition.java 1003-1004
StackInheritance.java 1001-1002
StackInheritanceTest.java 1002-1003
StackTest.java 1042
start 144-146, 154-155, 689-690, 771-772, 776-777, 781-782, 786-787, 795-796, 801, 924-926, 973
start, método de **DataSink** 1134-1135
start, método de **JApplet** 298
start, método de **SendStream** 1142-1143
start, método de **Sequencer** 1158-1159
startRecording, método 1162-1163
startsWith, método de **String** 516-517
State, padrão de projeto 494, 498-499, 730-731
stateChanged 683-684
static de classes empacotadoras de tipo, métodos 488-489
static não pode acessar membros de classe não **static**, método 417-418
static para manter a contagem do número de objetos em uma classe, variável de classe 415-416
static têm escopo de classe, variáveis de classe 414
static, classe interna 488-489, 586-587
static, membro de classe 414
static, método 103-104, 154-155, 265-266, 281-282, 804-805, 1192-1193
static, palavra-chave 176-177
static, variável de classe 414, 415-416
StaticCharMethods.java 536
StaticCharMethods2.java 538-539
stop 796, 980-982, 983-984
stop, método de **DataLine** 1151-1152
stop, método de **JApplet** 298
stopRecording 1162-1163
store 1055
Strategy, padrão de projeto 494-495, 498-499, 730-731
StreamTokenizer 1019-1020, 1024-1025, 1110-1111
string 97-98, 271
String 105-107, 192-193, 509-510
 string como objetos da classe **String**, literais 192-193

- string** de caracteres 97-98
string de caracteres 97-98, 144-145
string delimitador 541-542
String duplicado 1100-1101
String, argumento 154-155
String, *array* 319-320
String, classe 144-145
string, constante 509-510
String, construtores 510-511
String, desempenho na comparação de 526
string, literal 97-98, 509-510
String, métodos de comparação de 558
String, métodos de pesquisa da classe 519-520
String, referência 511-512
StringBuffer, classe 509, 511-512, 527-528
StringBuffer, construtores 528-529
StringBuffer, métodos *insert* da classe 534-535
StringBufferAppend.java 532-533
StringBufferCapLen.java 529-530
StringBufferChars.java 530-531
StringBufferConstructors.java 528-529
StringBufferInsert.java 534-535
StringConcat.java 521-52
StringConstructors.java 509-510
StringHashCode.java 518-519
StringIndexOutOfBoundsException, classe 521-522, 530-531
StringIntern.java 526
StringMisc.java 512-513
StringReader 824-825
strings constantes 528-529
strings, concatenação de 109, 118-119, 234-235, 378-379
strings, operador de concatenação de, + 109
StringStartEnd.java 516-517
StringTokenizer 271, 509, 541-542, 558, 937-938, 942-943
StringValueOf.java 524-525
StringWriter 824-825
Stroke, objeto 589-590
Stroustrup, Bjarne 58-59
StudentPoll.java 327-328
subclasse 143-144, 375-376, 431-436, 489-494, 549-550, 552, 725-728, 1182-1183, 1242, 1287-1288
subclasse de **Thread** 780
subclasse, construtor de 437, 442
subclasse, referência para 436-437
subdiretório 138-139
subida 576-579
sublista 1091-1092
submenu 690-691
subscrito 317-318, 342
subsistema 123-124, 957
substantivo em um definição de problema, frase com 1181-1182
substantivo em uma definição de problema 68-69
substantivo em uma definição de problema, frases com 160-161
substantivo na definição do problema, frases com 202-203
substantivos 202-203
substantivos em um documento de requisitos de um sistema 373-374
substring, método de **String** 521-52
SubString.java 521
subtração 111-112
Sum.java 222-223
sumário dos exemplos e exercícios de recursividade no livro 293-294
SumArray.java 323-324
Sun Audio 1113, 1148
Sun Audio, formato de arquivo 980-981
Sun Microsystems xxxv, 50-51, 53-54, 70-71, 94-95
Sun Microsystems, Inc. 1337-1338
Sun, compilador HotSpot da 66
Sunsite Japan Multimedia Collection 1181
super 435-437, 439-440, 442-444
super, palavra-chave 176-177
superclasse 143-144, 375-376, 430-436, 488-491, 728-729, 1242, 1258-1259, 1274-1275
superclasse abstrata 455-458, 465
superclasse é sobreescrito em uma subclasse, método da 442-443
superclasse Exception 740-741
superclasse finalize 442
superclasse para fazer referência a um objeto de subclasse, referência de 454-455
superclasse, construtor da 442
superclasse, construtor de 437
superclasse, construtor *default* 442
superclasse, construtor *default* da 437-438
superclasse, membros **private** da 435-436
superclasse, membros **protected** da 435-436
superclasse, objetos de 436-437
superclasse, referência para 436-437, 440-442, 445-446, 454-455, 462-463
superclasse, sintaxe de chamada para um construtor de 438-439
supercomputador 54
superfície do cilindro, área da 465
supermercado, simulação de 1019-1020
suspend 796, 800-801
suspender a execução de um *applet* 298
.swf 1113
.swf (extensão de arquivo Flash) 1113, 1194-1195
Swing Event Package 271
Swing, mecanismo de pintura do 672-673
Swing, pacote de componentes GUI 270-271, 605-606
SwingConstants 608-609, 681-682
SwingConstants.BOTTOM 608-609
SwingConstants.CENTER 608-609, 694
SwingConstants.HORIZONTAL 682-683
SwingConstants.LEFT 608-609
SwingUtilities 703-704, 787-788, 800-801
SwingUtilities, classe 953-954
switch, estrutura de seleção múltipla 175-177, 226, 228-230, 242-243, 246-247, 274-275, 432-433
switch, lógica de 229-231, 454-455
SwitchTest.java 227
Sybase, Inc. 1337-1338
Symantec Visual Cafe 61-62
SymbolTest, *applet* 138-139
synchronized 803-804, 809-810, 919-920, 922-923, 995, 1010-1011, 1104-1105
synchronized, bloco 1263-1264
synchronized, bloco de código 800-801
synchronized, blocos de código 777-778
synchronized, método 777-778, 784-786, 802-807, 1282, 1288
synchronized, palavra-chave 176-177, 782-783
Synthesizer, interface 1162-1163
SysexMessage, classe 1159-1160
System, classe 104-105
System.err (fluxo de erros padrão) 820-823
System.exit(0) 102-105, 183-184, 480-481, 485-486
System.gc() 416-417, 444-445
System.in (fluxo de entrada padrão) 820-821
System.out (fluxo de saída padrão) 820-823
System.out.print, método 100, 107-108
System.out.println, método 97-98, 100, 107-108
SystemColor 589-590

T

- 3-D com múltiplas *threads*, jogo-da-velha 965
Tab, tecla 96-97
tabela 342
tabela, elemento de 342

- tabela, formato de 320-322
 tabela-verdade 237-238
 tabela-verdade para o operador `!`, 239-240
 tabela-verdade para o operador `&&`, 237-238
 tabela-verdade para o operador `^`, 239-240
 tabela-verdade para o operador `||`, 238-239
 tabulação 100-101, 140-141
 tabulação, caracteres de 95-96
 tabulação, pontos de 96-97, 100-101
 tabuleiro de damas 213-214
tailSet 1101-1102
 tamanho 110
 tamanho da área de exibição do *applet* 146-147
 tamanho de um *array* 329
tan, método de **Math** 262-263
 tangente 262-263
 tarefa 769-770
 tartaruga e a lebre 365, 601-602, 814, 1204-1205
 tartaruga, gráficos de 358, 601-602
 TCP (Transmission Control Protocol) 891-892
 tecla de ação 640-641
 tecla de seta 640-641
 tecla, pressionamento de 740-741
 teclado 54, 105, 184-185, 188-189, 604-605, 639-640
 teclas, constantes para 642-643
 teclas, eventos de 616, 640-641, 675-676
 tela 55, 184-185
 tela a cores 145-146
 tela, captura de 140-141
 tela, cursor da 100-101
 tela, programa gerenciador de 456
 telefone, gerador palavras com números de telefone 888-889
 telefone, programa gerador de palavras com números de 888-889
 telefônico, sistema 912
Template File, lista escamoteável 157-159
Template Method, padrão de projeto 494-495, 498-499, 731-732
 tempo de duração 282-283
 tempo, compartilhamento de 55-56
 tempo, período de 770-774, 813-814
 temporário 192-193
 tentativa de conexão 901
 teoria da complexidade 292-293, 1093
 terminações de linha 589-590
 terminal 55-56
 terminar com sucesso 104-105
 terminar um aplicativo 186-187, 689-690, 695-696
 terminar um programa 102-105, 860-861
 terminar um programa anormalmente 329
 terminar uma estrutura de laço aninhada 234-235
 terminar uma *thread* 796
 término 329
 término de um bloco, chave à direita `{}` de 283-284
 término, fase de 189-190
 término, faxina de 413-414
 término, modelo de tratamento de exceções com 743-744
 término, teste de 292-293
 termário, operador 178-179
Test.java 440-441, 444-445, 448-450, 452-453, 462-463
 testando 454-455
 teste e depuração, dicas de 59-60
Text Package 271
TextAreaDemo.java 668-669
TextFieldTest.java 611-612
 texto piscando 137-138
 texto que pula 138-139
 texto, análise de 558-559
 texto, arquivo de 145-146
 texto, campo de 102-104, 107-108, 604-605
 texto, editor de 97-98, 509
TexturePaint 564, 588-589
 The Complete 2 Java Training Course Fourth Edition 50-51
 The Complete UML Training Course 70
 The Diversity Directory 1319-1320
 The National Business and Disability Council (NBDC) 1320-1321
 "The Twelve Days of Christmas" 229-230, 256-257
 The Unified Modeling Language User Guide 70
this 380-381
this, palavra-chave 176-177, 281, 297, 407-408, 417-418, 488-489, 633-634
ThisTest.java 407-408
 Thompson, Ken 57-58
thread 298, 445-446, 802-804, 809-810, 1256-1257, 1263-1264, 1273-1274, 1281-1282
Thread 769-771, 774-779, 794-797, 800-802, 924-925, 932-933, 937-938, 1287-1288
thread daemon 795-796
thread de execução 769
thread more 776-777, 780
thread morta 771-772
Thread.MAX_PRIORITY 772-773
Thread.MIN_PRIORITY 772-773
Thread.NORM_PRIORITY 772-773
ThreadDeath 769-770
ThreadDeath, objeto 769-770
ThreadGroup 769-772, 801
ThreadLocal 769-770
threads cooperativas de igual prioridade 773-774
threads modificando um *array* de células compartilhadas 794-795
threads por prioridades, escalonamento de 774-775
threads sendo executadas assincronamente 787-788
threads simultâneas 782-783
threads simultâneas produtora e consumidora 782-783
threads, estados de 771-772
threads, grupo de 801
threads, segurança quanto a 1104-1105
threads, sincronização de 800-801
ThreadTester.java 775
ThreeDimensionalShape 504-505
throw, instrução 743-744, 750-751
throw, palavra-chave 176-177, 743-744, 751-753
Throwable 743-744, 761-762
throws, cláusula 742-743, 752
throws, cláusula de um método de subclasse que sobrescreve um método de superclasse 752-753
throws, palavra-chave 176-177
TicTacToe 426-427
TicTacToe, *applet* 136-137, 139
TicTacToeClient 918, 924, 964-965
TicTacToeClient.java 924-925
TicTacToeServer 918, 921-924, 964-965
TicTacToeServer.java 918-919
Time 1123-1124
Time.java 477-478
Time1.java 374-375, 383-384
Time2.java 387-388
Time3.java 391-392
Time4.java 409-410
Timer 600-601, 973-974, 1180-1181, 1307-1309
Timer, retardo do 1180-1181
TimesRoman, fonte 341-342
TimeTest.java 378
TimeTest2.java 381-382
TimeTest3.java 385
TimeTest4.java 389-390
TimeTest5.java 395-396
TimeTest6.java 412-413
TimeTestWindow.java 479-480, 484
 tipo 109
 tipo da exceção disparada 742-743
 tipo de atributo 205-206
 tipo de dado 106-107
 tipo de dado abstrato (ADT) 374-375, 418-419
 tipo, classe empacotadora de 154-155, 205-206, 488-489, 535-536, 1035-1036, 1042
 tipográfico, sistema 509
 tipos primitivos de dados 106-107, 418-419
 título da janela interna, barra de 707
 título, barra de 104-105, 482-483, 638-639, 684-685
 título, *string* da barra de 109
toArray 1091-1092

- toCharArray**, método de `String` 363-364, 523-524
todo/parte, relacionamento 162-163
ToggleButton 617
token 271, 541-542
tokens, separar em 541-542
TokenTest.java 541-542
toLowerCase, método de `String` 523-524, 537-538
toLowerCaseTokenTest.java, método de `String` 536-537
tom 572-573
tomando decisões 120-121, 157-159
top.frame de destino 564-565, 896-897
topo 187-188, 194-195, 418-419, 1045-1046
torres de Hanói 313, 601-602, 1204-1205
torres de Hanói, exercício 601-602, 1204-1205
torta, arco em forma de 590
torta, gráfico de 601-602
toString invocado sobre qualquer objeto `Throwable` 744-745
toString, método 375-378
toString, método de `Integer` 281-282
toUpperCase, método de `String` 523-524, 536-538
Track, classe 1159-1160, 1162-1163
Track, método `get` de 1159-1160
TrackControl, interface 1142
tradução 56-57
tradutor, programa 56-57
transação, registro de 887-888
transações, aplicativos de processamento de 822-823, 846
transações, arquivo de 886-887
transações, programa de processamento de 846, 860-861
transferência de controle 173-174, 367-370
transição 247-250
transient 827-828
transient, palavra-chave 176-177
translate 590-591
transmissão baseada em fluxo orientada para conexão 911-912
transmissão sem conexão com datagramas 911-912
Transmitter, interface 1159-1160, 1162-1163, 1165-1166
transparência 672-673
transparentes, componentes GIF do Swing 678-679
trapézio 433-434
tratador de exceções *default* 742-743, 761-762
Tree, *link* 1238-1240
Tree.java 1007-1009
TreeMap 1102-1103
TreeSet 1100-1103
- TreeTest.java** 1010-1011
treinamento, curso de 160
três botões, mouse com 639-640
triângulo 213-214
tridimensionais de alta resolução, gráficos a cores 967
tridimensional, aplicativo 967
tridimensional, forma 138-139
tridimensional, retângulo 579
tridimensional, visão 138-139
trigonométrica, tangente 262-263
trigonométrico, co-seno 262-263
trigonométrico, seno 262-263
trilhões de instruções por segundo, computadores de 54
trim, método de `String-Buffer` 523-525
trimToSize 1035-1036
trocar 333-335
true, palavra-chave 176-177
truncar 110-111, 192-193, 833-834
try completado com sucesso 760-761
try, bloco 742-744, 750-751, 776-777
try, palavra-chave 176-177
tutoriais 1209-1210
Twelve Days of Christmas 256-257
TwoDimensionalShape 504-505
Two-Phase Termination, padrão de projeto 494-495, 809-810
TYPE_INT_RGB 587-588
- U**
- U+yyyy** (convenção de notação do Unicode) 1338-1339, 1343-1344
UDP 891-892
UIManager 701-704
UIManager.LookAndFeelInfo 700-701
último a entrar, primeiro a sair (LIFO), estrutura de dados 418-419, 1000-1001
um para um, mapeamento 1102-1103
um, complemento de 1066-1067, 1223
uma instrução por linha 118-119
UML 1.3, documento de especificações da 70
UML 66-67, 69-70, 81-85, 120-121, 123-124, 162-165, 205-206, 489-491, 493, 592-594, 652, 802-804, 1193-1194, 1251, 1288, 1311-1312
UML Distilled Second Edition 70
UML Partners 69-70
UML, anotações na 550-552
UML, diagrama da 123-124
únario, operador 192-193, 239-240
únario, operador de coerção 192-193
Undo, botão 665-666
união de dois conjuntos 427-428
única linha, comentário de uma, // 94-95, 98-99, 106-107, 141-143
única, linguagens com uma única *thread* 769
- unicast** 939-940
único usuário, processamento em lotes com um 55
Unicode 202-203, 515, 823-824, 1094-1095
Unicode Consortium 1337-1338, 1343-1345
Unicode Transformation Format (UTF) 1343-1344
Unicode, base de projeto do padrão 1337-1338
Unicode, caractere 818-819
Unicode, conjunto de caracteres 133-134, 509-510, 520, 536
Unicode, padrão 1337, 1343-1345
Unicode, valor do caractere digitado em 642-643
unidade central de processamento (CPU) 55
unidade lógica e aritmética (ALU) 55
unidimensional, *array* 342
Unified Modeling Language (UML) 66-67, 69-70
Uniform Resource Identifier (URI) 892-893
Uniform Resource Locator (URL) 892-893
uniforme (base de projeto do Unicode) 1337-1338, 1343-1344
universal (base de projeto do Unicode) 1343-1344
universal (princípio de projeto do Unicode) 1337-1338
universal, formato de hora 375-376, 378-379
UNIX 56-57, 61-62, 97-98, 104-105, 136-137, 565-566, 700
UNIX, aparência e comportamento do 668-669
UNIX, plataforma 700
UnknownHostException 901-902
uns, posição dos 121-127
UnsupportedAudioFileException 1150-1151
UnsupportedFormatException 1142-1143
UnsupportedOperationException 1086, 1104-1105
update 566-567, 606-607, 672-673, 1151-1152
update, método de `JApplet` 298-299
updateComponentTreeUI de `SwingUtilities` 701-702
updateComponentTreeUI, método de `SwingUtilities` 703-704
UpdateThread.java 788-789
URI (Uniform Resource Identifier) 892-893
URI 893-894
URL (Uniform Resource Locator) 892-893
URL 957-958, 969-970, 979-980
URLStreamHandler 957-958

- User Datagram Protocol 891-892
UsingArrays.java 1082-1083
UsingAsList.java 1085-1086
UsingExceptions.java 758-760, 762
UsingToArray.java 1091-1092
 uso, caso de 83-84, 652-654
 uso, diagrama de casos de 83-84, 123-124, 652-654
 usuário 298-299
 usuário, classes definidas pelo 95-96
 usuário, evento da interface com o 288-289
 usuário, interface 83
 usuário, interface com o 959-960
 usuário, síntese pelo 1155
 usuário, tipo definido pelo 67-68, 205-206, 373-374
 UTF (Unicode Transformation Format) 1343-1344
 UTF-16 1337-1338, 1343-1344
 UTF-32 1337-1338, 1343-1344
 UTF-8 1337-1338, 1343-1344
 utilitário, método 375-376
 utilitários, pacote de 271
- V**
- vagos, *bits* que ficaram 1070-1071
 validade, verificação de 391-392, 398-399
validate 649-650
validate, método de *Container* 650-651
 valor 110, 1052-1053
 valor absoluto 262-263
 valor de um parâmetro 892-893
 valor *default* 152-153, 202
 valor do cheque 559-560
 valor do código 1338-1339, 1341-1344
 valor do inteiro mais próximo 309
 valor fictício 187-188
 valores duplicados 1013-1014
valueof 524-525, 929-930
 variáveis automáticas devem ser inicializadas 282-283
 variável 68-69, 106-107, 154, 373-374
 variável com escopo de classe ocultada por variável com escopo de método 380-381
 variável constante 229-231, 279-280, 323-324
 variável contador 228-229
 variável de condição 782-783
 variável de controle 218-220
 variável não pode ser modificada 401
 variável, declaração de 152-153
 variável, escopo de 219-220
 variável, nome de 110
 varredura 581-582
 varrendo imagens 55
 varrer no sentido anti-horário 581-582
Vault.com 1317
 vazia, instrução (;) 181-182
 vazia, instrução (ponto-e-vírgula sozinho) 231-232
 vazia, instrução ; 118-119
 vazia, sequência 1162-1163
 vazio, conjunto 427-428
 vazio, corpo 635
 vazio, string "" 117-118
Vector 317, 665-666, 754-755, 893-895, 1040-1041, 1082, 1087, 1098-1099
vectorTest.java 1035-1036
 velocidade 1185-1187
 velocidade de animação 140-141
 velocidade de piscar 985-986
 verbo em definição de problema, frase com 298-301
 verbo em uma definição de problema 67-68
 verbo em uma definição de problema, frase com 347-348
 verbos um um documento de requisitos de um sistema 373-374
 verificador de *bytecode* 64-65
 verificador ortográfico, projeto 561-562
Version, nota 1237-1238
VERTICAL 715
 vertical, coordenada 564
 vertical, espaçamento 176-177
 vertical, espaço vazio 647-648
 vertical, orientação 681
 vertical, rolagem 670-671
 vertical, suporte 709-710
VERTICAL_SCROLLBAR_ALWAYS 671-672
VERTICAL_SCROLLBAR_AS_NEEDED 671-672
VERTICAL_SCROLLBAR_NEVER 671-672
vi 61-62
 vídeo 51-52, 967, 982
 vídeo game 272, 414, 1204-1205
 vídeo games, exercício 1204-1205
 vídeo para Windows 1123-1124
 vídeo, clipes 769-770, 1181
 vídeo, gravação de 55
View 604
 vinculação tardia 465
 vinculando o servidor à porta 900-901
 vincular o servidor a uma porta 912
 vírgula (,) 222-223
 vírgula em uma lista de argumentos 103-104
 virtual, código de chave 641-642
 visão (na arquitetura MVC) 957-959
 visão 83-84, 102-103, 120-121, 123-124, 721-725, 1085-1086, 1181-1183, 1190-1191, 1248-1249, 1251, 1256-1257, 1312
 visibilidade 420
Visitor, padrão de projeto 498-499
 visível, área 155-156
 visual, realimentação 618-619
 visualização antecipada, taxa de *frames* 1135-1136
 visualizando a recursão 314
 visualizar uma forma de ângulos diferentes 138-139
Viissides, John 70-71
void, palavra-chave 97-98, 144-145, 176-177, 266
volatile, palavra-chave 176-177
 volume de uma esfera 306-307, 308-309
 vôo, simulador de 505, 1204-1205
- W**
- wait** 772-774, 777-778, 784-785, 800-801, 809-810, 814
WAIT_CURSOR 898-899
waitForPackets 912-913
.wav 1113, 1148
.wav, arquivo 1190-1191
.wav, extensão 980-981
.wav (extensão de arquivo WAVE) 1113, 1148
 WAVE, áudio 1113, 1148
 Web com animação, página da 973
 Web, navegador da 136, 146-147, 155-156, 685-686, 896, 1238-1240
 Web, página da 801, 1230-1231
 Web, recursos na 982, 1181
 Web, servidor da 66, 905-906
 WebHire 1316-1318
weightx 714-715, 718-720
weighty 714-717, 718-720
Welcome1.java 94-95
Welcome2.java 100
Welcome3.java 100-101
Welcome4.java 102-103
WelcomeApplet.html 146-147
WelcomeApplet.java 142
WelcomeApplet2.html 148-149
WelcomeApplet2.java 148-149
WelcomeLines.htm 149-150
WelcomeLines.java 149-150
West 646-647
WEST 646-647, 715
while, condição da estrutura de repetição 191-192
while, estrutura de repetição 172, 176-177, 181-182, 185-186, 188-190, 242-243, 246-247
WhileCounter.java 216-217
widgets 604-605
 Win32 770-771
Window 684-685
window gadgets 604-605
Window, classe 605-606, 684-685
windowActivated 685-686
WindowAdapter 635
windowClosed 685-686
windowClosing 485-486, 685-686
WindowConstants 684-685

WindowConstants.DISPOSE_ON_CLOSE 684-685
windowDeactivated 685-686
windowDeiconified 685-686
windowIconified 685-686
WindowListener 485-486, 635, 685-686
windowOpened 685-686
Windows 56-57, 104-105, 136-137
Windows 2000 56-57
Windows 95/98 61-62, 770-771
Windows Notepad 61-62
Windows NT 770-771
Windows Performance Package 1113-1114
Windows Wave, formato de arquivo 980-981
Windows, aparência e comportamento do 668-669, 700-701
WireFrame, applet 138-139
wireless application protocol (WAP) 1323-1324
WirelessResumes.com 1323-1324
Wirth, Nicklaus 60-61
WORA (Write Once Run Anywhere) 202
word wrap 671-672
WorkingSolo.com 1324-1326
World Wide Web 51-52, 58-59, 61, 136-137, 159-160, 604, 769-770, 818, 892-893, 973-974
World Wide Web Consortium (W3C) 896-897
World Wide Web, navegador da 62-64, 100-102, 146-147
World Wide Web, site na 102-103
wrap 671-672

wrap de palavras automático 671-672
write 849-850, 854-855
writeBytes 823
writeChar 922-923
writeChars 847-848, 848-849
writeDouble 847-848, 848-849
writeInt 847-849, 922-923, 928-929
writeObject 823-824, 833-834, 904-905
Writer 823-824
WriteRandomFile.java 851-852
writeUTF 921-924
www.advantagehiring.com 1320-1321
www.advisorteam.net/AT/User/kcs.asp 1321-1322
www.careerpower.com 1326-1327
www.chiefmonster.com 1325-1326
www.deitel.com 50-51, 54, 85-86, 102-103
www.driveway.com 1322
www.etest.net 1321-1322
www.ework.com 1324-1326
www.execunet.com 1325-1326
www.freeagent.com 1324-1326
www.jars.com 160
www.jobfind.com 1318-1319
www.jobtrak.com 1322
www.midi.org 1154-1155
www.mindexchange.com 1319-1320
www.nationjob.com 1325-1326
www.omg.org 70
www.prenhall.com/deitel 54
www.recruitsoft.com/corpoVid eo 1320-1321

www.recruitsoft.com/process 1320-1321
www.review.com 1326-1327
www.sixfigurejobs.com 1325-1326
www.unicode.org 1339-1340
www.w3.org 896-897
www.webhire.com 1316-1318
www.xdrive.com 1322

X

x, coordenada 141-143, 145-146, 217-218, 228-229, 232, 564, 585-586
x, eixo dos 564
X3J11, comitê técnico 57-58
XDrive™ 1322
Xerox's Palo Alto Research Center (PARC) 58-59
xor 1071-1072

Y

y, coordenada 145-146, 228-229, 232, 564, 585-586
y, eixo dos 564
Yahoo! 1318-1319
yield 773-774, 813-814

Z

zero, contagem a partir de 218-220, 228-229, 320-321
zero, elemento de ordem 317-318

