

HERBERT SCHILDT | DALE SKRIEN

Programação com Java

UMA INTRODUÇÃO ABRANGENTE



Mc
Graw
Hill
Education



Os autores

Herbert Schildt escreve sobre programação desde 1984 e é autor de vários livros sobre Java, C++, C e C#. Seus livros de programação venderam milhões de cópias no mundo inteiro e foram traduzidos para muitos idiomas. Embora tenha interesse em todas as áreas da computação, seu foco principal são as linguagens de computador, inclusive a padronização de linguagens. Schildt tem graduação e pós-graduação pela University of Illinois, Urbana/Champaign. Ele forneceu os esboços iniciais da maioria dos capítulos deste livro.

Dale Skrien ensina matemática e ciência da computação no Colby College desde 1980 e ensina Java desde 1996. Seu interesse em ensinar os alunos não só a como programar, mas a como programar bem, levou-o à publicação de seu livro *Object-Oriented Design Using Java*, da McGraw-Hill. Ele é graduado pelo St. Olaf College e pós-graduado pela University of Illinois e pela University of Washington, instituição onde também obteve o diploma de PhD. Além das contribuições que fez no decorrer do livro, forneceu o Capítulo 16, que introduz o projeto orientado a objetos. Também forneceu os complementos online deste livro.



S334p

Schildt, Herbert.

Programação com Java [recurso eletrônico] : uma introdução abrangente / Herbert Schildt, Dale Skrien ; tradução: Aldir José Coelho Corrêa da Silva ; revisão técnica: Maria Lúcia Blanck Lisbôa. – Dados eletrônicos. – Porto Alegre : AMGH, 2013.

Editado também como livro impresso em 2013.
ISBN 978-85-8055-268-3

1. Ciência da computação. 2. Linguagem de programação – Java. I. Skrien, Dale. II. Título.

CDU 004.438Java

HERBERT SCHILDT

DALE SKRIEN
Colby College

Programação com Java

UMA INTRODUÇÃO ABRANGENTE

Tradução:

Aldir José Coelho Corrêa da Silva

Revisão técnica:

Maria Lúcia Blanck Lisbôa

Doutora em Ciência da Computação pela UFRGS
Professora do Instituto de Informática da UFRGS

Versão impressa
desta obra: 2013



AMGH Editora Ltda.

2013

Obra originalmente publicada sob o título
Java Programming: A Comprehensive Introduction, 1st Edition
ISBN 0-07-802207-X / 978-0-07-802207-4

Edição original copyright ©2012, The McGraw-Hill Companies, Inc., Nova York, Nova York 10020.
Todos os direitos reservados.

Tradução para língua portuguesa copyright © 2013, AMGH Editora Ltda., uma empresa do Grupo A Educação S.A. Todos os direitos reservados.

Gerente Editorial: *Arysinha Jacques Affonso*

Colaboraram nesta edição:

Editora: *Mariana Belloli*

Capa: *Maurício Pamplona*

Leitura final: *Fernanda Vier*

Editoração eletrônica: *Techbooks*

Reservados todos os direitos de publicação, em língua portuguesa, à
AMGH EDITORA LTDA., uma parceria entre GRUPO A EDUCAÇÃO S.A. e McGRAW-HILL
EDUCATION

Av. Jerônimo de Ornelas, 670 – Santana
90040-340 – Porto Alegre – RS
Fone: (51) 3027-7000 Fax: (51) 3027-7070

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.

Unidade São Paulo
Av. Embaixador Macedo Soares, 10.735 – Pavilhão 5 – Cond. Espace Center
Vila Anastácio – 05095-035 – São Paulo – SP
Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444 – www.grupoa.com.br

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Prefácio

Este livro ensina os fundamentos da programação via linguagem Java. Ele não exige experiência anterior em programação e começa com os aspectos básicos – por exemplo, como compilar e executar um programa Java. Em seguida, discute as palavras-chave, operadores e estruturas que formam a linguagem. O livro também aborda várias partes da biblioteca Java Application Programming Interfaces (APIs), inclusive o Swing, que é a estrutura usada para criar programas que têm uma interface gráfica de usuário (GUI, graphic user interface), e o Collections Framework, que é usado para armazenar coleções de objetos. Resumindo, ele foi planejado como uma introdução abrangente à linguagem Java. Como a maioria das linguagens de computador, Java evoluiu com o tempo. Quando este texto foi escrito, a versão mais recente era Java 7 (JDK 7), e essa é a versão abordada aqui. No entanto, grande parte do material também é aplicável a outras versões recentes, como a versão 6.

UMA ABORDAGEM ENCADEADA

O livro usa o que consideramos uma abordagem “encadeada”. Com esse termo queremos dizer que os tópicos são introduzidos em uma sequência coesa projetada para manter o foco de cada discussão no tópico em questão. Essa abordagem simplifica e otimiza a apresentação. Em ocasiões em que um desvio do fluxo principal da apresentação foi necessário, tentamos fazer isso de uma maneira que minimizasse a interrupção. O objetivo de nossa abordagem é apresentar a linguagem Java de uma forma que mostre claramente o relacionamento entre suas partes, e não como uma mistura de recursos desconectados.

Para facilitar o gerenciamento do material, este livro foi organizado em três partes. A Parte I descreve os elementos que definem a linguagem Java e os elementos básicos da programação. Ela começa com uma visão geral de Java seguida pelos conceitos básicos dos tipos de dados, operadores e instruções de controle. Em seguida, introduz progressivamente os recursos mais sofisticados da linguagem, como as classes, os métodos, a herança, as interfaces, os pacotes, as exceções, o uso de várias threads e os tipos genéricos. A Parte I também descreve o I/O (input/output, ou entrada/saída), porque ele faz parte de muitos programas Java, e os aspectos básicos dos applets, porque o applet é um aplicativo Java fundamental. Ela termina com um capítulo sobre o projeto orientado a objetos.

No que diz respeito à Parte I, nossa abordagem “encadeada” mantém o foco nos elementos da linguagem Java e nos fundamentos da programação, com cada nova seção se baseando no que veio antes. Sempre que possível, evitamos divagações que se afastem do tópico principal. Por exemplo, a programação de GUIs com o Swing é abordada na Parte II, em vez de ser intercalada com discussões dos conceitos básicos. Assim, mantivemos a Parte I firmemente fixada nas questões da linguagem Java e da programação.

A Parte II introduz o Swing. Ela começa com uma visão geral da programação de GUIs com o Swing, incluindo os conceitos básicos de componentes, eventos e gerenciadores de leiaute. Os capítulos subsequentes avançam de maneira ordenada, apresentando uma visão geral dos diversos componentes do Swing, seguida pelos menus, caixas de diálogo, geração de componentes e assim por diante. Essa abordagem “encadeada” tem como objetivo ajudar os alunos a integrar mais facilmente cada novo recurso ao quadro geral que estão formando da estrutura do Swing.

A Parte III examina fragmentos da biblioteca de APIs Java. Como a biblioteca de APIs é muito extensa, não é possível discuti-la em sua totalidade neste livro. Em vez disso, nós nos concentramos nas partes da biblioteca que qualquer programador Java deve conhecer. Além de abordar grandes parcelas dos pacotes **java.lang** e **java.util** (com ênfase especial no Collections Framework), também apresentamos uma visão geral sobre redes e introduzimos a API de concorrência, o que inclui o Framework Fork/Join. O material é apresentado de maneira encadeada, planejada para dar ao aluno uma visão geral sólida dos diversos elementos básicos da biblioteca.

OBJETOS LOGO, MAS NÃO TANTO

Uma das primeiras perguntas que normalmente são feitas sobre um livro de programação é se ele usa uma abordagem de apresentação dos objetos “mais cedo” ou “mais tarde” para ensinar os princípios-chave da programação orientada a objetos. Claro que “mais cedo” ou “mais tarde” pode ser um pouco subjetivo, e nenhum dos dois termos descreve precisamente a organização deste livro. A expressão que usamos para caracterizar nossa abordagem é “logo, mas não tanto”. Nossa objetivo é introduzir os objetos no momento apropriado para o aluno. Achamos que isso só deve ocorrer após o aprendizado dos principais recursos da linguagem.

Para atingirmos esse objetivo, os três primeiros capítulos se concentram nos fundamentos da linguagem Java, como sintaxe, tipos de dados, operadores e instruções de controle. Acreditamos que dominar esses elementos é uma primeira etapa necessária porque eles formam a base da linguagem e a base da programação em geral. (Em outras palavras, é difícil criar programas coerentes sem entender esses elementos.) Em nossa opinião, só depois de aprender os elementos básicos de um programa é que o aluno está pronto para passar para os objetos.

Após a apresentação dos aspectos básicos, os objetos são introduzidos no Capítulo 4, e daí em diante, os recursos, técnicas e conceitos orientados a objetos são integrados nos capítulos restantes. Além disso, os objetos são introduzidos em um ritmo moderado, passo a passo. O objetivo é ajudar o aluno a trazer cada novo recurso para o contexto, sem ficar sobrecarregado.

RECURSOS PEDAGÓGICOS

Este livro inclui vários recursos pedagógicos para facilitar e reforçar o aprendizado. Os recursos permitem que os alunos conheçam as habilidades básicas, avaliem seu progresso e verifiquem se todos os conceitos foram aprendidos.

- Principais habilidades e conceitos: cada capítulo começa com uma lista das principais habilidades e conceitos apresentados nele.
- Pergunte ao especialista: em vários pontos no decorrer do livro encontram-se caixas Pergunte ao especialista. Essas caixas contêm informações adicionais ou comentários interessantes sobre um tópico e usam um formato de pergunta/resposta. Elas fornecem informações complementares sem romper o fluxo principal da apresentação.
- Tente isto: cada capítulo contém uma ou mais seções Tente isto. São exemplos passo a passo percorrendo o desenvolvimento de um programa que demonstra um aspecto de Java relacionado ao tópico do capítulo. Normalmente, são exemplos mais longos que mostram um recurso em um contexto mais prático.
- Verificação do progresso: no decorrer de cada capítulo, verificações do progresso são apresentadas para testar a compreensão da seção anterior. As respostas a essas perguntas ficam na parte inferior da mesma página.
- Exercícios: todos os capítulos terminam com exercícios contendo questões diretas, de preenchimento de lacunas ou de “verdadeiro/falso”, além de exercícios de codificação. As respostas dos exercícios encontram-se no Apêndice C.

RECOMENDAÇÕES DA ACM

A atualização de 2008 do ACM (Association for Computing Machinery) Curricula Recommendations (<http://www.acm.org/education/curricula/ComputerScience2008.pdf>) recomenda que todos os alunos de ciência da computação sejam fluentes em pelo menos uma linguagem de programação e tenham algum conhecimento em programação orientada a objetos e dirigida por eventos. Acreditamos que os alunos que aprenderem o conteúdo abordado neste livro terão os conhecimentos e habilidades desejados. Incluímos no livro não apenas uma introdução à programação com o uso da linguagem Java, mas também uma abordagem mais ampla que abrange recursos avançados, a estrutura do Swing e partes extensas de vários pacotes importantes de API.

A primeira parte do livro aborda uma parcela significativa dos tópicos da área de conhecimento Fundamentos da Programação (PF, Programming Fundamentals) das recomendações da ACM (as principais exceções são as unidades de conhecimento FoundationsInformationSecurity e SecureProgramming). A primeira parte também inclui um capítulo sobre projeto orientado a objetos que aborda vários tópicos das unidades de conhecimento PL/ObjectOrientedProgramming e SE/SoftwareDesign. A segunda parte, que introduz a programação de GUIs com o Swing, aborda alguns dos tópicos da unidade de conhecimento HC/GUIProgramming. A terceira parte inclui, entre outros, tópicos relacionados à concorrência. Na verdade, dedicamos os Capí-

tulos 12 e 27 ao uso de várias threads e à concorrência porque acreditamos, como o ACM Curricula Recommendations discute, que a concorrência está se tornando cada vez mais relevante para a disciplina da ciência da computação.

RECURSOS ONLINE

Acesse o material complementar (em inglês) do livro no site do Grupo A:

- Entre no site do Grupo A, em www.grupoa.com.br.
- Clique em “Acesse ou crie a sua conta”.
- Se você já tem cadastro no site, insira seu endereço de e-mail ou CPF e sua senha na área “Acesse sua conta”; se ainda não é cadastrado, cadastre-se preenchendo o campo da área “Crie sua conta”.
- Depois de acessar a sua conta, digite o título do livro ou o nome do autor no campo de busca do site e clique no botão “Buscar”.
- Localize o livro entre as opções oferecidas e clique sobre a imagem de capa ou sobre o título para acessar a página do livro.

Na página do livro:

Os alunos podem acessar livremente todos os códigos-fonte dos programas apresentados no livro. Para fazer download dos códigos, basta clicar no link “Conteúdo Online”.

Os professores podem acessar conteúdo exclusivo (em inglês) clicando no link “Material para o Professor”. Esse conteúdo inclui:

- Manual de soluções para os exercícios de fim de capítulo.
- Notas do instrutor, incluindo sugestões curriculares e para o ensino de tópicos específicos.
- Exercícios complementares, que podem ser utilizados na criação de questionários e testes.
- Apresentações em PowerPoint®, que servem como orientação para o ensino em sala de aula.

Sumário

PARTE I A LINGUAGEM JAVA	1
Capítulo 1 Fundamentos da programação Java	3
ASPECTOS BÁSICOS DA COMPUTAÇÃO	4
Os componentes de hardware de um computador	4
Bits, bytes e binário	6
O sistema operacional	6
O PROGRAMA	7
LINGUAGENS DE PROGRAMAÇÃO	8
A LINGUAGEM JAVA	9
Origem da linguagem Java	10
Contribuição da linguagem Java para a Internet	11
Applets Java	11
Segurança	12
Portabilidade	12
O segredo da linguagem Java: o bytecode	12
A evolução de Java	13
AS PRINCIPAIS CARACTERÍSTICAS DA PROGRAMAÇÃO ORIENTADA A OBJETOS	14
Encapsulamento	15
Polimorfismo	16
Herança	16
O JAVA DEVELOPMENT KIT	17
UM PRIMEIRO PROGRAMA SIMPLES	18
Inserindo o programa	18
Compilando o programa	19
Executando o programa	19
Primeiro exemplo de programa linha a linha	19
TRATANDO ERROS DE SINTAXE	22
UM SEGUNDO PROGRAMA SIMPLES	23

OUTRO TIPO DE DADO	25
DUAS INSTRUÇÕES DE CONTROLE	28
A instrução if	28
O laço for	30
CRIE BLOCOS DE CÓDIGO	32
PONTO E VÍRGULA E POSICIONAMENTO	33
PRÁTICAS DE RECUO	34
AS PALAVRAS-CHAVE JAVA	36
IDENTIFICADORES EM JAVA	37
AS BIBLIOTECAS DE CLASSES JAVA	38
EXERCÍCIOS	39
Capítulo 2 Introdução aos tipos de dados e operadores	42
POR QUE OS TIPOS DE DADOS SÃO IMPORTANTES	42
TIPOS PRIMITIVOS DA LINGUAGEM JAVA	43
Inteiros	43
Tipos de ponto flutuante	45
Caracteres	46
O tipo booleano	47
LITERAIS	49
Literais hexadecimais, octais e binários	50
Sequências de escape de caracteres	51
Literais de strings	51
UM EXAME MAIS DETALHADO DAS VARIÁVEIS	52
Inicializando uma variável	53
Inicialização dinâmica	53
ESSCOPO E O TEMPO DE VIDA DAS VARIÁVEIS	54
OPERADORES	57
OPERADORES ARITMÉTICOS	57
Incremento e decremento	58
OPERADORES RELACIONAIS E LÓGICOS	59
OPERADORES LÓGICOS DE CURTO-CIRCUITO	61
O OPERADOR DE ATRIBUIÇÃO	63
ATRIBUIÇÕES ABREVIADAS	63
CONVERSÃO DE TIPOS EM ATRIBUIÇÕES	64
USANDO UMA COERÇÃO	65
PRECEDÊNCIA DE OPERADORES	67
EXPRESSÕES	69
Conversão de tipos em expressões	69
Espaçamento e parênteses	71
EXERCÍCIOS	72

Capítulo 3 Instruções de controle de programa	75
CARACTERES DE ENTRADA DO TECLADO	76
A INSTRUÇÃO if	77
Ifs ANINHADOS	79
A ESCADA if-else-if	80
A INSTRUÇÃO switch	81
INSTRUÇÕES switch ANINHADAS	84
O LAÇO for	88
ALGUMAS VARIAÇÕES DO LAÇO for	89
Partes ausentes	90
O laço infinito	91
Laços sem corpo	91
DECLARANDO VARIÁVEIS DE CONTROLE DE LAÇO DENTRO DA INSTRUÇÃO for	92
O LAÇO for MELHORADO	93
O LAÇO while	93
O LAÇO do-while	95
USE break PARA SAIR DE UM LAÇO	100
USE break COMO UMA FORMA DE goto	102
USE continue	106
LAÇOS ANINHADOS	111
EXERCÍCIOS	112
Capítulo 4 Introdução a classes, objetos e métodos	116
FUNDAMENTOS DAS CLASSES	116
Forma geral de uma classe	117
Definindo uma classe	118
COMO OS OBJETOS SÃO CRIADOS	121
AS VARIÁVEIS DE REFERÊNCIA E A ATRIBUIÇÃO	121
MÉTODOS	123
Adicionando um método à classe Vehicle	123
RETORNANDO DE UM MÉTODO	125
RETORNANDO UM VALOR	126
USANDO PARÂMETROS	128
Adicionando um método parametrizado a Vehicle	130
CONSTRUTORES	138
CONSTRUTORES PARAMETRIZADOS	139
Adicionando um construtor à classe Vehicle	140
O OPERADOR new REVISITADO	142
COLETA DE LIXO E FINALIZADORES	142
O método finalize()	143
A PALAVRA-CHAVE this	146
EXERCÍCIOS	148

Capítulo 5 Mais tipos de dados e operadores	152
ARRAYS	152
Arrays unidimensionais	153
ARRAYS MULTIDIMENSIONAIS	158
Arrays bidimensionais	158
Arrays irregulares	159
Arrays de três ou mais dimensões	161
Inicializando arrays multidimensionais	161
SINTAXE ALTERNATIVA PARA A DECLARAÇÃO DE ARRAYS	163
ATRIBUINDO REFERÊNCIAS DE ARRAYS	163
USANDO O MEMBRO length	165
O LAÇO for DE ESTILO FOR-EACH	173
Iterando por arrays multidimensionais	176
Aplicando o laço for melhorado	177
STRINGS	178
Construindo strings	178
Operando com strings	179
Arrays de strings	181
Strings não podem ser alterados	182
Usando um string para controlar uma instrução switch	183
USANDO ARGUMENTOS DE LINHA DE COMANDO	184
OS OPERADORES BITWISE	186
Os operadores bitwise AND, OR, XOR e NOT	186
Os operadores de deslocamento	191
Atribuições abreviadas bitwise	193
O OPERADOR ?	197
EXERCÍCIOS	198
Capítulo 6 Verificação minuciosa dos métodos e classes	202
CONTROLANDO O ACESSO A MEMBROS DE CLASSES	202
Modificadores de acesso da linguagem Java	203
PASSE OBJETOS PARA OS MÉTODOS	208
COMO OS ARGUMENTOS SÃO PASSADOS	209
RETORNANDO OBJETOS	212
SOBRECARGA DE MÉTODOS	214
SOBRECARREGANDO CONSTRUTORES	219
RECURSÃO	225
ENTENDENDO static	229
Variáveis estáticas	230
Métodos estáticos	232
Blocos estáticos	233

INTRODUÇÃO ÀS CLASSES ANINHADAS E INTERNAS	237
VARARGS: ARGUMENTOS EM QUANTIDADE VARIÁVEL	241
Aspectos básicos dos varargs	242
Sobrecarregando métodos varargs	244
Varargs e ambiguidade	246
EXERCÍCIOS	247
Capítulo 7 Herança	253
ASPECTOS BÁSICOS DE HERANÇA	253
ACESSO A MEMBROS E HERANÇA	256
CONSTRUTORES E HERANÇA	259
USANDO super PARA CHAMAR CONSTRUTORES DA SUPERCLASSE	261
USANDO super PARA ACESSAR MEMBROS DA SUPERCLASSE	265
criando UMA HIERARQUIA DE VÁRIOS NÍVEIS	269
QUANDO OS CONSTRUTORES SÃO EXECUTADOS?	272
REFERÊNCIAS DA SUPERCLASSE E OBJETOS DA SUBCLASSE	273
SOBREPOSIÇÃO DE MÉTODOS	278
MÉTODOS SOBREPOSTOS DÃO SUPORTE AO POLIMORFISMO	281
POR QUE SOBREPOR MÉTODOS?	283
Aplicando a sobreposição de métodos a TwoDShape	283
USANDO CLASSES ABSTRATAS	287
USANDO final	292
A palavra-chave final impede a sobreposição	292
A palavra-chave final impede a herança	292
Usando final com membros de dados	293
A CLASSE Object	294
EXERCÍCIOS	295
Capítulo 8 Interfaces	298
ASPECTOS BÁSICOS DA INTERFACE	298
criando UMA INTERFACE	299
IMPLEMENTANDO UMA INTERFACE	300
USANDO REFERÊNCIAS DE INTERFACES	304
IMPLEMENTANDO VÁRIAS INTERFACES	306
CONSTANTES EM INTERFACES	314
INTERFACES PODEM SER ESTENDIDAS	316
INTERFACES ANINHADAS	317
CONSIDERAÇÕES FINAIS SOBRE AS INTERFACES	318
EXERCÍCIOS	318

Capítulo 9 Pacotes	321
ASPECTOS BÁSICOS DOS PACOTES	321
Definindo um pacote	322
Encontrando pacotes e CLASSPATH	323
Exemplo breve de pacote	323
PACOTES E O ACESSO A MEMBROS	325
Exemplo de acesso a pacote	326
Entendendo os membros protegidos	328
IMPORTANDO PACOTES	330
Importando pacotes Java padrão	331
IMPORTAÇÃO ESTÁTICA	335
EXERCÍCIOS	338
Capítulo 10 Tratamento de exceções	341
HIERARQUIA DE EXCEÇÕES	342
FUNDAMENTOS DO TRATAMENTO DE EXCEÇÕES	342
Usando try e catch	343
Exemplo de exceção simples	343
CONSEQUÊNCIAS DE UMA EXCEÇÃO NÃO CAPTURADA	346
EXCEÇÕES PERMITEM QUE VOCÊ TRATE ERROS NORMALMENTE	347
USANDO VÁRIAS CLÁUSULAS CATCH	349
CAPTURANDO EXCEÇÕES DE SUBCLASSES	350
BLOCOS try PODEM SER ANINHADOS	351
LANÇANDO UMA EXCEÇÃO	353
Relançando uma exceção	354
EXAME MAIS DETALHADO DE Throwable	355
USANDO finally	357
USANDO throws	359
EXCEÇÕES INTERNAS DA LINGUAGEM JAVA	360
NOVOS RECURSOS DE EXCEÇÕES ADICIONADOS PELO JDK7	363
criando subclasses de exceções	364
EXERCÍCIOS	371
Capítulo 11 Usando I/O	376
I/O JAVA É BASEADO EM FLUXOS	377
FLUXOS DE BYTES E FLUXOS DE CARACTERES	377
CLASSE DE FLUXOS DE BYTES	377
CLASSE DE FLUXOS DE CARACTERES	378
FLUXOS PREDEFINIDOS	379
USANDO OS FLUXOS DE BYTES	380

Lendo a entrada do console	381
Gravando a saída no console	382
LENDENDO E GRAVANDO ARQUIVOS USANDO FLUXOS DE BYTES	383
Obtendo entradas de um arquivo	384
Gravando em um arquivo	387
FECHANDO AUTOMATICAMENTE UM ARQUIVO	389
LENDO E GRAVANDO DADOS BINÁRIOS	392
ARQUIVOS DE ACESSO ALEATÓRIO	397
USANDO OS FLUXOS BASEADOS EM CARACTERES DA LINGUAGEM JAVA	400
Entrada do console com o uso de fluxos de caracteres	400
Saída no console com o uso de fluxos de caracteres	404
I/O DE ARQUIVO COM O USO DE FLUXOS DE CARACTERES	405
Usando um FileWriter	405
Usando um FileReader	406
File	408
Obtendo as propriedades de um arquivo	408
Obtendo uma listagem de diretório	410
Usando FilenameFilter	411
A alternativa listFiles()	412
Vários métodos utilitários de File	412
USANDO OS ENCAPSULADORES DE TIPOS DA LINGUAGEM JAVA	
PARA CONVERTER STRINGS NUMÉRICOS	414
EXERCÍCIOS	423
Capítulo 12 Programação com várias threads	428
FUNDAMENTOS DO USO DE VÁRIAS THREADS	428
A CLASSE Thread e a INTERFACE Runnable	429
CRIANDO UMA THREAD	430
Algumas melhorias simples	433
CRIANDO VÁRIAS THREADS	438
DETERMINANDO QUANDO UMA THREAD TERMINA	440
PRIORIDADES DAS THREADS	443
SÍNCRONIZAÇÃO	444
USANDO MÉTODOS SÍNCRONIZADOS	445
A INSTRUÇÃO synchronized	448
COMUNICAÇÃO ENTRE THREADS COM O USO DE notify(), wait() E notifyAll()	451
Exemplo que usa wait() e notify()	451
SUSPENDENDO, RETOMANDO E ENCERRANDO THREADS	457
EXERCÍCIOS	462

Capítulo 13 Enumerações, autoboxing e anotações	467
ENUMERAÇÕES	467
Fundamentos da enumeração	468
AS ENUMERAÇÕES JAVA SÃO TIPOS DE CLASSE	471
MÉTODOS values() E valueOf()	471
CONSTRUTORES, MÉTODOS, VARIÁVEIS DE INSTÂNCIA E ENUMERAÇÕES	472
Duas restrições importantes	474
ENUMERAÇÕES HERDAM Enum	474
AUTOBOXING	482
Encapsuladores de tipos	482
Fundamentos do autoboxing	484
Autoboxing e os métodos	485
Autoboxing/unboxing ocorre em expressões	486
Advertência	488
ANOTAÇÕES (METADADOS)	489
Criando e usando uma anotação	489
Anotações internas	490
EXERCÍCIOS	492
Capítulo 14 Tipos genéricos	496
FUNDAMENTOS DOS TIPOS GENÉRICOS	497
Exemplo simples de genérico	497
Genéricos só funcionam com objetos	501
Tipos genéricos diferem de acordo com seus argumentos de tipo	501
Classe genérica com dois parâmetros de tipo	501
A forma geral de uma classe genérica	503
TIPOS LIMITADOS	504
USANDO ARGUMENTOS CURINGAS	507
CURINGAS LIMITADOS	510
MÉTODOS GENÉRICOS	513
CONSTRUTORES GENÉRICOS	515
HIERARQUIAS DE CLASSES GENÉRICAS	516
INTERFACES GENÉRICAS	519
TIPOS BRUTOS E CÓDIGO LEGADO	526
INFERÊNCIA DE TIPOS COM O OPERADOR LOSANGO	529
ERASURE	531
ERROS DE AMBIGUIDADE	531
ALGUMAS RESTRIÇÕES DOS GENÉRICOS	532
Parâmetros de tipos não podem ser instanciados	532
Restrições aos membros estáticos	533
Restrições aos arrays genéricos	533

Restrições a exceções genéricas	534
EXERCÍCIOS	534
Capítulo 15 Applets e as outras palavras-chave Java	539
ASPECTOS BÁSICOS DOS APPLETS	539
ESQUELETO DE APPLET COMPLETO	543
INICIALIZAÇÃO E ENCERRAMENTO DO APPLET	544
ASPECTO-CHAVE DA ARQUITETURA DE UM APPLET	544
SOLICITANDO ATUALIZAÇÃO	545
USANDO A JANELA DE STATUS	550
PASSANDO PARÂMETROS PARA APPLETS	551
AS OUTRAS PALAVRAS-CHAVE JAVA	553
Modificador volatile	554
Modificador transient	554
instanceof	554
strictfp	554
assert	555
Métodos nativos	555
EXERCÍCIOS	557
Capítulo 16 Introdução ao projeto orientado a objetos	559
UM SOFTWARE ELEGANTE E POR QUE ISSO IMPORTA	560
Propriedades de um software elegante	561
MÉTODOS ELEGANTES	563
Convenções de nomenclatura	563
Coesão dos métodos	564
Objetos bem-formados	566
Documentação interna	567
Documentação externa	568
CLASSE ELEGANTES	571
A coesão das classes e o padrão Expert	571
Evitando duplicação	573
Interface completa	575
Projete pensando em mudanças	576
Lei de Demeter	579
HERANÇA VERSUS DELEGAÇÃO	581
Diagramas de classes UML	581
Possibilidade de reutilização do código	584
O relacionamento É-um	585
Comportamento semelhante	587
Polimorfismo	590

Custos da herança	590
PADRÕES DE PROJETO	593
Padrão Adapter	594
Padrão Observer	597
EXERCÍCIOS	602
 PARTE II INTRODUÇÃO À PROGRAMAÇÃO DE GUIs COM SWING	 607
Capítulo 17 Aspectos básicos de Swing	609
ORIGENS E FILOSOFIA DE PROJETO DE SWING	610
COMPONENTES E CONTÊINERES	612
Componentes	612
Contêineres	613
Painéis do contêiner de nível superior	613
GERENCIADORES DE LEIAUTE	614
PRIMEIRO PROGRAMA SWING SIMPLES	615
Primeiro exemplo de Swing linha a linha	617
TRATAMENTO DE EVENTOS	621
Eventos	621
Fontes de eventos	621
Ouvintes de eventos	622
Classes de eventos e interfaces de ouvintes	622
Classes adaptadoras	624
USANDO UM BOTÃO DE AÇÃO	625
INTRODUÇÃO AO JTextField	633
USE CLASSE INTERNAS ANÔNIMAS PARA TRATAR EVENTOS	645
EXERCÍCIOS	646
 Capítulo 18 Examinando os controles de Swing	 649
JLabel e ImageIcon	650
OS BOTÕES DE SWING	653
Tratando eventos de ação	654
Tratando eventos de item	654
JButton	655
JToggleButton	658
Caixas de seleção	660
Botões de rádio	662
JTextField	665
JScrollPane	675
JList	682
JComboBox	686

ÁRVORES	689
JTable	693
UMA EXPLICAÇÃO RÁPIDA DOS MODELOS	696
EXERCÍCIOS	697
Capítulo 19 Trabalhando com menus	700
ASPECTOS BÁSICOS DOS MENUS	700
UMA VISÃO GERAL DE JMenuBar, JMenu e JMenuItem	702
JMenuBar	702
JMenu	703
JMenuItem	704
CRIE UM MENU PRINCIPAL	704
ADICIONE MNEMÔNICOS E ACELERADORES AOS ITENS DE MENU	709
ADICIONE IMAGENS E DICAS DE FERRAMENTAS AOS ITENS DE MENU	712
USE JRadioButtonMenuItem e JCheckBoxMenuItem	720
EXERCÍCIOS	722
Capítulo 20 Caixas de diálogo	725
JOptionPane	726
showMessageDialog()	728
showConfirmDialog()	732
showInputDialog()	736
showOptionDialog()	741
JDialog	746
CRIE UMA CAIXA DE DIÁLOGO NÃO MODAL	750
SELEÇÃO ARQUIVOS COM JFileChooser	751
EXERCÍCIOS	762
Capítulo 21 Threads, applets e geração de componentes	766
O USO DE VÁRIAS THREADS EM SWING	766
USE Timer	773
CRIE APPLETS SWING	779
Um applet Swing simples	780
GERANDO COMPONENTES	787
Fundamentos da geração de componentes	787
O contexto gráfico	788
Calcule a área de desenho	789
Solicite a geração do componente	789
Um exemplo de geração de componente	789
EXERCÍCIOS	795

PARTE III EXAMINANDO A BIBLIOTECA DE APIs JAVA	797
Capítulo 22 Manipulação de strings	799
ASPECTOS BÁSICOS DOS STRINGS	799
OS CONSTRUTORES DE STRING	800
TRÊS RECURSOS DA LINGUAGEM RELACIONADOS A STRINGS	802
Literais de strings	803
Concatenação de strings	803
Concatenação de strings com outros tipos de dados	803
Sobrepondo <code>toString()</code>	804
O MÉTODO <code>length()</code>	809
OBTENDO OS CARACTERES DE UM STRING	809
<code>charAt()</code>	810
<code>getChars()</code>	810
<code>toCharArray()</code>	811
COMPARAÇÃO DE STRINGS	812
<code>equals()</code> e <code>equalsIgnoreCase()</code>	812
<code>equals()</code> versus <code>==</code>	813
<code>regionMatches()</code>	814
<code>startsWith()</code> e <code>endsWith()</code>	814
<code>compareTo()</code> e <code>compareToIgnoreCase()</code>	815
USANDO <code>indexOf()</code> E <code>lastIndexOf()</code>	817
OBTENDO UM STRING MODIFICADO	819
<code>substring()</code>	819
<code>replace()</code>	820
<code>trim()</code>	821
ALTERANDO A CAIXA DOS CARACTERES DE UM STRING	822
StringBuffer E StringBuilder	825
EXERCÍCIOS	825
Capítulo 23 Examinando o pacote java.lang	828
ENCAPSULADORES DE TIPOS PRIMITIVOS	829
Number	829
Double e Float	830
Byte, Short, Integer e Long	831
Character	834
Boolean	837
O autoboxing e os encapsuladores de tipos	837
A CLASSE Math	838
A CLASSE Process	842
A CLASSE ProcessBuilder	842
A CLASSE Runtime	844
A CLASSE System	846

Usando currentTimeMillis() para marcar o tempo de execução do programa	847
Usando arraycopy()	848
Obtendo valores de propriedades	849
Redirecionando fluxos de I/O padrão	849
A CLASSE Object	851
A CLASSE Class	851
A CLASSE Enum	853
CLASSES RELACIONADAS A THREADS E A INTERFACE Runnable	853
OUTRAS CLASSES	854
AS INTERFACES DE java.lang	854
A interface Comparable	855
A interface Appendable	857
A interface Iterable	857
A interface Readable	858
A interface CharSequence	858
A interface AutoCloseable	859
EXERCÍCIOS	860
Capítulo 24 Examinando o pacote java.util	862
A CLASSE Locale	864
TRABALHANDO COM DATA E HORA	867
Date	867
Calendar e GregorianCalendar	868
FORMATANDO A SAÍDA COM Formatter	873
Os construtores de Formatter	873
Aspectos básicos da formatação	874
Formatando strings e caracteres	877
Formatando números	877
Formatando data e hora	878
Os especificadores %n e %%	880
Especificando uma largura de campo mínima	880
Especificando precisão	881
Usando os flags de formatação	882
A opção de uso de maiúsculas	884
Usando um índice de argumento	885
Formatação para um local diferente	886
Fechando um Formatter	887
A FORMATAÇÃO E O MÉTODO printf()	890
A CLASSE Scanner	892
Os construtores de Scanner	892
Aspectos básicos da varredura	892
Alguns exemplos com Scanner	894

Mais alguns recursos de Scanner	899
A CLASSE Random	900
USE Observable e Observer	901
AS CLASSES Timer e TimerTask	905
CLASSES E INTERFACES UTILITÁRIAS VARIADAS	907
EXERCÍCIOS	908
Capítulo 25 Usando as estruturas de dados do Collections Framework	911
VISÃO GERAL DAS ESTRUTURAS DE DADOS	912
Pilhas e filas	912
Listas encadeadas	913
Árvores	914
Tabelas hash	915
Selecionando uma estrutura de dados	915
VISÃO GERAL DAS COLEÇÕES	916
AS INTERFACES DE COLEÇÕES	917
A interface Collection	918
A interface List	920
A interface Set	920
A interface SortedSet	920
A interface NavigableSet	922
A interface Queue	922
A interface Deque	924
AS CLASSES DE COLEÇÕES	926
A classe ArrayList	927
A classe LinkedList	930
A classe HashSet	934
A classe TreeSet	936
A classe LinkedHashSet	938
A classe ArrayDeque	939
A classe PriorityQueue	941
ACESSANDO UMA COLEÇÃO COM UM ITERADOR	942
Usando um iterador	942
A alternativa aos iteradores com o uso de for-each	946
TRABALHANDO COM MAPAS	946
As interfaces de mapas	946
A interface Map	947
A interface SortedMap	947
A interface NavigableMap	948
A interface Map.Entry	949
As classes de mapas	951

<i>A classe HashMap</i>	952
<i>A classe TreeMap</i>	953
<i>A classe LinkedHashMap</i>	955
COMPARADORES	956
OS ALGORITMOS DE COLEÇÕES	959
A CLASSE Arrays	962
AS CLASSES E INTERFACES LEGADAS	963
A interface Enumeration	963
Vector	963
Stack	963
Dictionary	963
Hashtable	963
Properties	964
EXERCÍCIOS	964
Capítulo 26 Redes com java.net	967
ASPECTOS BÁSICOS DE REDES	967
AS CLASSES E INTERFACES DE REDES	968
A CLASSE InetAddress	969
A CLASSE Socket	971
A CLASSE URL	975
A CLASSE URLConnection	977
A CLASSE HttpURLConnection	982
DATAGRAMAS	984
DatagramSocket	985
DatagramPacket	985
Um exemplo de datagrama	986
EXERCÍCIOS	989
Capítulo 27 Os utilitários de concorrência	991
OS PACOTES DA API DE CONCORRÊNCIA	992
java.util.concurrent	992
java.util.concurrent.atomic	993
java.util.concurrent.locks	993
USANDO OBJETOS DE SINCRONIZAÇÃO	994
Semaphore	994
CountDownLatch	997
CyclicBarrier	1000
Exchanger	1002
Phaser	1005
USANDO UM EXECUTOR	1012
Um exemplo de executor simples	1013

USANDO Callable E Future	1015
A ENUMERAÇÃO TimeUnit	1018
AS COLEÇÕES DE CONCORRÊNCIA	1020
BLOQUEIOS	1020
OPERAÇÕES ATÔMICAS	1023
PROGRAMAÇÃO PARALELA COM O FRAMEWORK FORK/JOIN	1024
AS PRINCIPAIS CLASSES DO FRAMEWORK FORK/JOIN	1025
ForkJoinTask<V>	1025
RecursiveAction	1026
RecursiveTask<V>	1026
ForkJoinPool	1027
A ESTRATÉGIA DE DIVIDIR E CONQUISTAR	1028
Um primeiro exemplo simples do Framework Fork/Join	1029
Entendendo o impacto do nível de paralelismo	1031
Um exemplo que usa RecursiveTask<V>	1034
Executando uma tarefa de forma assíncrona	1036
OS UTILITÁRIOS DE CONCORRÊNCIA VERSUS A ABORDAGEM TRADICIONAL JAVA	1037
EXERCÍCIOS	1038
Apêndice A Usando comentários de documentação da linguagem Java	1041
TAGS DE javadoc	1041
FORMA GERAL DE UM COMENTÁRIO DE DOCUMENTAÇÃO	1045
O QUE javadoc GERA	1046
EXEMPLO QUE USA COMENTÁRIOS DE DOCUMENTAÇÃO	1046
Apêndice B Introdução às expressões regulares	1049
A CLASSE Pattern	1049
A CLASSE Matcher	1050
ASPECTOS BÁSICOS DA SINTAXE DAS EXPRESSÕES REGULARES	1050
DEMONSTRANDO A CORRESPONDÊNCIA DE PADRÕES	1051
USANDO O CARACTERE CURINGA E QUANTIFICADORES	1053
TRABALHANDO COM CLASSES DE CARACTERES	1055
USANDO replaceAll()	1055
A CONEXÃO COM A CLASSE String	1056
ASSUNTOS A EXPLORAR	1056
Apêndice C Respostas de exercícios selecionados	1057
Índice	1111

PARTE I

A linguagem Java

A Parte I deste livro descreve os elementos que compõem a linguagem de programação Java e as técnicas que seu uso requer. O Capítulo 1 começa apresentando vários conceitos básicos de programação, a história e a filosofia de projeto de Java e uma visão geral de alguns recursos importantes da linguagem. Os demais capítulos enfocam aspectos específicos de Java, capítulo a capítulo. A Parte I termina com a introdução a um aspecto importante de uma programação bem-sucedida em Java: o projeto orientado a objetos.

Fundamentos da programação Java

PRINCIPAIS HABILIDADES E CONCEITOS

- Conhecer os componentes básicos do computador
- Entender os bits, os bytes e o sistema de numeração binário
- Conhecer as duas formas de um programa
- Saber a história e a filosofia de Java
- Entender os princípios básicos da programação orientada a objetos
- Criar, compilar e executar um programa Java simples
- Usar variáveis
- Usar as instruções de controle **if** e **for**
- Criar blocos de código
- Entender como as instruções são posicionadas, recuadas e finalizadas
- Saber as palavras-chave Java
- Entender as regras dos identificadores Java

No intervalo de apenas algumas décadas, a programação deixou de ser uma disciplina obscura, praticada por poucos, para se tornar parte integrante do mundo moderno, praticada por muitos. A razão desse desenvolvimento é fácil de entender. Se o mundo moderno pudesse ser caracterizado por uma palavra, ela seria *tecnologia*. O que dá apoio a grande parte dessa tecnologia é o computador, e o que torna um computador útil são os programas que ele executa. Logo, em muitos aspectos, é a programação que torna possível o nosso mundo tecnológico. Ela é importante assim.

A finalidade deste livro é ensinar os fundamentos da programação usando a linguagem Java. Como disciplina, a programação é bem extensa. Além de envolver muitas habilidades, conceitos e técnicas, há várias especializações, como as que envolvem análise numérica, teoria da informação, rede e controle de dispositivos. Também há muitos ambientes de computação diferentes em que os programas são executados. No entanto, seja qual for o caso, dominar os fundamentos da programação é necessário. O que você aprenderá neste curso formará a base de seus estudos.

Este capítulo começa definindo vários termos-chave, examinando o conceito dos bits, dos bytes e do sistema de numeração binário e os componentes básicos do computador. Embora isso seja território familiar para muitos leitores, é um modo de assegurar que todos começem com o conhecimento necessário. Em seguida, introduzimos a linguagem Java apresentando sua história, filosofia de design e vários de seus atributos mais importantes.

O capítulo discute então vários recursos básicos de Java. Uma das coisas mais difíceis quando aprendemos a programar é o fato de nenhum elemento de uma linguagem de computador existir isoladamente. Em vez disso, os componentes da linguagem estão relacionados e trabalham em conjunto. Nesse ponto, Java não é exceção. É difícil discutir um aspecto de Java sem envolver outros aspectos implicitamente. Para ajudar a resolver esse problema, este capítulo fornece uma breve visão geral de vários recursos Java, inclusive a forma geral de um programa Java, algumas instruções de controle básicas, uma amostra dos tipos de dados e os operadores. Ele não entra em muitos detalhes, concentrando-se nos conceitos gerais comuns a qualquer programa Java. Muitos desses recursos serão examinados com mais detalhes posteriormente no livro, mas essa introdução o ajudará a ver como partes essenciais de Java “se encaixam”, e também permitirá que você comece a criar e executar programas Java.

ASPECTOS BÁSICOS DA COMPUTAÇÃO

Se você está estudando programação, é muito provável que já tenha pelo menos um conhecimento geral sobre computação. No entanto, as pessoas não têm necessariamente o mesmo conhecimento, ou esse conhecimento pode ser impreciso. Por isso, antes de introduzirmos a linguagem Java, uma visão geral de diversos conceitos básicos da computação será apresentada. No processo, vários termos-chave serão definidos.

Os componentes de hardware de um computador

Como o computador é que acabará executando os programas que você criar, é útil entender de uma maneira geral o que as partes de um computador fazem. Todos os computadores são compostos por um grupo de componentes que funcionam em conjunto para formar o computador em sua totalidade. Embora seja verdade que a forma exata do computador tenha evoluído com o tempo, todos os computadores ainda compartilham certos recursos-chave. Por exemplo, os mesmos elementos básicos contidos em um computador de mesa também são encontrados em um smartphone.

Para ser útil, um computador deve conter, no mínimo, o seguinte:

- Uma Unidade Central de Processamento (CPU, Central Processing Unit)
- Memória
- Dispositivos de entrada/saída

Examinemos cada um desses itens, um por vez.

A CPU fornece os recursos computacionais primários do computador. Ela faz isso executando as instruções que compõem um programa. Todas as CPUs são projetadas para entender um *conjunto de instruções* específico. O conjunto de instruções define os diversos tipos de operações que a CPU pode executar. Por exemplo, a maioria das CPUs dá suporte a instruções que executam operações aritméticas básicas, carregam dados da e armazenam dados na memória, fazem comparações lógicas e

alteram o fluxo do programa, para citar apenas algumas. Além de poder acessar a memória, grande parte das CPUs contém um número limitado de *registradores* que fornecem armazenamento de dados rápido e de curto prazo.

As instruções que uma CPU processa, que costumam ser chamadas de *instruções de máquina*, ou *código de máquina*, executam operações muito pequenas. Por exemplo, uma instrução pode mover um valor de um registrador para outro, mover um valor de um registrador para a memória ou comparar o conteúdo de dois registradores. Em geral, o conjunto de instruções de um tipo de CPU difere do de outro tipo. Por isso, normalmente um conjunto de instruções projetado para um tipo de CPU não pode ser usado em CPUs de outro tipo. Há famílias de CPUs com compatibilidade regressiva, mas geralmente CPUs não relacionadas diferem em seus conjuntos de instruções.

As instruções de máquina não estão em uma forma que possa ser facilmente lida por uma pessoa. Elas são codificadas para uso do computador. É possível, no entanto, representar código de máquina em uma forma legível para humanos usando representações mnemônicas das instruções. Isso se chama *linguagem simbólica* (também conhecida como “linguagem de montagem” ou “linguagem assembly”). Por exemplo, a representação mnemônica de uma instrução que move dados de um local para outro poderia se chamar MOV. A instrução de comparação de dois valores poderia se chamar CMP. Uma linguagem simbólica é convertida por um programa chamado *montador* em uma forma que o computador pode executar. No entanto, poucas pessoas escrevem em linguagem simbólica hoje porque, geralmente, linguagens como Java fornecem uma alternativa bem melhor.

A memória do computador é usada para armazenar instruções (na forma de código de máquina) e dados. Seu objetivo principal é manter informações somente durante a execução de um programa. Não é para armazenamento de longo prazo. A memória é *endereçável*, ou seja, a CPU pode acessar um local específico na memória, dado seu endereço. Geralmente ela é chamada de RAM, que significa Random Access Memory.

Quando a CPU executa um programa, ela faz isso acessando uma instrução na memória e então executando a operação especificada por essa instrução. Em seguida, ela obtém a próxima instrução e a executa, e assim por diante. Por padrão, as instruções são obtidas em locais sequenciais da memória. No entanto, algumas instruções podem alterar esse fluxo, fazendo a execução “saltar” para um local diferente na memória.

Atualmente, há uma ampla variedade de dispositivos de entrada/saída (I/O, input/output), como teclados, monitores, o mouse, telas sensíveis ao toque, entrada de voz e saída de som. Todos têm a mesma função: dar ao computador uma maneira de receber ou transmitir informações. Com frequência, os dispositivos de I/O (input/output, ou entrada/saída) permitem que os humanos interajam com o computador. Contudo, em alguns casos, o computador usa o I/O para se comunicar com outro dispositivo, como um dispositivo de armazenamento, um adaptador de rede ou até mesmo uma interface de controle robótico.

Além dos três componentes básicos do computador que acabamos de descrever, muitos computadores também incluem dispositivos de armazenamento, como unidades de disco, DVDs e unidades flash. E muitos computadores estão em rede, via Internet ou uma rede local. Para dar suporte à rede, o computador precisa de um adaptador de rede.

Bits, bytes e binário

Nos dias de hoje, é raro encontrar alguém que não tenha ouvido falar nos termos *bits*, *bytes* e *binário*. Eles fazem parte do vocabulário cotidiano. No entanto, já que descrevem alguns dos aspectos mais básicos da computação, é importante que sejam formalmente definidos.

O sistema de numeração binário

No nível mais baixo, os computadores trabalham com 1s e 0s. Como resultado, um sistema de numeração baseado em 1s e 0s é necessário. Esse sistema de numeração se chama *binário*. O sistema binário funciona da mesma forma que nosso sistema de numeração decimal comum, exceto pelo significado da posição de cada dígito ser diferente. Como você sabe, no sistema decimal, conforme nos movemos da direita para a esquerda, a posição de cada dígito representa valores que são 10 vezes maiores do que o dígito anterior. Logo, o sistema decimal é baseado em potências de 10, com o dígito da extrema direita sendo a posição unitária, à sua esquerda ficando a posição decimal, depois a posição da centena e assim por diante. Por exemplo, o número 423 significa quatrocentos e vinte e três, porque há quatro centenas, 2 dezenas e 3 unidades.

No sistema binário, o processo funciona da mesma maneira, exceto pelo fato de, ao nos movermos para a esquerda, a posição de cada dígito aumentar segundo um fator igual a 2. Portanto, a posição do primeiro dígito binário (o da extrema direita) representa 1. À sua esquerda é a posição do 2 e depois a posição do 4, seguida pela posição do 8, etc. Ou seja, as oito primeiras posições dos dígitos binários representam os valores a seguir:

128 64 32 16 8 4 2 1

Por exemplo, o valor binário 1010 é o valor decimal 10. Por quê? Porque não tem 1, tem um 2, não tem 4 e tem um 8. Logo, $0 + 2 + 0 + 8$ é igual a 10. Outro exemplo: o valor binário 1101 é 13 em decimal porque tem 1, não tem 2, tem um 4 e um 8. Logo, $1 + 0 + 4 + 8$ é igual a 13. Como podemos ver, para fazer a conversão de binário para decimal, só temos que somar os valores representados pelos dígitos 1.

Bits e bytes

No computador, um dígito binário é representado individualmente por um *bit*. Um bit pode estar ativado ou desativado. Um bit ativado é igual a 1 e um bit desativado é igual a 0. Os bits ficam organizados em grupos. O mais comum é o *byte*. Normalmente um byte é composto por 8 bits. Ou seja, ele pode representar os valores de 0 a 255.

Outra unidade organizacional é a *palavra*. Normalmente, a palavra é dimensionada para ser compatível com uma arquitetura de CPU específica. Por exemplo, um computador de 32 bits costuma usar uma palavra de 32 bits (4 bytes).

Por conveniência, muitas vezes os números binários são mostrados agrupados em unidades de 4 (ou às vezes de 8) dígitos – por exemplo, 1011 1001. Isso facilita a visualização dos dígitos. No entanto, temos que entender que esses agrupamentos visuais não têm relação com o valor que está sendo representado.

O sistema operacional

Os componentes de hardware do computador são gerenciados e disponibilizados pelo *sistema operacional*. Um sistema operacional é um programa mestre que controla o

computador. Os sistemas operacionais são um dos principais tópicos da ciência da computação e não é possível descrevê-los em detalhes aqui. Felizmente, uma visão geral breve é suficiente para o que pretendemos.

Um sistema operacional serve a duas funções básicas. Em primeiro lugar, fornece um nível básico de funcionalidade que outros programas usarão para acessar os recursos do computador. Por exemplo, para salvar informações em uma unidade de disco, você usará um serviço fornecido pelo sistema operacional. Em segundo lugar, o sistema operacional controla a execução de outros programas. Por exemplo, ele fornece espaço na memória para o armazenamento do programa enquanto este estiver sendo executado, agenda tempo da CPU para sua execução e supervisiona o seu uso dos recursos.

Vários sistemas operacionais são comuns, como Windows, Unix, Linux, Mac OS, iOS e Android. Como regra geral, um programa deve ser projetado para execução em (tendo como destino) um sistema operacional específico. Por exemplo, um programa destinado ao Windows não pode ser executado no Unix, a menos que seja especificamente adaptado.

Verificação do progresso

1. A CPU executa instruções de _____.
2. Como é 27 em binário?
3. Que programa supervisiona a operação do computador?

O PROGRAMA

A base da programação é o *programa*. Já que este livro é sobre programação, faz sentido definirmos formalmente esse termo. Aqui está uma definição bem genérica: um programa é composto por uma sequência de instruções que pode ser executada por um computador. No entanto, o termo *programa* pode significar coisas diferentes, dependendo de seu contexto, porque um programa pode ter duas formas básicas. Uma é legível para humanos, e a outra, para máquinas.

Quando você escrever um programa, estará criando um arquivo de texto contendo seu *código-fonte*. Essa é a forma do programa legível para humanos. É a forma que normalmente os programadores consideram ser “o programa”. No entanto, não é a forma realmente executada pelo computador. Em vez disso, o código-fonte de um programa deve ser convertido em instruções que o computador possa executar. É o chamado *código-objeto*. É difícil (quase impossível) para os humanos lerem um arquivo de código-objeto. É por isso que os programadores trabalham com o código-fonte de seus programas, convertendo-o em código-objeto apenas quando chega a hora de executá-lo.

Um programa é convertido de código-fonte para código-objeto por um *compilador*. Em alguns casos, o compilador gera instruções de máquina reais que são executadas diretamente pela CPU do computador. (Normalmente é assim que funciona

Respostas:

1. máquina
2. 11011
3. O sistema operacional.

um compilador para a programação em linguagens como C++, por exemplo.) Um ponto que devemos entender sobre o código-objeto é que ele é projetado para um tipo específico de CPU. Como explicado anteriormente, as instruções de máquina de um tipo de CPU não costumam funcionar com outro tipo de CPU.

Pode parecer estranho o fato de que, em alguns casos, as instruções do código-objeto produzidas por um compilador não sejam destinadas a uma CPU real! Em vez disso, devem ser executadas por uma *máquina virtual*. Uma máquina virtual é um programa que emula uma CPU em software. Assim, cria o que é, essencialmente, uma CPU em lógica em vez de em hardware. Como tal, ela define seu próprio conjunto de instruções, o qual é capaz de executar. Normalmente o processo de execução dessas instruções é chamado de *interpretação*, e às vezes a máquina virtual é chamada de *interpretador*. Como você verá em breve, Java usa uma máquina virtual e há vantagens significativas nessa abordagem.

Independentemente de seu código-fonte ser compilado para código de máquina diretamente executável ou para código a ser executado por uma máquina virtual, o processo de conversão do código-fonte em código-objeto via compilador é o mesmo.

LINGUAGENS DE PROGRAMAÇÃO

Quem define os elementos específicos do código-fonte de um programa é a linguagem de programação que está sendo usada. Há duas categorias básicas de linguagens: a de baixo nível e a de alto nível. A linguagem de baixo nível tem uma ligação direta com o conjunto de instruções da CPU. A linguagem simbólica é um exemplo de linguagem de baixo nível. Como explicado antes, há uma correspondência de um para um entre cada instrução de código simbólico e uma instrução de máquina. Isso torna a criação de código simbólico uma tarefa tediosa.

Atualmente, grande parte da programação é feita com o uso de uma linguagem de alto nível. (Por exemplo, Java é uma linguagem de alto nível.) As linguagens de alto nível permitem a criação de programas de maneira mais rápida, fácil e confiável. Uma linguagem de alto nível define estruturas que ajudam a organizar, estruturar e controlar a lógica do programa. Cada estrutura da linguagem de alto nível é convertida em muitas instruções de máquina.

Há muitas linguagens de programação de alto nível, mas quase todas definem três elementos básicos:

- palavras-chave
- operadores
- pontuação

Esses elementos devem ser combinados de acordo com as *regras de sintaxe* definidas pela linguagem. As regras de sintaxe especificam com bastante precisão o que constitui o uso válido de um elemento do programa. Para ser compilado, o código-fonte deve aderir a essas regras.

Em uma definição geral, as palavras-chave definem os blocos de construção da linguagem. Elas são usadas para especificar as estruturas de alto nível suportadas pela linguagem. Por exemplo, as palavras-chave são usadas para controlar o fluxo de execução, definir vários tipos de dados e fornecer opções e mecanismos que permitam o gerenciamento da execução de um programa.

Os operadores são usados por expressões e uma das mais comuns é a expressão aritmética. Por exemplo, quase todas as linguagens usam + para especificar adição. A pontuação abrange os elementos da linguagem que são usados para separar um elemento de outro, agrupar instruções, evitar ambiguidade ou até mesmo tornar mais clara a sintaxe da linguagem.

Embora muitas linguagens de programação tenham sido inventadas, só algumas passaram a ser amplamente usadas. Entre elas estão FORTRAN, COBOL, Pascal, vários dialetos do BASIC, C, C++ e, é claro, Java. Felizmente, depois que você aprender uma linguagem de programação, será muito mais fácil aprender outra. Portanto, o tempo que investir no aprendizado de Java o beneficiará não só agora como no futuro.

Verificação do progresso

1. A forma de um programa legível para humanos se chama _____.
2. A forma executável de um programa se chama _____.
3. O que são regras de sintaxe?

Pergunte ao especialista

P Ouvi programadores usarem a expressão “escrever código”. O que significa?

R Com frequência, programadores profissionais chamam o ato de programar (isto é, criar código-fonte) de “escrever código”. Outra expressão que você deve ouvir é “codificar um programa”, que também se refere à criação de código-fonte. Na verdade, é comum ouvirmos um excelente programador ser chamado de um “ótimo codificador”.

A LINGUAGEM JAVA

Este livro usa a linguagem Java para ensinar os fundamentos da programação. Embora outras linguagens de programação também pudesse ser usadas para esse fim, Java foi selecionada principalmente por duas razões. Em primeiro lugar, é uma das linguagens de computador mais usadas no mundo. Portanto, de um ponto de vista prático, é uma ótima linguagem para se aprender. Em segundo lugar, seus recursos são projetados e implementados de tal maneira que é fácil demonstrar as bases da programação.

Mas também há uma terceira razão. Java representa muito do que caracteriza a programação moderna. Conhecer Java dá uma ideia do que os programadores profissionais pensam sobre a tarefa de programar. É uma das linguagens que definem nossa época.

Java faz parte do progressivo processo histórico de evolução das linguagens de computador. Como tal, é uma mistura dos melhores elementos de sua rica herança combinados com os conceitos inovadores inspirados por seu lugar exclusivo na his-

Respostas:

1. código-fonte
2. código-objeto
3. As regras de sintaxe determinam como os elementos de uma linguagem são usados.

tória da programação. Enquanto o resto deste livro descreve os aspectos práticos da linguagem Java, aqui examinaremos as razões de sua criação, as forças que a moldaram e o legado que ela herdou.

Origem da linguagem Java

Java foi concebida por James Gosling e outras pessoas da Sun Microsystems em 1991. Inicialmente, a linguagem se chamava “Oak”, mas foi renomeada como “Java” em 1995. Ainda que Java tenha se tornado inexoravelmente vinculada ao ambiente online, a Internet não foi o ímpeto original! Em vez disso, a principal motivação foi a necessidade de uma linguagem independente de plataforma que pudesse ser usada na criação de softwares para serem embutidos em vários dispositivos eletrônicos dos consumidores, como fornos de micro-ondas e controles remotos. Como era de se esperar, muitos tipos de CPUs diferentes são usados como controladores. O problema era que, na época, a maioria das linguagens de computador era projetada para ser compilada para código de máquina de um tipo específico de CPU. Por exemplo, considere C++, outra linguagem que também foi muito popular na época (e ainda é).

Embora fosse possível compilar um programa C++ para quase todo tipo de CPU, era preciso um compilador C++ completo destinado a essa CPU. Isso ocorre porque normalmente o C++ é compilado para instruções de máquina que são executadas diretamente pela CPU e cada CPU requeria um conjunto de instruções de máquina diferente. O problema, no entanto, é que criar compiladores é caro e demorado. Em uma tentativa de encontrar uma solução melhor, Gosling e outros trabalharam em uma linguagem com portabilidade entre plataformas que pudesse produzir código para ser executado em várias CPUs com ambientes diferentes. Esse esforço acabou levando à criação de Java.

Mais ou menos na época em que os detalhes de Java estavam sendo esboçados, surgiu um segundo fator muito importante que desempenharia papel crucial no futuro da linguagem. É claro que essa segunda força foi a World Wide Web. Se a Web não estivesse se formando quase ao mesmo tempo em que Java estava sendo implementada, talvez ela continuasse sendo uma linguagem útil, mas obscura, para a programação de utensílios eletrônicos. No entanto, com o surgimento da Web, Java foi impulsionada para a dianteira do design das linguagens de computador, porque a Web também precisava de programas portáveis. Por quê? Porque a Internet é frequentada por vários tipos de computadores, usando diferentes tipos de CPUs e sistemas operacionais. Algum meio de permitir que esse variado grupo de computadores executasse o mesmo programa era altamente desejado.

Perto de 1993, ficou óbvio para os membros da equipe de projeto de Java que, com frequência, os problemas de portabilidade encontrados na criação de código para controladores embutidos também são encontrados quando tentamos criar código para a Internet. Essa percepção fez com que o foco de Java mudasse dos utensílios eletrônicos domésticos para a programação na Internet. Assim, embora a fagulha inicial tenha sido gerada pelo desejo por uma linguagem de programação independente da arquitetura, foi a Internet que acabou levando ao sucesso em larga escala de Java. É útil mencionar que Java está diretamente relacionada a duas linguagens mais antigas: C e C++. Ela herda sua sintaxe da linguagem C, e seu modelo de objetos é adaptado de C++. O relacionamento de Java com C e C++ é importante: na época em que Java foi criada, muitos programadores conheciam a sintaxe C/C++,

o que facilitou para um programador C/C++ aprender Java e, da mesma forma, um programador Java aprender C/C++. Além disso, os projetistas não “reinventaram a roda”. Eles conseguiram adaptar, refinar e enriquecer um paradigma de programação já altamente bem-sucedido.

Devido às semelhanças entre Java e C++, principalmente seu suporte à programação orientada a objetos, é tentador pensar em Java simplesmente como a “versão de C++ para a Internet”. No entanto, isso seria um erro. Java tem algumas diferenças significativas. Embora tenha sido influenciada por C++, não é uma versão melhorada dessa linguagem. Por exemplo, ela não é compatível com versões anteriores ou futuras de C++. Além do mais, Java não foi projetada para substituir C++, mas sim para resolver um determinado conjunto de problemas, e C++ para resolver um conjunto de problemas diferente. Elas ainda coexistirão por muitos anos.

Verificação do progresso

1. Java é útil para a Internet porque pode produzir programas _____.
2. Java é descendente direta de quais linguagens?

Contribuição da linguagem Java para a Internet

A Internet ajudou a impulsionar Java para a dianteira da programação; por sua vez, Java teve um efeito profundo sobre a Internet. Além de simplificar a programação geral na Web, ela inovou com um tipo de programa de rede chamado *applet* que, na época, mudou a maneira de o mundo online pensar em conteúdo. Java também resolveu alguns dos problemas mais complicados associados à Internet: portabilidade e segurança. Examinemos mais detalhadamente cada um deles.

Applets Java

Um applet é um tipo especial de programa Java que é projetado para ser transmitido pela Internet e executado automaticamente por um navegador Web compatível com Java. Além disso, ele é baixado sob demanda. Se o usuário clicar em um link que contém um applet, este será automaticamente baixado e executado no navegador. Os applets são projetados como programas pequenos. Normalmente, são usados para exibir dados fornecidos pelo servidor, tratar entradas do usuário ou fornecer funções simples, como uma calculadora de empréstimos, que é executada localmente em vez de no servidor. Basicamente, os applets permitem que uma funcionalidade seja movida do servidor para o cliente. Sua criação mudou a programação na Internet porque expandiu o universo de objetos que podem se mover livremente no ciberespaço.

Mesmo sendo tão desejáveis, os applets também enfrentaram problemas sérios nas áreas de segurança e portabilidade. É claro que um programa que é baixado e executado automaticamente no computador cliente deve ser impedido de causar danos.

Respostas:

1. portáveis
2. C e C++.

Ele também deve poder ser executado em vários ambientes diferentes e em sistemas operacionais distintos. Como você verá, Java resolveu esses problemas de uma maneira eficaz e elegante. Examinemos os dois problemas com mais detalhes.

Segurança

Como você deve saber, sempre que baixamos um programa “normal”, estamos nos arriscando, porque o código baixado pode conter um vírus, cavalo de Troia ou outro código danoso. A parte mais importante do problema é o fato de que um código malicioso pode causar dano, já que ganhou acesso não autorizado a recursos do sistema. Por exemplo, um vírus pode coletar informações privadas, como números de cartão de crédito, saldos de conta bancária e senhas, pesquisando o conteúdo do sistema local de arquivos do computador. Para Java permitir que o applet fosse baixado e executado com segurança no computador cliente, era necessário impedir que ele iniciasse esse tipo de ataque.

A linguagem conseguiu fornecer essa proteção confinando o applet ao ambiente de execução Java e não permitindo que ele acesse outras partes do computador. (Você verá como isso é feito em breve.) Poder baixar applets com a certeza de que nenhum dano será causado e de que a segurança não será violada é um dos recursos mais importantes de Java.

Portabilidade

A portabilidade é um aspecto importante da Internet, porque há muitos tipos de computadores e sistemas operacionais diferentes conectados a ela. Se fosse para um programa Java ser executado em praticamente qualquer computador conectado à Internet, teria que haver alguma maneira de permitir que esse programa fosse executado em diferentes sistemas. Por exemplo, no caso de um applet, o mesmo applet tem que poder ser baixado e executado pela grande variedade de diferentes CPUs, sistemas operacionais e navegadores. Não é prático haver diferentes versões do applet para computadores distintos. O *mesmo* código deve funcionar em *todos* os computadores. Portanto, algum meio de gerar código executável portável era necessário. Felizmente, o mesmo mecanismo que ajuda a manter a segurança também ajuda a gerar portabilidade.

O segredo da linguagem Java: o bytecode

O segredo que permite que Java resolva os problemas de segurança e portabilidade que acabamos de descrever é a saída do compilador Java não ser código de máquina diretamente executável. Em vez disso, é bytecode. O *bytecode* é um conjunto de instruções altamente otimizado projetado para ser executado pela *Máquina Virtual Java* (JVM, Java Virtual Machine). Na verdade, a JVM original foi projetada como um *interpretador de bytecode*. O fato de o programa Java ser executado pela JVM ajuda a resolver os principais problemas de portabilidade e segurança associados a programas baseados na Web. Vejamos por quê.

Converter um programa Java em bytecode facilita muito a execução de um programa em uma grande variedade de ambientes, porque só a JVM tem que ser implementada para cada plataforma. Uma vez que a JVM estiver presente em um determinado sistema, qualquer programa Java poderá ser executado nele. Embora os detalhes da JVM sejam diferentes de uma plataforma para outra, todas interpretam o mesmo

bytecode Java. Se um programa Java fosse compilado para código nativo, deveriam existir diferentes versões do mesmo programa para cada tipo de CPU conectada à Internet. É claro que essa não é uma solução viável. Logo, a execução de bytecode pela JVM é a maneira mais fácil de criar programas realmente portáveis.

O fato de um programa Java ser executado pela JVM também ajuda a torná-lo seguro. Já que a JVM está no controle, ela pode reter o programa e impedir-lo de gerar efeitos colaterais fora do sistema. A segurança também é aumentada por certas restrições existentes na linguagem Java.

Quando um programa é executado por uma máquina virtual, geralmente ele é executado mais lentamente do que o mesmo programa sendo executado quando compilado para código de máquina. No entanto, em Java, a diferença entre os dois não é tão grande. Já que o bytecode foi altamente otimizado, seu uso permite que a JVM execute programas de maneira muito mais rápida do que o esperado. Além disso, é possível usar a compilação dinâmica de bytecode para código de máquina visando a melhoria do desempenho, o que pode ser feito com o uso de um compilador *just-in-time* (JIT) para bytecode.

Quando um compilador JIT faz parte da JVM, partes de bytecode selecionadas são compiladas em tempo real, fragmento a fragmento e sob demanda para código executável. É importante ressaltar que um compilador JIT não compila um programa Java inteiro para código executável de uma só vez. Em vez disso, um compilador JIT compila código quando necessário, durante a execução. Mas nem todas as sequências de bytecode são compiladas – só as que se beneficiarão da compilação. Até mesmo quando a compilação dinâmica é aplicada ao bytecode, os recursos de portabilidade e segurança continuam aplicáveis, porque a JVM ainda está no comando do ambiente de execução.

Uma última coisa: a JVM faz parte do sistema Java de tempo de execução, que também é chamado de Java Runtime Environment (JRE).

A evolução de Java

Só algumas linguagens reformularam de maneira fundamental a essência básica da programação. Nesse grupo de elite, Java se destaca porque seu impacto foi rápido e difuso. Não é exagero dizer que o lançamento original de Java 1.0 pela Sun Microsystems, Inc., causou uma revolução na programação. Além de ter ajudado a transformar a Web em um ambiente altamente interativo, Java também definiu um novo padrão no projeto de linguagens de computador.

Com o passar dos anos, Java continuou a crescer, evoluir e se redefinir. Diferentemente de muitas outras linguagens, que são lentas na incorporação de novos recursos, Java com frequência está na dianteira do desenvolvimento das linguagens de computador. Uma razão para que isso ocorra é a cultura de inovação e mudança que foi criada ao seu redor. Como resultado, Java passou por várias atualizações – algumas relativamente pequenas, outras mais significativas.

Quando este texto foi escrito, a versão mais recente de Java se chamava Java SE 7, com Java Development Kit sendo chamado de JDK 7. O SE de Java SE 7 significa Standard Edition. Java SE 7 é a primeira grande versão de Java desde que a Sun Microsystems foi adquirida pela Oracle. Ela contém muitos recursos novos – vários deles serão apresentados no decorrer deste livro.

Pergunte ao especialista

P Você explicou que os applets são executados no lado do cliente (navegador) da Internet. Há um tipo paralelo de programa Java que seja executado no lado do servidor?

R Sim. Pouco tempo depois do lançamento inicial de Java, ficou óbvio que a linguagem também seria útil do lado do servidor. O resultado foi o *servlet*. Um servlet é um programa pequeno que é executado no servidor. Assim como os applets estendem dinamicamente a funcionalidade de um navegador Web, os servlets estendem dinamicamente a funcionalidade de um servidor Web. Logo, com o advento do servlet, Java se estendeu pelos dois lados da conexão cliente/servidor.

Verificação do progresso

1. O que é um applet?
2. O que é bytecode Java?
3. O uso de bytecode ajuda a resolver dois problemas da programação na Internet. Quais?

AS PRINCIPAIS CARACTERÍSTICAS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

A *programação orientada a objetos* (OOP, object-oriented programming) é a essência de Java. A metodologia orientada a objetos é inseparável da linguagem, e todos os programas Java são, pelo menos até certo ponto, orientados a objetos. Devido à importância da OOP para Java, é útil entendermos seus princípios básicos antes de escrever até mesmo um programa Java simples.

A OOP é uma maneira poderosa de abordar a tarefa de programar. As metodologias de programação mudaram drasticamente desde a invenção do computador, principalmente para acomodar a crescente complexidade dos programas. Por exemplo, quando os computadores foram inventados, a programação era feita pela ativação das instruções binárias da máquina com o uso do painel frontal do computador. Contanto que os programas tivessem apenas algumas centenas de instruções, essa abordagem funcionava. À medida que os programas cresceram, a linguagem simbólica foi inventada para que o programador pudesse lidar com programas maiores e cada vez mais complexos, usando representações simbólicas das instruções de máquina. Como os programas continuaram a crescer, foram introduzidas linguagens de alto nível que davam ao programador mais ferramentas para lidar com a complexidade. A primeira linguagem amplamente disseminada foi FORTRAN. Embora fosse uma primeira etapa bem impressionante, programas grandes em FORTRAN eram muito difíceis de entender.

Respostas:

1. Um applet é um programa pequeno que é baixado dinamicamente da Web.
2. Um conjunto altamente otimizado de instruções que pode ser executado pela Máquina Virtual Java (JVM).
3. Portabilidade e segurança.

Os anos de 1960 deram origem à programação estruturada. Esse é o método encorajado por linguagens como C e Pascal. O uso de linguagens estruturadas tornou possível criar programas de complexidade moderada mais facilmente. As linguagens estruturadas são caracterizadas por seu suporte a sub-rotinas autônomas, variáveis locais, estruturas de controle sofisticadas e por não dependerem de GOTO. Embora sejam uma ferramenta poderosa, elas também têm um limite.

Considere isto: a cada marco no desenvolvimento da programação, técnicas e ferramentas eram criadas para permitir que o programador lidasse com a crescente complexidade. A cada etapa do percurso, a nova abordagem pegava os melhores elementos dos métodos anteriores e fazia avanços. Antes da invenção da OOP, muitos projetos estavam perto do ponto de ruptura (ou excedendo-o). Os métodos orientados a objetos foram criados para ajudar os programadores a ultrapassar essas barreiras.

A programação orientada a objetos pegou as melhores ideias da programação estruturada e combinou-as com vários conceitos novos. O resultado foi uma maneira diferente de organizar um programa. De um modo mais geral, um programa pode ser organizado de uma entre duas maneiras: a partir de seu código (o que está ocorrendo) ou a partir de seus dados (o que está sendo afetado). Com o uso somente de técnicas de programação estruturada, normalmente os programas são organizados a partir do código. Essa abordagem pode ser considerada como “o código atuando sobre os dados”.

Os programas orientados a objetos funcionam ao contrário. São organizados a partir dos dados, com o seguinte princípio-chave: “dados controlando o acesso ao código”. Em uma linguagem orientada a objetos, você define os dados e as rotinas que podem atuar sobre eles. Logo, um tipo de dado define precisamente que tipo de operações pode ser aplicado a esses dados.

Para dar suporte aos princípios da programação orientada a objetos, todas as linguagens OOP, inclusive Java, têm três características em comum: encapsulamento, polimorfismo e herança. Examinemos cada uma.

Encapsulamento

O *encapsulamento* é um mecanismo de programação que vincula o código e os dados que ele trata, e isso mantém os dois seguros contra a interferência e a má utilização externa. Em uma linguagem orientada a objetos, o código e os dados podem ser vinculados de tal forma que uma *caixa preta* autônoma seja criada. Dentro da caixa, estão todo o código e os dados necessários. Quando o código e os dados são vinculados dessa maneira, um objeto é criado. Em outras palavras, um objeto é o dispositivo que dá suporte ao encapsulamento.

Dentro de um objeto, o código, os dados ou ambos podem ser *privados* desse objeto ou *públicos*. O código ou os dados privados só são conhecidos e acessados por outra parte do objeto. Isto é, o código ou os dados privados não podem ser acessados por uma parte do programa que exista fora do objeto. Quando o código ou os dados são públicos, outras partes do programa podem acessá-los mesmo que estejam definidos dentro de um objeto. Normalmente, as partes públicas de um objeto são usadas para fornecer uma interface controlada para os elementos privados do objeto.

A unidade básica de encapsulamento de Java é a *classe*. Embora a classe seja examinada com mais detalhes posteriormente neste livro, a breve discussão a seguir será útil agora. Uma classe define a forma de um objeto. Ela especifica tanto os dados quanto o código que operará sobre eles. Java usa uma especificação de classe

para construir *objetos*. Os objetos são *instâncias* de uma classe. Logo, uma classe é essencialmente um conjunto de planos que especificam como construir um objeto.

O código e os dados que constituem uma classe são chamados de *membros* da classe. Especificamente, os dados definidos pela classe são chamados de *variáveis membro* ou *variáveis de instância*. Os códigos que operam sobre esses dados são chamados de *métodos membro* ou apenas *métodos*.

Polimorfismo

Polimorfismo (do grego, “muitas formas”) é a qualidade que permite que uma interface acesse uma classe geral de ações. A ação específica é determinada pela natureza exata da situação. Um exemplo simples de polimorfismo é encontrado no volante de um automóvel. O volante (isto é, a interface) é o mesmo não importando o tipo de mecanismo de direção usado. Ou seja, o volante funciona da mesma forma se seu carro tem direção manual ou direção hidráulica. Portanto, se você souber como operar o volante, poderá dirigir qualquer tipo de carro, não importando como a direção foi implementada.

O mesmo princípio também pode ser aplicado à programação. Vejamos um exemplo simples. Você poderia criar uma interface para definir uma operação chamada **get**, que obtivesse o próximo item de dados de algum tipo de lista. Essa ação, a obtenção do próximo item, pode ser implementada de várias maneiras, dependendo de como os itens são armazenados. Por exemplo, os itens podem ser armazenados na ordem primeiro a entrar, primeiro a sair; na ordem primeiro a entrar, último a sair; com base em alguma prioridade; ou de alguma maneira entre muitas outras. Porém, se todos os mecanismos de armazenamento implementarem sua interface, você poderá usar **get** para recuperar o próximo item.

Geralmente, o conceito de polimorfismo é representado pela expressão “uma interface, vários métodos”. Ou seja, é possível projetar uma interface genérica para um grupo de atividades relacionadas. O polimorfismo ajuda a reduzir a complexidade permitindo que a mesma interface seja usada para especificar uma *classe geral de ação*. É tarefa do compilador selecionar a *ação* (isto é, método) *específica* conforme cada situação. Você, o programador, não precisa fazer essa seleção manualmente. Só tem que usar a interface geral.

Herança

Herança é o processo pelo qual um objeto pode adquirir as propriedades de outro objeto. Isso é importante porque dá suporte ao conceito de classificação hierárquica. Se você pensar bem, grande parte do conhecimento pode ser gerenciada por classificações hierárquicas (isto é, top-down). Por exemplo, uma maçã Red Delicious faz parte da classificação *maçã*, que por sua vez faz parte da classe *fruta*, que fica sob a classe maior *alimento*. Isto é, a classe *alimento* possui certas qualidades (comestível, nutritivo, etc.) que, logicamente, também se aplicam à sua subclasse, *fruta*. Além dessas qualidades, a classe *fruta* tem características específicas (suculenta, doce, etc.) que a distinguem de outros alimentos. A classe *maçã* define as qualidades específicas de uma maçã (cresce em árvores, não é tropical, etc.). Por sua vez, uma maçã Red Delicious herdaria as qualidades de todas as classes precedentes e só definiria as qualidades que a tornam única.

Sem o uso de hierarquias, cada objeto teria que definir explicitamente todas as suas características. Com o uso da herança, um objeto só tem que definir as qualida-

des que o tornam único dentro de sua classe. Ele pode herdar seus atributos gerais de seu pai. Logo, é o mecanismo de herança que possibilita um objeto ser uma instância específica de um caso mais geral.

Verificação do progresso

1. Cite os princípios da OOP.
2. Qual é a unidade básica de encapsulamento em Java?

O JAVA DEVELOPMENT KIT

Agora que a história e a base teórica de Java foram explicadas, é hora de começar a escrever programas Java. No entanto, antes de você poder compilar e executar esses programas, precisa ter o Java Development Kit (JDK) instalado em seu computador.

Nota: Se quiser instalá-lo em seu computador, o JDK pode ser baixado de www.oracle.com/technetwork/java/javase/downloads/index.html. Siga as instruções para o tipo de computador que você tem. Após ter instalado o JDK, você poderá compilar e executar programas.

O JDK fornece dois programas principais. O primeiro é o **javac**, que é o compilador Java. Ele converte código-fonte em bytecode. O segundo é o **java**. Também chamado de *iniciador de aplicativos*, esse é o programa que você usará para executar um programa Java. Ele opera sobre o bytecode, usando a JVM para executar o programa.

Mais uma coisa: o JDK é executado no ambiente de prompt de comando e usa ferramentas de linha de comando. Ele não é um aplicativo de janelas. Também não é um ambiente de desenvolvimento integrado (IDE, integrated development environment).

Nota: Além das ferramentas básicas de linha de comando fornecidas com o JDK, há vários ambientes de desenvolvimento integrado de alta qualidade disponíveis para Java. Um IDE pode ser muito útil no desenvolvimento e na implantação de aplicativos comerciais. Como regra geral, você também pode usar um IDE para compilar e executar os programas deste livro, se assim quiser. No entanto, as instruções apresentadas aqui para a compilação e execução de um programa Java só descrevem as ferramentas de linha de comando do JDK. É fácil entender o motivo. Em primeiro lugar, o JDK está prontamente disponível. Em segundo lugar, as instruções para uso do JDK são as mesmas para todos os ambientes. Em terceiro lugar, devido às diferenças entre os IDEs, não é possível fornecer um conjunto geral de instruções que funcione para todas as pessoas.

Respostas:

1. Encapsulamento, polimorfismo e herança.
2. A classe.

Pergunte ao especialista

P Você diz que a programação orientada a objetos é uma maneira eficaz de gerenciar programas grandes. No entanto, parece que ela pode adicionar uma sobrecarga significativa aos relativamente pequenos. Já que você diz que todos os programas Java são, até certo ponto, orientados a objetos, isso dá uma desvantagem aos programas pequenos?

R Não. Como você verá, para programas pequenos, os recursos orientados a objetos de Java são quase transparentes. É verdade que Java segue um modelo de objeto rigoroso, mas você é livre para decidir até que nível quer empregá-lo. Em programas pequenos, a “orientação a objetos” é quase imperceptível. À medida que seus programas crescerem, você poderá integrar mais recursos orientados a objetos sem esforço.

UM PRIMEIRO PROGRAMA SIMPLES

A melhor maneira de introduzir vários dos elementos-chave de Java é compilando e executando um exemplo de programa curto. Usaremos o mostrado aqui:

```
/*
Este é um programa Java simples.

Chame este arquivo de Example.java.
*/
class Example {
    // Um programa Java começa com uma chamada a main().
    public static void main(String[] args) {
        System.out.println("Java drives the Web.");
    }
}
```

Você seguirá estas três etapas:

1. Insira o programa.
2. Compile o programa.
3. Execute o programa.

Inserindo o programa

A primeira etapa da criação de um programa é inserir seu código-fonte no computador. Como explicado antes, o código-fonte é a forma do programa legível para humanos. Você deve inserir o programa em seu computador usando um editor e não um processador de texto. Normalmente, os processadores de texto armazenam informações de formato junto com o texto, as quais confundirão o compilador Java. O código-fonte deve ser composto somente por texto. Se estiver usando um IDE, ele fornecerá um editor de código-fonte para você usar. Caso contrário, qualquer editor de texto simples servirá. Por exemplo, se você estiver usando o Windows, pode usar o Bloco de Notas.

Na maioria das linguagens de computador, o nome do arquivo que contém o código-fonte de um programa é arbitrário. Porém, não é esse o caso em Java. A primeira coisa que você deve aprender sobre Java é que *o nome dado a um arquivo*

-fonte é muito importante. Para esse exemplo, o nome do arquivo-fonte deve ser **Example.java**. Vejamos o porquê.

Em Java, um arquivo-fonte é chamado oficialmente de *unidade de compilação*. É um arquivo de texto que contém (entre outras coisas) uma ou mais definições de classe. (Por enquanto, usaremos arquivos-fonte contendo apenas uma classe.) O compilador Java requer que o arquivo-fonte use a extensão de nome de arquivo **.java**. Como você pode ver examinando o programa, o nome da classe definida por ele também é **Example**. Isso não é coincidência. Em Java, todo código deve residir dentro de uma classe. Por convenção, o nome da classe principal deve coincidir com o nome do arquivo que contém o programa. Você também deve se certificar de que a capitalização do nome do arquivo coincida com a do nome da classe. Isso ocorre porque Java diferencia maiúsculas de minúsculas. Nesse momento, a convenção de que os nomes de arquivo devem corresponder aos nomes das classes pode parecer arbitrária, mas segui-la facilita a manutenção e a organização dos programas.

Compilando o programa

Antes de executar o programa, você deve compilá-lo usando o **javac**. Para compilar o programa **Example**, execute o **javac**, especificando o nome do arquivo-fonte na linha de comando, como mostrado aqui:

```
| javac Example.java
```

O compilador **javac** criará um arquivo chamado **Example.class** contendo a versão em bytecode do programa. Lembre-se, bytecode não é código executável. Ele deve ser executado por uma Máquina Virtual Java. Não pode ser executado diretamente pela CPU.

Executando o programa

Para executar realmente o programa, você deve usar **java**. Lembre-se, **java** opera sobre a forma bytecode do programa. Para executar o programa **Example**, passe o nome da classe **Example** como argumento de linha de comando, como mostrado abaixo:

```
| java Example
```

Quando o programa for executado, a saída a seguir será exibida:

```
| Java drives the Web.
```

Quando o código-fonte Java é compilado, cada classe é inserida em seu próprio arquivo de saída com o mesmo nome da classe usando a extensão **.class**. Por isso, é uma boa ideia dar a um arquivo-fonte Java o mesmo nome da classe que ele contém – o nome do arquivo-fonte coincidirá com o nome do arquivo **.class**. Quando você executar **java** como acabamos de mostrar, estará especificando o nome da classe que deseja executar. **Java** procurará automaticamente um arquivo com esse nome que tenha a extensão **.class**. Se encontrar, executará o código contido na classe especificada.

Primeiro exemplo de programa linha a linha

Embora **Example.java** seja bem curto, ele inclui vários recursos-chave que são comuns a todos os programas Java. Examinemos com detalhes cada parte do programa.

O programa começa com as linhas a seguir:

```
/*
Esse é um programa Java simples.

Chame esse arquivo de Example.java.
*/
```

Isso é um *comentário*. Como a maioria das outras linguagens de programação, Java permite a inserção de uma observação no arquivo-fonte de um programa. O conteúdo de um comentário é ignorado pelo compilador. Em vez disso, o comentário descreve ou explica a operação do programa para quem estiver lendo seu arquivo-fonte. Nesse caso, ele está descrevendo o programa e lembrando que o arquivo-fonte deve se chamar **Example.java**. É claro que, em aplicativos reais, geralmente os comentários explicam como alguma parte do programa funciona ou o que um recurso específico faz.

O comentário mostrado no início do programa se chama *comentário de várias linhas*. Esse tipo de comentário começa com `/*` e termina com `*/`. Qualquer coisa que estiver entre esses dois símbolos de comentário será ignorada pelo compilador. Como o nome sugere, um comentário de várias linhas pode ter muitas linhas.

A próxima linha de código do programa é mostrada aqui:

```
| class Example {
```

Essa linha usa a palavra-chave **class** para declarar que uma nova classe está sendo definida. Como mencionado, a classe é a unidade básica de encapsulamento de Java. **Example** é o nome da classe. A definição da classe começa com a chave de abertura (`{`) e termina com a chave de fechamento (`}`). Os elementos existentes entre as duas chaves são membros da classe. Por enquanto, não se preocupe tanto com os detalhes de uma classe; é preciso saber apenas que em Java toda a atividade do programa ocorre dentro de uma. Essa é uma das razões por que todos os programas Java são (pelo menos um pouco) orientados a objetos.

A linha seguinte do programa é o *comentário de linha única*, mostrado aqui:

```
| // Um programa Java começa com uma chamada a main().
```

Esse é o segundo tipo de comentário suportado por Java. Um comentário de linha única começa com `//` e termina no fim da linha. Como regra geral, os programadores usam comentários de várias linhas para observações mais longas e comentários de linha única para descrições breves, linha a linha.

A próxima linha de código é a mostrada abaixo:

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

Essa linha começa o método **main()**. Como mencionado anteriormente, em Java, uma sub-rotina é chamada de *método*. Como o comentário que a precede sugere, essa é a linha em que o programa começará a ser executado. Todos os aplicativos Java começam a execução chamando **main()**. O significado exato de cada parte dessa linha não pode ser fornecido agora, já que envolve uma compreensão detalhada de vários outros recursos da linguagem Java. No entanto, como muitos dos exemplos deste livro usarão essa linha de código, um resumo lhe dará uma ideia geral do que ela significa.

A linha começa com a palavra-chave **public**. Ela é um *modificador de acesso*. Um modificador de acesso determina como outras partes do programa podem acessar os membros da classe. Quando o membro de uma classe é precedido por **public**, ele pode ser acessado por um código de fora da classe em que foi declarado. (O oposto de **public** é **private**, que impede que um membro seja usado por um código definido fora de sua classe.) O método **main()** deve ser declarado como **public** porque é executado por um código de fora da classe **Example**. (Nesse caso, é o iniciador de aplicativos **java** que chama **main()**.)

A palavra-chave **static** permite que **main()** seja executado independentemente de qualquer objeto. Isso é necessário porque **main()** é executado pela JVM antes de qualquer objeto ser criado. A palavra-chave **void** simplesmente informa ao compilador que **main()** não retorna um valor. (Como você verá, os métodos também podem retornar valores.) Se tudo isso parece um pouco confuso, não se preocupe. Todos esses conceitos serão discutidos com detalhes em capítulos subsequentes.

Como mencionado, **main()** é o método chamado quando um aplicativo Java começa a ser executado. Qualquer informação que você tiver que passar para um método será recebida por variáveis especificadas dentro do conjunto de parênteses que seguem o nome do método. Essas variáveis são chamadas de *parâmetros*. (Mesmo se nenhum parâmetro for necessário em um determinado método, você terá que incluir os parênteses vazios.) O método **main()** requer que haja um parâmetro. Isso é especificado no programa **Example** com **String args[]**, que declara um parâmetro chamado **args**. Ele é um array de objetos de tipo **String**. (Arrays são conjuntos de objetos semelhantes.) Os objetos de tipo **String** armazenam sequências de caracteres. (Tanto os arrays quanto o tipo **String** serão discutidos com detalhes em capítulos subsequentes.) Nesse caso, **args** recebe qualquer argumento de linha de comando presente quando o programa é executado. O programa **Example** não usa argumentos de linha de comando, mas outros programas mostrados posteriormente neste livro usarão.

O último caractere da linha é **{**. Ele sinaliza o início do corpo de **main()**. Todo o código incluído em um método ocorrerá entre a chave de abertura do método e sua chave de fechamento.

A próxima linha de código é mostrada a seguir. Observe que ela ocorre dentro de **main()**.

```
| System.out.println("Java drives the Web.");
```

Essa linha exibe o string “Java drives the Web.” seguida por uma nova linha na tela. Na verdade, a saída é exibida pelo método interno **println()**. Nesse caso, **println()** exibe o string que é passado para ele. Como você verá, **println()** também pode ser usado para exibir outros tipos de informações. A linha começa com **System.out**. Embora seja muito complicada para explicarmos com detalhes nesse momento, **System**, em resumo, é uma classe predefinida que dá acesso ao sistema, e **out** é o fluxo de saída que está conectado ao console. Portanto, **System.out** é um objeto que encapsula a saída do console. O fato de Java usar um objeto para definir a saída do console é mais uma evidência de sua natureza orientada a objetos.

Como você deve ter notado, a saída (e a entrada) do console não é usada com frequência em aplicativos Java do mundo real. Já que a maioria dos ambientes de computação modernos tem janelas e é gráfica, o I/O do console é mais usado para

programas utilitários simples, programas de demonstração (como os deste livro) e código do lado do servidor. Posteriormente, você aprenderá a criar interfaces gráficas de usuário (GUIs), mas, por enquanto, continuaremos a usar os métodos de I/O do console. Observe que a instrução `println()` termina com um ponto e vírgula. Todas as instruções em Java terminam com um ponto e vírgula. As outras linhas do programa não terminam em um ponto e vírgula porque, tecnicamente, não são instruções.

O primeiro símbolo } do programa termina `main()` e o último termina a definição da classe `Example`.

Um último ponto: Java diferencia maiúsculas de minúsculas. Esquecer disso pode causar problemas graves. Por exemplo, se você digitar accidentalmente `Main` em vez de `main`, ou `PrintLn` em vez de `println`, o programa anterior estará incorreto. Além disso, embora o compilador Java *compile* classes que não contêm um método `main()`, ele não tem como executá-las. Logo, se você digitasse errado `main`, o compilador compilaria seu programa. No entanto, o programa `java` relataria um erro por não conseguir encontrar o método `main()`.

Verificação do progresso

1. Onde um programa Java começa a ser executado?
2. O que `System.out.println()` faz?
3. Qual é o nome do compilador Java? O que você deve usar para executar um programa Java?

TRATANDO ERROS DE SINTAXE

Se ainda não tiver feito isso, insira, compile e execute o programa anterior. Como você deve saber, é muito fácil digitar algo incorretamente por acidente ao inserir código no computador. Felizmente, se você inserir algo errado em seu programa, o compilador exibirá uma mensagem de *erro de sintaxe* quando tentar compilá-lo. O compilador Java tenta entender o código-fonte não importando o que foi escrito. Portanto, o erro que é relatado nem sempre reflete a causa real do problema. No programa anterior, por exemplo, uma omissão accidental da chave de abertura depois do método `main()` faria o compilador relatar os dois erros a seguir:

```
Example.java:8: ';' expected
    public static void main(String[] args)
                           ^
Example.java:11: class, interface, or enum expected
}
^
```

Respostas:

1. `main()`
2. Exibe informações no console.
3. O compilador Java padrão é o `javac`. Para executar um programa Java, use o utilitário `java`.

É claro que a primeira mensagem de erro está totalmente errada, porque o que está faltando não é um ponto e vírgula, mas uma chave. A segunda mensagem de erro não está errada, mas é simplesmente resultado do compilador tentar interpretar o resto do programa após sua sintaxe ter sido distorcida pela chave ausente.

O importante nessa discussão é que, quando seu programa tiver um erro de sintaxe, você não deve aceitar literalmente as mensagens do compilador. Elas podem ser enganosas. Você pode ter de “decifrar” uma mensagem de erro para encontrar o problema real. Examine também as últimas linhas de código de seu programa que antecedem a linha que está sendo indicada. Às vezes, um erro só é relatado várias linhas após o ponto em que ele realmente ocorreu.

UM SEGUNDO PROGRAMA SIMPLES

Talvez nenhuma outra estrutura seja tão importante para uma linguagem de programação quanto a atribuição de um valor a uma variável. Uma *variável* é um local nomeado na memória ao qual pode ser atribuído um valor. Além disso, o valor de uma variável pode ser alterado durante a execução de um programa, isto é, o conteúdo de uma variável é alterável e não fixo.

O programa a seguir cria duas variáveis chamadas **var1** e **var2**. Observe como elas são usadas:

```
/*
Este código demonstra uma variável.

Chame este arquivo de Example2.java.
*/
class Example2 {
    public static void main(String[] args) {
        int var1; // esta instrução declara uma variável ←———— Declara variáveis.
        int var2; // esta instrução declara outra variável

        var1 = 1024; // esta instrução atribui 1024 a var1 ←———— Atribui um valor
                     a uma variável.

        System.out.println("var1 contains" + var1);

        var2 = var1 / 2;

        System.out.print("var2 contains var1 / 2: ");
        System.out.println(var2);
    }
}
```

Quando você executar esse programa, verá a saída abaixo:

```
var1 contains 1024
var2 contains var1 / 2: 512
```

Esse programa introduz vários conceitos novos. Primeiro, a instrução

```
| int var1; // essa instrução declara uma variável
```

declara uma variável chamada **var1** de tipo inteiro. Em Java, todas as variáveis devem ser declaradas antes de serem usadas. Além disso, o tipo de valor que a variável pode conter também deve ser especificado. Ele é chamado de *tipo* da variável. Nesse caso, **var1** pode conter valores inteiros. São valores que representam números inteiros. Em Java, para declarar uma variável como de tipo inteiro, é preciso preceder seu nome com a palavra-chave **int**. Portanto, a instrução anterior declara uma variável chamada **var1** de tipo **int**.

A linha seguinte declara uma segunda variável chamada **var2**:

```
| int var2; // essa instrução declara outra variável
```

Observe que essa linha usa o mesmo formato da primeira, exceto pelo nome da variável ser diferente.

Em geral, para declarar uma variável, usamos uma instrução como esta:

tipo nome-var;

Aqui, *tipo* especifica o tipo de variável que está sendo declarado, e *nome-var* é o nome da variável. Além de **int**, Java dá suporte a vários outros tipos de dados.

A linha de código abaixo atribui a **var1** o valor 1024:

```
| var1 = 1024; // essa instrução atribui 1024 a var1
```

Em Java, o operador de atribuição é o sinal de igualdade simples. Ele copia o valor do lado direito para a variável à sua esquerda.

A próxima linha de código exibe o valor de **var1** precedido pelo string “var1 contains”:

```
| System.out.println("var1 contains" + var1);
```

Nessa instrução, o sinal de adição faz o valor de **var1** ser exibido após o string que o precede. Essa abordagem pode ser generalizada. Usando o operador +, você pode encadear quantos itens quiser dentro da mesma instrução **println()**.

A linha de código a seguir atribui a **var2** o valor de **var1** dividido por 2:

```
| var2 = var1 / 2;
```

Essa linha divide o valor de **var1** por 2 e armazena o resultado em **var2**. Portanto, após a linha ser executada, **var2** conterá o valor 512. O valor de **var1** permanecerá inalterado. Como a maioria das outras linguagens de computador, Java dá suporte a um conjunto completo de operadores aritméticos, inclusive os mostrados aqui:

+	Adição
-	Subtração
*	Multiplicação
/	Divisão

Estas são as duas linhas seguintes do programa:

```
| System.out.print("var2 contains var1/2: ");
| System.out.println(var2);
```

Dois fatos novos estão ocorrendo aqui. Em primeiro lugar, o método interno **print()** é usado para exibir o string “var2 contains var1 / 2: ”. Esse string *não* é seguido por uma nova linha. Ou seja, quando a próxima saída for gerada, ela começará na mesma linha. O método **print()** é exatamente igual a **println()**, exceto por não exibir uma nova linha após cada chamada. Em segundo lugar, na chamada a **println()**, observe que **var2** é usada sozinha. Tanto **print()** quanto **println()** podem ser usados para exibir valores de qualquer um dos tipos internos de Java.

Mais uma coisa sobre a declaração de variáveis antes de avançarmos: é possível declarar duas ou mais variáveis usando a mesma instrução de declaração. Apenas separe seus nomes com vírgulas. Por exemplo, **var1** e **var2** poderiam ter sido declaradas assim:

```
| int var1, var2; // as duas declaradas com o uso de uma instrução
```

OUTRO TIPO DE DADO

No programa anterior, uma variável de tipo **int** foi usada. No entanto, a variável de tipo **int** só pode conter números inteiros. Logo, não pode ser usada quando um componente fracionário for necessário. Por exemplo, uma variável **int** pode conter o valor 18, mas não o valor 18,3. Felizmente, **int** é apenas um dos vários tipos de dados definidos por Java. Para permitir números com componentes fracionários, Java define dois tipos de ponto flutuante: **float** e **double**, que representam valores de precisão simples e dupla, respectivamente. Dos dois, **double** é o mais usado.

Para declarar uma variável de tipo **double**, use uma instrução semelhante à mostrada abaixo:

```
| double x;
```

Aqui, **x** é o nome da variável, que é de tipo **double**. Já que **x** tem um tipo de ponto flutuante, pode conter valores como 122,23, 0,034 ou -19,0.

Para entender melhor a diferença entre **int** e **double**, teste o programa a seguir:

```
/*
Este programa ilustra as diferenças
entre int e double.

Chame este arquivo de Example3.java.
*/
class Example3 {
    public static void main(String[] args) {
        int w; // esta instrução declara uma variável int
        double x; // esta instrução declara uma variável de ponto flutuante

        w = 10; // atribui a w o valor 10

        x = 10.0; // atribui a x o valor 10,0
        System.out.println("Original value of w: " + w);
        System.out.println("Original value of x: " + x);
```

```

System.out.println(); // exibe uma linha em branco ← Exibe uma linha
                     em branco.

// agora, divide as duas por 4
w = w / 4;
x = x / 4;

System.out.println("w after division: " + w);
System.out.println("x after division: " + x);
}
}

```

A saída do programa é mostrada aqui:

```

Original value of w: 10
Original value of x: 10.0

w after division: 2 ← Componente fracionário perdido.
x after division: 2.5 ← Componente fracionário preservado.

```

Como você pode ver, quando **w** (uma variável **int**) é dividida por 4, uma divisão de números inteiros é executada e o resultado é 2 – o componente fracionário é perdido. No entanto, quando **x** (uma variável **double**) é dividida por 4, o componente fracionário é preservado e a resposta apropriada é exibida.

Há outro fato novo a ser observado no programa. Para exibir uma linha em branco, simplesmente chamamos **println()** sem nenhum argumento.

Pergunte ao especialista

P Por que Java tem tipos de dados diferentes para inteiros e valores de ponto flutuante? Isto é, por que não são todos valores numéricos do mesmo tipo?

R Java fornece tipos de dados diferentes para que você possa criar programas eficientes. Por exemplo, a aritmética de inteiros é mais rápida do que os cálculos de ponto flutuante. Logo, se você não precisar de valores fracionários, não terá que sofrer a sobrecarga associada aos tipos **float** ou **double**. Além disso, a quantidade de memória requerida para um tipo de dado pode ser menor do que a requerida para outro. Fornecendo tipos diferentes, Java permite que você use melhor os recursos do sistema. Para concluir, alguns algoritmos requerem (ou pelo menos se beneficiam do) o uso de um tipo de dado específico. Em geral, Java fornece vários tipos internos para proporcionar maior flexibilidade.

TENTE ISTO 1.1 Convertendo galões em litros

GalToLit.java

Embora os exemplos de programas anteriores ilustrem vários recursos importantes da linguagem Java, eles não são muito úteis. Mesmo que você ainda não saiba muito sobre Java, pode colocar em ação o que aprendeu para criar um programa prático. Neste projeto, criaremos um programa que converte galões em litros. O programa funcionará declarando duas variáveis **double**. Uma conterá o número de galões e a outra o número de litros após a conversão. Um galão é equivalente a 3,7854 litros. Logo, na conversão de galões em litros, o valor do galão é multiplicado por 3,7854. O programa exibe tanto o número de galões quanto o número equivalente em litros.

PASSO A PASSO

1. Crie um novo arquivo chamado **GalToLit.java**.
2. Insira o programa a seguir no arquivo:

```
/*
Tente isto 1-1

Este programa converte galões em litros.

Chame-o de GalToLit.java.

*/
class GalToLit {
    public static void main(String[] args) {
        double gallons; // contém o número de galões
        double liters; // contém a conversão para litros

        gallons = 10; // começa com 10 galões

        liters = gallons * 3.7854; // converte para litros

        System.out.println(gallons + " gallons is " + liters +
                           " liters.");
    }
}
```

3. Compile o programa usando a linha de comando a seguir:

```
| javac GalToLit.java
```

4. Execute o programa usando este comando:

```
| java GalToLit
```

Você verá esta saída:

```
| 10.0 gallons is 37.854 liters.
```

5. Como se encontra, este programa converte 10 galões em litros. No entanto, alterando o valor atribuído a **gallons**, você pode fazer o programa converter um número diferente de galões em seu número equivalente em litros.

Verificação do progresso

1. Qual é a palavra-chave Java para o tipo de dado inteiro?
2. O que é **double**?

DUAS INSTRUÇÕES DE CONTROLE

Dentro de um método, a execução prossegue na sequência em que as instruções ocorrem. Em outras palavras, a execução se dá da instrução atual para a próxima, de cima para baixo. No entanto, com frequência queremos alterar esse fluxo com base em algumas condições. Essas situações são extremamente comuns em programação. Veja um exemplo: um site pode pedir uma senha e seu código não deve dar acesso a ele se a senha for inválida. Logo, o código que dá acesso não deve ser executado se uma senha inválida for inserida. Continuando com o exemplo, se uma senha inválida fosse inserida, você poderia dar ao usuário mais duas (e somente duas) oportunidades de inseri-la corretamente. Para tratar situações em que o fluxo de execução do programa deve ser alterado, Java fornece um amplo conjunto de *instruções de controle*. Vamos examinar as instruções de controle com detalhes no Capítulo 3, mas duas delas, **if** e **for**, serão introduzidas brevemente aqui porque iremos usá-las para criar exemplos de programas.

A instrução if

Você pode executar seletivamente parte de um programa com o uso da instrução **if**. A instrução **if** é a instrução básica de “tomada de decisão” em Java. Como tal, é um dos elementos básicos de Java e da programação em geral. Você usaria uma instrução **if** para determinar se um número é menor do que outro, para saber se uma variável contém um valor de destino ou para verificar alguma condição de erro, apenas para citar três exemplos entre muitos.

A forma mais simples de **if** é mostrada abaixo:

```
if(condição) instrução;
```

Aqui, *condição* é uma expressão que tem resultado verdadeiro ou falso. (Esse tipo de expressão é chamado de *expressão booleana*.) Se a *condição* for verdadeira, a instrução será executada. Se a *condição* for falsa, a instrução será ignorada. Logo, a condição controla se a instrução que vem a seguir será ou não executada. Veja um exemplo:

```
| if(10 < 11) System.out.println("10 is less than 11");
```

Nessa linha, o operador **<** (menor que) é usado para verificarmos se 10 é menor do que 11. Como 10 é menor do que 11, a expressão condicional é verdadeira e **println()** será executado. No entanto, considere o seguinte:

```
| if(10 < 9) System.out.println("this won't be displayed");
```

Respostas:

1. **int**
2. A palavra-chave do tipo de dado de ponto flutuante de dupla precisão (**double**).

Neste caso, 10 não é menor do que 9. Logo, a chamada a `println()` não ocorrerá.

O símbolo `<` é apenas um dos *operadores relacionais* de Java. Um operador relacional determina o relacionamento entre dois valores. Java define uma lista completa dos operadores relacionais que podem ser usados em uma expressão condicional. Eles são mostrados aqui:

Operador	Significado
<code><</code>	Menor que
<code><=</code>	Menor ou igual
<code>></code>	Maior que
<code>>=</code>	Maior ou igual
<code>==</code>	Igual a
<code>!=</code>	Diferente

Observe que o teste de igualdade usa o sinal de igual duplo. Em todos os casos, o resultado de um operador relacional é um valor verdadeiro ou falso.

Aqui está um programa que ilustra a instrução `if` e vários operadores relacionais:

```
/*
Demonstra a instrução if.

Chame este arquivo de IfDemo.java.
*/
class IfDemo {
    public static void main(String[] args) {
        int a, b, c;

        a = 2;
        b = 3;

        if(a < b) System.out.println("a is less than b");

        // essa instrução não exibirá nada
        if(a == b) System.out.println("you won't see this");

        System.out.println();

        c = a - b; // c contém -1

        System.out.println("c contains -1");
        if(c >= 0) System.out.println("c is non-negative");
        if(c < 0) System.out.println("c is negative");

        System.out.println();

        c = b - a; // agora c contém 1

        System.out.println("c contains 1");
```

```
if(c >= 0) System.out.println("c is non-negative");
if(c < 0) System.out.println("c is negative");
}
```

A saída gerada pelo programa é mostrada abaixo:

```
a is less than b

c contains -1
c is negative

c contains 1
c is non-negative
```

Observe outra coisa nesse programa. A linha

```
int a, b, c;
```

declara três variáveis, **a**, **b** e **c**, usando uma lista separada por vírgulas. Como mencionado anteriormente, quando você precisar de duas ou mais variáveis do mesmo tipo, elas poderão ser declaradas na mesma instrução. Apenas separe os nomes das variáveis com vírgulas.

Pergunte ao especialista

P Na discussão da instrução **if**, você mencionou que uma expressão verdadeiro/falso é chamada de expressão booleana. Por que esse termo é usado?

R O termo *booleana* é em homenagem a George Boole (1815-1864). Ele desenvolveu e formalizou as leis que controlam as expressões verdadeiro/falso. Isso ficou conhecido como álgebra booleana. Seu trabalho acabou formando a base da lógica dos computadores.

O laço **for**

Em vários momentos um programa terá que executar uma tarefa mais de uma vez. Por exemplo, você poderia querer exibir a hora do dia, com a atualização ocorrendo a cada segundo. É claro que não seria prático criar um programa assim usando centenas de instruções **println()** separadas, uma para cada hora possível, em intervalos de um segundo. Em vez disso, essa operação repetitiva seria executada por um *laço*. Um laço é uma instrução de controle que executa repetidamente uma sequência de código. Os laços são amplamente usados por quase todos os programas. Como a instrução **if**, eles são uma parte fundamental da programação.

Java fornece um grupo poderoso de estruturas de laço. A que introduziremos aqui é a do laço **for**. A forma mais simples do laço **for** é mostrada a seguir:

for(*inicialização*; *condição*; *iteração*) *instrução*;

Em sua forma mais comum, a parte de *inicialização* do laço define uma *variável de controle de laço* com um valor inicial. *Condição* é uma expressão booleana que

testa a variável de controle do laço. Se o resultado desse teste for verdadeiro, o laço **for** continuará a iterar. Se for falso, o laço será encerrado. A expressão de *iteração* determina como a variável de laço é alterada sempre que o laço itera. Aqui está um programa curto que ilustra o laço **for**:

```
/*
Demonstra o laço for.

Chame este arquivo de ForDemo.java.
*/
class ForDemo {
    public static void main(String[] args) {
        int count;

        for(count = 0; count < 5; count = count+1) ←— Este laço itera cinco vezes.
        System.out.println("This is count: " + count);

        System.out.println("Done!");
    }
}
```

A saída gerada pelo programa é mostrada aqui:

```
This is count: 0
This is count: 1
This is count: 2
This is count: 3
This is count: 4
Done!
```

Nesse exemplo, **count** é a variável de controle do laço. Ela é configurada com zero na parte de inicialização de **for**. No começo de cada iteração (inclusive a primeira), o teste condicional **count < 5** é executado. Se o resultado desse teste for verdadeiro, será executada a instrução **println()**, e então a parte de iteração do laço será executada. Esse processo continua até o teste condicional ser falso, momento em que a execução é retomada no final do laço. O interessante é que em programas Java criados profissionalmente quase nunca vemos a parte de iteração do laço escrita como mostrado no programa anterior. Isto é, raramente vemos instruções como esta:

```
| count = count + 1;
```

Isso ocorre porque Java inclui um operador de incremento especial que executa essa operação com mais eficiência. O operador de incremento é **++** (ou seja, dois sinais de adição seguidos). Ele aumenta seu operando em uma unidade. Com o uso do operador de incremento, a instrução anterior pode ser escrita assim:

```
| count++;
```

Logo, o laço **for** do programa anterior normalmente será escrito desta forma:

```
| for(count = 0; count < 5; count++)
```

Se quiser, faça o teste. Como verá, o laço continuará sendo executado exatamente como antes.

Java também fornece um operador de decremento, que é especificado na forma `--`. Esse operador diminui seu operando em uma unidade.

Verificação do progresso

1. O que a instrução **if** faz?
2. O que a instrução **for** faz?
3. Quais são os operadores relacionais Java?

CRIE BLOCOS DE CÓDIGO

Outro elemento-chave de Java é o *bloco de código*. Um bloco de código é um agrupamento de duas ou mais instruções. Isso é feito com a inclusão das instruções entre chaves de abertura e fechamento. Quando um bloco de código é criado, ele se torna uma unidade lógica que pode ser usada em qualquer local onde seria usada uma única instrução, o que é importante porque permite o uso de um conjunto de instruções como alvo de uma instrução de controle, como as instruções **if** ou **for**, descritas na seção anterior. Por exemplo, considere esta instrução **if**:

```
if(w < h) { ← Início do bloco
    v = w * h;
    w = 0;
} ← Fim do bloco
```

Aqui, o alvo da instrução **if** é um bloco de código que contém duas instruções. Se **w** for menor do que **h**, as duas instruções do bloco serão executadas. Se **w** não for menor do que **h**, o bloco será ignorado e nenhuma instrução será executada. Logo, as duas instruções do bloco formam uma unidade lógica, e uma instrução não pode ser executada sem a outra. Esse conceito pode ser generalizado: sempre que você precisar vincular logicamente duas ou mais instruções, pode fazer isso criando um bloco.

O programa a seguir demonstra um bloco de código usando-o como alvo de uma instrução **if** para impedir uma divisão por zero:

```
/*
Demonstra um bloco de código.

Chame este arquivo de BlockDemo.java.
*/
class BlockDemo {
    public static void main(String[] args) {
        double i, j, d;
```

Respostas:

1. A instrução **if** é a instrução condicional de Java.
2. A instrução **for** é uma das instruções de laço Java.
3. Os operadores relacionais são `==`, `!=`, `<`, `>`, `<=` e `>=`.

```

i = 5;
j = 10;

// o alvo desta instrução if é um bloco
if(i != 0) {
    System.out.println("i does not equal zero");
    d = j / i;
    System.out.print("j / i is " + d);
}
}
}

```

O alvo de **if** é este bloco inteiro.

A saída gerada por esse programa é mostrada abaixo:

```

i does not equal zero
j / i is 2.0

```

Nesse exemplo, o alvo da instrução **if** é um bloco de código que só é executado se **i** não for igual a zero. Se a condição que controla **if** for verdadeira (como é aqui), as três instruções do bloco serão executadas. Tente configurar **i** com zero e observe o resultado. Você verá que o bloco inteiro é ignorado.

Como veremos posteriormente, os blocos de código têm propriedades e usos adicionais. No entanto, a principal razão de sua existência é a criação de unidades de código logicamente inseparáveis.

Pergunte ao especialista

P O uso de um bloco de código introduz alguma ineficiência de tempo de execução? Em outras palavras, Java executa realmente { e }?

R Não. Os blocos de código não adicionam nenhuma sobrecarga. Na verdade, devido à sua habilidade de simplificar a codificação de certos algoritmos, geralmente seu uso aumenta a velocidade e a eficiência. Além disso, os símbolos { e } existem apenas no código-fonte do programa. Java não executa { ou }.

PONTO E VÍRGULA E POSICIONAMENTO

Em Java, o ponto e vírgula é um *separador* que é usado para terminar uma instrução. Isto é, cada instrução individual deve ser finalizada com ponto e vírgula. Ele indica o fim de uma entidade lógica.

Como você sabe, um bloco é um conjunto de instruções conectadas logicamente que são delimitadas por chaves de abertura e fechamento. Ele *não* é finalizado com ponto e vírgula. Já que é um grupo de instruções, com um ponto e vírgula após cada instrução, faz sentido que o bloco não seja terminado com ponto e vírgula; em vez disso, o fim do bloco é indicado pela chave de fechamento.

Java não reconhece o fim da linha como um terminador. Portanto, não importa onde inserimos uma instrução na linha. Por exemplo,

```
x = y;  
y = y + 1;  
System.out.println(x + " " + y);
```

é o mesmo que o seguinte, em Java:

```
x = y; y = y + 1; System.out.println(x + " " + y);
```

Além disso, os elementos individuais de uma instrução também podem ser inseridos em linhas separadas. Por exemplo, o código a seguir é perfeitamente aceitável:

```
System.out.println("This is a long line of output" +  
                    x + y + z +  
                    "more output");
```

A divisão de linhas longas dessa forma costuma ser usada para a criação de programas mais legíveis. Também pode ajudar a impedir que linhas excessivamente longas passem para a próxima linha.

PRÁTICAS DE RECUO

Você deve ter notado nos exemplos anteriores que certas instruções foram recuadas. Java é uma linguagem de forma livre, ou seja, não importa onde inserimos as instruções em uma linha em relação umas às outras. No entanto, com o passar dos anos, desenvolveu-se um estilo de recuo comum e aceito que proporciona programas mais legíveis. Este livro segue o estilo e é recomendável que você faça o mesmo. Usando esse estilo, você recuará um nível após cada chave de abertura e se moverá para trás em um nível após cada chave de fechamento. Certas instruções encorajam algum recuo adicional; elas serão abordadas posteriormente.

Verificação do progresso

1. Como é criado um bloco de código? O que ele faz?
2. Em Java, as instruções são terminadas com um_____.
3. Todas as instruções Java devem começar e terminar na mesma linha. Verdadeiro ou falso?

Respostas:

1. Um bloco é iniciado por uma chave de abertura e terminado com uma chave de fechamento. Ele cria uma unidade de código lógica.
2. ponto e vírgula
3. Falso.

TENTE ISTO 1-2 Melhorando o conversor de galões em litros

GalToLitTable.java

Você pode usar o laço **for**, a instrução **if** e blocos de código para criar uma versão melhorada do conversor de galões em litros desenvolvida na seção Tente isto 1-1. Essa nova versão exibirá uma tabela de conversões começando com 1 galão e terminando em 100 galões. A cada 10 galões, uma linha em branco será exibida. Isso é feito com o uso de uma variável chamada **counter** que conta o número de linhas que foram exibidas. Preste atenção especial no seu uso.

PASSO A PASSO

1. Crie um novo arquivo chamado **GalToLitTable.java**.
2. Insira o programa a seguir no arquivo:

```
/*
Tente isto 1-2

Este programa exibe uma tabela de conversões
de galões em litros.

Chame-o de "GalToLitTable.java".
*/
class GalToLitTable {
    public static void main(String[] args) {
        double gallons, liters;
        int counter;

        counter = 0; ← Inicialmente o contador de linhas é configurado com zero.
        for(gallons = 1; gallons <= 100; gallons++) {
            liters = gallons * 3.7854; // converte para litros
            System.out.println(gallons + " gallons is " +
                               liters + " liters.");

            counter++; ← Incrementa o contador de linhas a cada iteração do laço.
            // a cada décima linha, exibe uma linha em branco
            if(counter == 10) { ← Se o valor do contador for 10,
                System.out.println();           exibe uma linha em branco.
                counter = 0; // zera o contador de linhas
            }
        }
    }
}
```

3. Compile o programa usando a linha de comando abaixo:

```
| javac GalToLitTable.java
```

4. Execute o programa usando este comando:

```
| java GalToLitTable
```

Aqui está uma parte da saída que você verá:

```
1.0 gallons is 3.7854 liters.  
2.0 gallons is 7.5708 liters.  
3.0 gallons is 11.356200000000001 liters.  
4.0 gallons is 15.1416 liters.  
5.0 gallons is 18.927 liters.  
6.0 gallons is 22.712400000000002 liters.  
7.0 gallons is 26.4978 liters.  
8.0 gallons is 30.2832 liters.  
9.0 gallons is 34.0686 liters.  
10.0 gallons is 37.854 liters.  
  
11.0 gallons is 41.6394 liters.  
12.0 gallons is 45.42480000000005 liters.  
13.0 gallons is 49.2102 liters.  
14.0 gallons is 52.9956 liters.  
15.0 gallons is 56.781 liters.  
16.0 gallons is 60.5664 liters.  
17.0 gallons is 64.3518 liters.  
18.0 gallons is 68.1372 liters.  
19.0 gallons is 71.9226 liters.  
20.0 gallons is 75.708 liters.  
  
21.0 gallons is 79.49340000000001 liters.  
22.0 gallons is 83.2788 liters.  
23.0 gallons is 87.0642 liters.  
24.0 gallons is 90.84960000000001 liters.  
25.0 gallons is 94.635 liters.  
26.0 gallons is 98.4204 liters.  
27.0 gallons is 102.2058 liters.  
28.0 gallons is 105.9912 liters.  
29.0 gallons is 109.7766 liters.  
30.0 gallons is 113.562 liters.
```

AS PALAVRAS-CHAVE JAVA

Cinquenta palavras-chave estão definidas atualmente na linguagem Java (consulte a Tabela 1-1). Essas palavras-chave, combinadas com a sintaxe dos operadores e separadores, formam a base da linguagem. Elas não podem ser usadas como nomes de variável, classe ou método.

Tabela 1-1 As palavras-chave Java

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	extends	final
finally	float	for	goto	if	implements
import	instanceof	int	interface	long	native
new	package	private	protected	public	return
short	static	strictfp	super	switch	synchronized
this	throw	throws	transient	try	void
volatile	while				

As palavras-chave **const** e **goto** estão reservadas, mas não são usadas. Nos primórdios de Java, várias outras palavras-chave estavam reservadas para possível uso futuro. No entanto, a especificação atual só define as palavras-chave mostradas na Tabela 1-1.

Além das palavras-chave, Java reserva as palavras **true**, **false** e **null**, que são valores definidos pela linguagem. Você não pode usar essas palavras em nomes de variáveis, classes e assim por diante.

IDENTIFICADORES EM JAVA

Em Java, um identificador é o nome dado a um método, a uma variável ou a qualquer outro item definido pelo usuário. Os identificadores podem ter de um a vários caracteres. Os nomes de variável podem começar com qualquer letra do alfabeto, um sublinhado ou um cifrão. Em seguida pode haver uma letra, um dígito, um cifrão ou um sublinhado. O sublinhado pode ser usado para melhorar a legibilidade do nome da variável, como em **line_count**. As letras maiúsculas e minúsculas são diferentes, ou seja, para Java, **myvar** e **MyVar** são nomes diferentes. Aqui estão alguns exemplos de identificadores válidos:

Test	x	y2	maxLoad
\$up	_top	my_var	sample23

Lembre-se, você não pode iniciar um identificador com um dígito. Logo, **12x** é um identificador inválido, por exemplo.

Como explicado na seção anterior, você não pode usar nenhuma das palavras reservadas ou palavras-chave Java como nomes de identificador. Também não deve atribuir o nome de nenhum método padrão, como **println**, a um identificador. Além dessas duas restrições, a boa prática de programação preconiza o uso de nomes de identificador que refletem o significado ou o uso dos itens que estão sendo nomeados.

Pergunte ao especialista

P Você poderia fornecer alguma diretriz para seguirmos na seleção de bons nomes de variáveis?

R Sim. Geralmente, as variáveis devem receber nomes que descrevam seu significado ao serem usadas no programa. Por exemplo, se você estiver criando variáveis que conterão a largura e a altura de um retângulo, os nomes **largura** e **altura** são apropriados. É claro que, às vezes, o uso de uma única palavra não é adequado, sendo necessário um termo maior. Por exemplo, uma variável que conterá a concentração de algum material em partes por milhão pode se chamar **partesPorMilhão**, ou talvez abreviaremos isso para **partesPorMil**. Nesses exemplos, observe a capitalização. Após a primeira palavra, as palavras subsequentes são capitalizadas. Isso é chamado de *capitalização camel* e é um estilo comum entre programadores Java.

Embora nomes descritivos sejam muito importantes, às vezes o nome de uma variável é arbitrário e um nome descritivo não é aplicável. Normalmente isso ocorre com variáveis de controle de laço, variáveis que contêm um resultado temporário e variáveis usadas em exemplos curtos de programas, como os mostrados neste livro, que apenas demonstram um recurso da linguagem. Nesses casos, com frequência são empregadas letras individuais, como **x**, **i** ou **v**.

Verificação do progresso

1. Qual dessas é uma palavra-chave: **for**, **For** ou **FOR**?
2. Um identificador Java pode conter que tipo de caracteres?
3. Os identificadores **index21** e **Index21** são iguais?

AS BIBLIOTECAS DE CLASSES JAVA

Os exemplos de programa mostrados neste capítulo fazem uso de dois dos métodos internos da linguagem Java: **println()** e **print()**. Esses métodos são membros da classe **System**, uma classe predefinida por Java que é incluída automaticamente nos programas. De um modo geral, o ambiente Java depende de várias *bibliotecas de classes* internas que contêm muitos métodos internos para dar suporte a coisas como I/O, manipulação de strings, rede e uma interface gráfica de usuário. Portanto, Java como um todo é uma combinação da própria linguagem Java mais suas classes padrão. Como você verá, as bibliotecas de classes fornecem uma porção considerável da funcionalidade que vem com Java. Na verdade, faz parte de se tornar programador Java aprender a usar as classes Java padrão. No decorrer deste livro, vários elementos das classes e métodos de biblioteca padrão são descritos. No entanto, é preciso

Respostas:

1. A palavra-chave é **for**. Em Java, todas as palavras-chave são em minúsculas.
2. Letras, dígitos, sublinhado e \$.
3. Não. Java é sensível a maiúsculas/minúsculas.

entender que a biblioteca Java é muito grande. Ela contém muito mais recursos do que poderíamos descrever neste livro. É algo que você vai querer explorar melhor por conta própria ao desenvolver suas habilidades de programação com Java.

EXERCÍCIOS

1. Cite as três partes essenciais de um computador.
2. O que é código-fonte? E código-objeto?
3. Como fica o valor 14 em binário? Qual é o equivalente decimal ao número binário 1010 0110?
4. Como regra geral, um byte é composto por _____ bits.
5. O que é bytecode e por que ele é importante para o uso de Java em programação na Internet?
6. Quais são os três princípios básicos da programação orientada a objetos?
7. Onde os programas Java começam a ser executados?
8. O que é uma variável?
9. Quais dos nomes de variável a seguir são inválidos?
 - A. count
 - B. \$count
 - C. count27
 - D. 67count
10. Como se cria um comentário de linha única? E um comentário de várias linhas?
11. Mostre a forma geral da instrução **if**. Mostre também a do laço **for**.
12. Como se cria um bloco de código?
13. A gravidade da Lua é cerca de 17% a da Terra. Crie um programa que calcule seu peso na Lua.
14. Adapte o código da seção Tente isto 1-2 para que ele exiba uma tabela de conversões de polegadas para metros. Exiba 12 pés de conversões, polegada a polegada. Gere uma linha em branco a cada 12 polegadas. (Um metro é igual a aproximadamente 39,37 polegadas.)
15. Se você se enganar na digitação ao inserir seu programa, isso resultará em que tipo de erro?
16. O local onde inserimos uma instrução em uma linha é importante?
17. Verdadeiro ou falso:
 - A. Os comentários contêm informações importantes para o compilador.
 - B. Você pode ter comentários de várias linhas aninhados no formato `/*.../*.../*...*/`.
 - C. O sinal de igualdade = é usado para testar a igualdade em instruções **if**.

18. Cite dois dispositivos de I/O para computadores que não foram mencionados neste capítulo.
19. Por que os programadores não escrevem código na linguagem de máquina da CPU? Isto é, porque quase todos os programadores usam uma linguagem de alto nível como Java?
20. Qual é a diferença entre um compilador e um interpretador?
21. Como mencionado no texto, geralmente um byte é composto por 8 bits. Faça uma pequena pesquisa para determinar o que são os termos a seguir:
 - A. kilobyte
 - B. megabyte
 - C. gigabyte
 - D. terabyte
22. Diga se cada um dos símbolos ou palavras a seguir é uma palavra-chave Java, um operador, uma marca de pontuação ou nenhum desses:
 - A. 33
 - B. for
 - C. ;
 - D. int
 - E. {}
23. Dê um exemplo, diferente do exemplo do alimento usado no capítulo, de classificação hierárquica na qual cada classe herde todos os atributos de sua classe pai.
24. O que há de errado em cada um dos comandos a seguir?

```
| javac Example.class  
| java Example.class
```

25. Qual é a diferença entre o uso de `x = 3;` e `{ x = 3; }` como alvo de uma instrução `if`?
26. Use recuo, espaçamento e várias linhas para tornar o programa a seguir mais legível:

```
/* Este programa calcula e exibe a soma dos 10 primeiros inteiros  
positivos */ class SumFrom1To10{public static void main(String[]  
args){int sum,i;sum=0;for(i=1;i<=10;i++)sum=sum+i;System.out.  
println("The sum  
1 + 2+...+10 is "+sum);}}
```

27. Sugira nomes mais apropriados para a classe e as variáveis do programa abaixo:

```
/* Este programa converte Fahrenheit em Celsius. */  
class XXX {  
    public static void main(String[] args) {  
        double x, xx;
```

```
x = 62;  
xx = (x-32) * 5.0/9.0;  
  
System.out.println(x + " degrees Fahrenheit is " +  
                   xx + " degrees Celsius.");  
}  
}
```

28. Suponhamos que **x** fosse uma variável declarada como de tipo **int**. O que há de errado em cada uma das instruções a seguir?
- A. **x = 3.5;**
 - B. **if(x = 3) x = 4;**
 - C. **x = "34";**
29. Escreva um programa que exiba os 20 primeiros quadrados (1, 4, 9, 16, ..., 400), um por linha. Use um laço **for**.
30. Modifique sua resposta ao Exercício 29 para que ele exiba a soma dos 20 primeiros quadrados ($1 + 4 + 9 + 16 + \dots + 400$).
31. Modifique sua resposta ao Exercício 30 para que ele encontre e exiba a média dos 20 primeiros quadrados.

2

Introdução aos tipos de dados e operadores

PRINCIPAIS HABILIDADES E CONCEITOS

- Conhecer os tipos primitivos de Java
- Usar literais
- Inicializar variáveis
- Saber as regras de escopo de variáveis dentro de um método
- Usar os operadores aritméticos
- Usar os operadores relacionais e lógicos
- Entender os operadores de atribuição
- Usar atribuições abreviadas
- Entender a conversão de tipos em atribuições
- Usar uma coerção
- Entender a conversão de tipos em expressões

Na base de qualquer linguagem de programação estão seus tipos de dados e operadores, e Java não é exceção. Esses elementos definem os limites de uma linguagem e determinam o tipo de tarefas às quais ela pode ser aplicada. Felizmente, a linguagem Java dá suporte a um rico grupo tanto de tipos de dados quanto de operadores, o que a torna adequada a quase qualquer tipo de programação.

Os tipos de dados e operadores são um assunto extenso. Começaremos aqui com uma verificação dos tipos de dados básicos de Java e seus operadores mais usados. Também examinaremos com detalhes as variáveis e estudaremos as expressões.

POR QUE OS TIPOS DE DADOS SÃO IMPORTANTES

Os tipos de dados são particularmente importantes em Java porque essa é uma linguagem fortemente tipada. Ou seja, todas as operações têm a compatibilidade de seus tipos verificada pelo compilador. Operações inválidas não serão compiladas. Logo, a verificação minuciosa dos tipos ajuda a impedir a ocorrência de erros e melhora a confiabilidade. Para que seja possível fazer a verificação cuidadosa dos tipos, todas as variáveis, expressões e valores têm um tipo. Não há o conceito de

uma variável “sem tipo”, por exemplo. Além disso, o tipo de um valor determina as operações que podem ser executadas nele. Uma operação aplicada a um tipo pode não ser permitida em outro.

Tabela 2-1 Tipos de dados primitivos internos de Java

Tipo	Significado
boolean	Representa os valores verdadeiro/falso
byte	Inteiro de 8 bits
char	Caractere
double	Ponto flutuante de precisão dupla
float	Ponto flutuante de precisão simples
int	Inteiro
long	Inteiro longo
short	Inteiro curto

TIPOS PRIMITIVOS DA LINGUAGEM JAVA

Java contém duas categorias gerais de tipos de dados internos: orientados a objetos e não orientados a objetos. Os tipos orientados a objetos são definidos por classes, mas a discussão das classes será deixada para depois. Porém, na base de Java, temos oito tipos de dados primitivos (também chamados de elementares ou simples), que são mostrados na Tabela 2-1. O termo *primitivo* é usado aqui para indicar que esses tipos não são objetos no sentido da orientação a objetos, mas sim valores binários comuns. Esses tipos primitivos não são objetos devido a questões de eficiência.

Java especifica rigorosamente um intervalo e um comportamento para cada tipo primitivo, que todas as implementações da Máquina Virtual Java devem suportar. Devido ao requisito de portabilidade de Java, a linguagem é inflexível nesse aspecto. Por exemplo, um **int** é igual em todos os ambientes de execução. Isso permite que os programas sejam totalmente portáveis. Não precisamos reescrever um código para adequá-lo a uma plataforma específica. Embora a especificação rigorosa do intervalo dos tipos primitivos possa causar uma pequena piora no desempenho em alguns ambientes, ela é necessária para a obtenção da portabilidade.

Inteiros

Java define quatro tipos inteiros: **byte**, **short**, **int** e **long**, que são mostrados aqui:

Tipo	Tamanho em bits	Intervalo
byte	8	-128 a 127
short	16	32.768 a 32.767
int	32	-2.147.483.648 a 2.147.483.647
long	64	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

Como a tabela mostra, todos os tipos inteiros são valores de sinal positivo e negativo. Java não suporta inteiros sem sinal (somente positivos). Outras linguagens de computador suportam inteiros com e sem sinal. No entanto, os projetistas de Java decidiram que inteiros sem sinal eram desnecessários.

***Nota:** Tecnicamente, o sistema de tempo de execução Java pode usar qualquer tamanho para armazenar um tipo primitivo. Contudo, em todos os casos, os tipos devem agir como especificado.*

O tipo inteiro mais usado é **int**. Variáveis de tipo **int** costumam ser empregadas no controle de laços, na indexação de arrays e na execução de cálculos de inteiros para fins gerais.

Quando você precisar de um inteiro que tenha um intervalo maior do que o de **int**, use **long**. Por exemplo, aqui está um programa que calcula quantas polegadas há em um cubo com 1x1x1 milhas:

```
/*
  Calcula quantas polegadas cúbicas
  há em uma milha cúbica.
*/
class Inches {
    public static void main(String[] args) {
        long cubicInches;
        long inchesPerMile;

        // calcula quantas polegadas há em uma milha
        inchesPerMile = 5280 * 12;

        // calcula o número de polegadas cúbicas
        cubicInches = inchesPerMile * inchesPerMile * inchesPerMile;

        System.out.println("There are " + cubicInches +
                           " cubic inches in a cubic mile.");
    }
}
```

Lembre-se, uma milha tem 5.280 pés. Portanto, para calcularmos o número de polegadas cúbicas existente em uma milha cúbica, primeiro o número de polegadas existente em uma milha é obtido e então o valor é usado no cálculo do volume. Esta é a saída do programa:

```
| There are 254358061056000 cubic inches in a cubic mile.
```

É claro que o resultado não poderia ser mantido em uma variável **int**.

O menor tipo inteiro é **byte**. Variáveis de tipo **byte** são especialmente úteis no trabalho com dados binários brutos que podem não ser diretamente compatíveis com outros tipos internos Java. O tipo **short** cria um inteiro curto. Variáveis de tipo **short**

são apropriadas quando queremos economizar memória e não precisamos do intervalo maior oferecido por **int**.

Pergunte ao especialista

P Você diz que há quatro tipos de inteiros: **int**, **short**, **long** e **byte**. No entanto, ouvi falar que **char** também pode ser categorizado como um tipo inteiro em Java. Pode explicar?

R A especificação formal de Java define uma categoria de tipo chamada tipos integrais, que inclui **byte**, **short**, **int**, **long** e **char**. Eles são chamados de tipos integrais porque todos contêm valores binários inteiros. No entanto, a finalidade dos quatro primeiros é representar quantidades inteiras numéricas. A finalidade de **char** é representar caracteres. Logo, os usos principais de **char** e os dos outros tipos integrais são basicamente diferentes. Devido às diferenças, o tipo **char** é tratado separadamente neste livro.

Tipos de ponto flutuante

Como explicado no Capítulo 1, os tipos de ponto flutuante podem representar números que têm componentes fracionários. Há duas espécies de tipos de ponto flutuante. Elas são **float** e **double**, que representam números de precisão simples e dupla, respectivamente. O tipo **float** tem 32 bits e o tipo **double** tem 64. As diferenças significam que o maior literal **float** tem aproximadamente $3,4 \times 10^{38}$ de tamanho e o maior literal **double** tem cerca de $1,8 \times 10^{308}$.

Dos dois, **double** é o mais usado, porque todas as funções matemáticas da biblioteca de classes Java usam valores **double**. Por exemplo, o método **sqrt()** (que é definido pela classe padrão **Math**) retorna um valor **double** que é a raiz quadrada de seu argumento **double**. Abaixo, **sqrt()** é usado para calcular o comprimento da hipotenusa, dados os comprimentos dos dois lados opostos:

```
/*
  Usa o teorema de Pitágoras para
  encontrar o comprimento da hipotenusa
  dados os comprimentos dos dois lados
  opostos.
*/
class Hypotenuse {
    public static void main(String[] args) {
        double side1, side2, hypot;

        side1 = 3;
        side2 = 4;                                — Observe como sqrt() é chamado. Ele é precedido
                                                    pelo nome da classe da qual é membro.
        hypot = Math.sqrt(side1*side1 + side2*side2);

        System.out.println("Hypotenuse is " + hypot);
    }
}
```

A saída do programa é mostrada a seguir:

```
| Hypotenuse is 5.0
```

Outra coisa sobre o exemplo anterior: como mencionado, `sqrt()` é membro da classe padrão **Math**. Observe como `sqrt()` é chamado; é precedido pelo nome **Math**. Isso é semelhante à maneira como **System.out** precede `println()`. Embora nem todos os métodos padrão sejam chamados com a especificação do nome de sua classe antes, vários o são.

Caracteres

Em Java, os caracteres não são valores de 8 bits como em muitas outras linguagens de computador. Em vez disso, Java usa caracteres de 16 bits. A razão dessa diferença é que Java dá suporte a caracteres *Unicode*. O *Unicode* define um conjunto de caracteres que pode representar todos os caracteres encontrados em todos os idiomas humanos. Ele foi projetado originalmente como um valor de 16 bits e Java refletiu esse fato dando ao **char** 16 bits de tamanho. Em Java, **char** é um tipo de 16 bits sem sinal com um intervalo que vai de 0 a 65.536. O conjunto de caracteres ASCII de 8 bits padrão é um subconjunto do Unicode e vai de 0 a 127. Logo, os caracteres ASCII ainda são caracteres Java válidos. (ASCII é a abreviatura de American Standard Code for Information Interchange.)

Uma variável de caractere pode receber um valor pela inserção do caractere entre aspas simples. Por exemplo, este código atribui à variável **ch** a letra X:

```
| char ch;  
| ch = 'X';
```

Você pode exibir um valor **char** usando a instrução `println()`. Por exemplo, a linha seguinte exibe o valor de **ch**:

```
| System.out.println("This is ch: " + ch);
```

Como **char** é um tipo de 16 bits sem sinal, podemos tratar aritmeticamente uma variável **char** de muitas maneiras. Por exemplo, considere o programa a seguir:

```
// Variáveis de caracteres podem ser tratadas como inteiros.  
class CharArithDemo {  
    public static void main(String[] args) {  
        char ch;  
  
        ch = 'X';  
        System.out.println("ch contains " + ch);  
  
        ch++; // incrementa ch ←———— Um char pode ser incrementado.  
        System.out.println("ch is now " + ch);  
  
        ch = 90; // dá a ch o valor Z ←———— Um char pode receber um valor inteiro.  
        System.out.println("ch is now " + ch);  
    }  
}
```

A saída gerada por esse programa é mostrada aqui:

```
ch contains X
ch is now Y
ch is now Z
```

No programa, primeiro é dado a **ch** o valor X. Em seguida, **ch** é incrementado. Isso resulta em **ch** contendo Y, o próximo caractere na sequência ASCII (e Unicode). Depois, **ch** recebe o valor 90, que é o valor ASCII (e Unicode) correspondente à letra Z. Como o conjunto de caracteres ASCII ocupa os primeiros 127 valores do conjunto de caracteres Unicode, todos os “velhos truques” que os programadores usam com caracteres de outras linguagens também funcionarão em Java.

Pergunte ao especialista

P Porque Java usa Unicode?

R Java foi projetada para uso mundial. Logo, tem de usar um conjunto de caracteres que possa representar os idiomas do mundo todo. O Unicode é o conjunto de caracteres padrão projetado especialmente para esse fim. É claro que o uso do Unicode é ineficiente para idiomas como inglês, alemão, espanhol ou francês, cujos caracteres podem ser armazenados em 8 bits. Mas esse é o preço a ser pago pela portabilidade global.

O tipo booleano

O tipo **boolean** representa os valores verdadeiro/falso. Java define os valores verdadeiro e falso usando as palavras reservadas **true** e **false**. Logo, uma variável ou expressão de tipo **boolean** terá um desses dois valores.

Aqui está um programa que demonstra o tipo **boolean**:

```
// Demonstra valores booleanos.
class BoolDemo {
    public static void main(String[] args) {
        boolean b;

        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);

        // um valor booleano pode controlar a instrução if
        if(b) System.out.println("This is executed.");

        b = false;
        if(b) System.out.println("This is not executed.");

        // o resultado de um operador relacional é um valor booleano
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

A saída gerada por esse programa é mostrada abaixo:

```
b is false
b is true
This is executed.
10 > 9 is true
```

Três fatos interessantes se destacam nesse programa. Em primeiro lugar, como você pode ver, quando um valor **boolean** é exibido por **println()**, a palavra “true” ou “false” é usada. Em segundo lugar, o valor de uma variável **boolean** é suficiente para controlar a instrução **if**. Não há necessidade de escrever uma instrução **if** como esta:

```
| if(b == true) ...
```

Em terceiro lugar, o resultado de um operador relacional, como **<**, é um valor **boolean**. Portanto, a expressão **10 > 9** exibe o valor “true”. Além disso, o conjunto de parênteses adicional delimitando **10 > 9** é necessário porque o operador **+** tem *precedência* maior do que **>**. Quando um operador tem precedência maior do que o outro, ele é avaliado antes deste em uma expressão.

Verificação do progresso

1. Quais são os tipos inteiros Java?
2. O que é Unicode?
3. Que valores uma variável **boolean** pode ter?

TENTE ISTO 2-1 Qual é a distância do relâmpago?

Sound.java

Neste projeto, você criará um programa que calcula a que distância, em pés, um ouvinte está da queda de um relâmpago. O som viaja a aproximadamente 1.100 pés por segundo pelo ar. Logo, conhecer o intervalo entre o momento em que você viu um relâmpago e o momento em que o som o alcançou lhe permitirá calcular a distância do relâmpago. Para este projeto, considere que o intervalo seja de 7,2 segundos.

Respostas:

1. Os tipos inteiros Java são **byte**, **short**, **int** e **long**.
2. Unicode é um conjunto de caracteres internacional e multilíngue.
3. As variáveis de tipo **boolean** podem ser **true** ou **false**.

PASSO A PASSO

1. Crie um novo arquivo chamado **Sound.java**.
2. Para calcular a distância, você terá que usar valores de ponto flutuante. Por quê? Porque o intervalo de tempo, 7,2, tem um componente fracionário. Embora pudéssemos usar um valor de tipo **float**, usaremos **double** no exemplo.
3. Para calcular a distância, você multiplicará 7,2 por 1.100. Em seguida, atribuirá esse valor a uma variável.
4. Por fim, exibirá o resultado.
5. Aqui está o programa **Sound.java** inteiro:

```
/*
Tente isto 2-1

Calcule a distância da queda de um raio
cujo som leve 7,2 segundos
para alcançá-lo.

*/
class Sound {
    public static void main(String[] args) {
        double distance;

        distance = 7.2 * 1100;

        System.out.println("The lightning is approximately " + distance +
+
                    " feet away.");
    }
}
```

6. Compile e execute o programa. O resultado a seguir será exibido:

| The lightning is approximately 7920.0 feet away.

7. Desafio extra: você pode calcular a distância de um objeto grande, como uma parede de pedra, medindo o eco. Por exemplo, se você bater palmas e medir quanto tempo leva para ouvir o eco, saberá o tempo total que o som leva para ir e voltar. A divisão desse valor por dois gera o tempo que o som leva para se propagar em uma direção. Então, você poderá usar esse valor para calcular a distância do objeto. Modifique o programa anterior para que ele calcule a distância, supondo que o intervalo de tempo seja igual ao de um eco.

LITERAIS

Em Java, os *literais* são valores fixos representados em sua forma legível por humanos. Por exemplo, o número 100 é um literal. Normalmente os literais também são chamados de *constantes*. Quase sempre, os literais, e sua aplicação, são tão intuitivos

que eles foram usados de alguma forma por todos os exemplos de programa anteriores. Agora chegou a hora de serem explicados formalmente.

Os literais Java podem ser de qualquer um dos tipos de dados primitivos. A maneira como cada literal é representado depende de seu tipo. Como explicado anteriormente, constantes de caracteres são delimitadas por aspas simples. Por exemplo, "a" e "%" são constantes de caracteres.

Os literais inteiros são especificados como números sem componentes fracionários. Por exemplo, 10 e -100 são literais inteiros. Os literais de ponto flutuante requerem o uso do ponto decimal seguido pelo componente fracionário do número. Por exemplo, 11,123 é um literal de ponto flutuante. Java também permite o uso de notação científica para números de ponto flutuante. Para usá-la, especifique a mantissa, depois um E ou um e e então o expoente (que deve ser um inteiro). Por exemplo, 1,234E2 representa o valor 123,4 e 1,234E-2 representa o valor 0,01234.

Por padrão, os literais inteiros são de tipo **int**. Se quiser especificar um literal **long**, acrescente um l ou L. Por exemplo, 12 é um **int**, mas 12L é um **long**.

Também é padrão os literais de ponto flutuante serem de tipo **double**. Para especificar um literal **float**, acrescente um F ou f à constante. Por exemplo, 10,19F é de tipo **float**.

Embora os literais inteiros criem um valor **int** por padrão, eles podem ser atribuídos a variáveis de tipo **char**, **byte** ou **short**, contanto que o valor atribuído possa ser representado pelo tipo de destino. Um literal inteiro sempre pode ser atribuído a uma variável **long**.

A partir do JDK 7, é permitido embutir um ou mais sublinhados em um literal inteiro ou de ponto flutuante. Isso pode facilitar a leitura de valores compostos por muitos dígitos. Quando o literal é compilado, os sublinhados são simplesmente descartados. Aqui está um exemplo:

```
| 123_45_1234
```

Essa linha especifica o valor 123.451.234. O uso de sublinhados é particularmente útil na codificação de coisas como números de peças, identificações de clientes e códigos de status que normalmente são criados como uma combinação de subgrupos de dígitos.

Literais hexadecimais, octais e binários

Em programação, às vezes é mais fácil usar um sistema numérico baseado em 8 ou 16 em vez de 10. O sistema numérico baseado em 8 se chama *octal* e usa os dígitos de 0 a 7. No sistema octal, o número 10 é igual ao 8 do sistema decimal. O sistema numérico de base 16 se chama *hexadecimal* e usa os dígitos de 0 a 9 mais as letras A a F (ou a a f), que representam 10, 11, 12, 13, 14 e 15. Por exemplo, o número hexadecimal 10 é o 16 do sistema decimal. Devido à frequência com que esses dois sistemas numéricos são usados, Java permite a especificação de literais inteiros em hexadecimal ou octal em vez de decimal. Um literal hexadecimal deve começar com **0x** ou **0X** (um zero seguido por um x ou X). Um literal octal começa com um zero. Aqui estão alguns exemplos:

```
| hex = 0xFF; // 255 em decimal
| oct = 011; // 9 em decimal
```

Java também permite o uso de literais de ponto flutuante hexadecimais, mas raramente eles são usados.

A partir do JDK 7, é possível especificar um literal inteiro com o uso de binários. Para fazer isso, use um **0b** ou **0B** antes do número binário. Por exemplo, este número especifica o valor 12 em binário: **0b1100**.

Tabela 2-2 Sequências de escape de caracteres

Sequência de escape	Descrição
\'	Aspas simples
\\"	Aspas duplas
\\\	Barra invertida
\r	Retorno de carro
\n	Nova linha
\f	Avanço de página
\t	Tabulação horizontal
\b	Retrocesso
\ddd	Constante octal (onde <i>ddd</i> é uma constante octal)
\xxxxx	Constante hexadecimal (onde <i>xxxx</i> é uma constante hexadecimal)

Sequências de escape de caracteres

A inserção de constantes de caracteres entre aspas simples funciona para a maioria dos caracteres imprimíveis, mas alguns caracteres, como o retorno de carro, impõem um problema especial quando um editor de texto é usado. Além disso, outros caracteres específicos, como as aspas simples e duplas, têm um significado especial em Java, logo, você não pode usá-los diretamente. É por isso que Java fornece *sequências de escape* especiais, às vezes chamadas de constantes de caracteres de barra invertida, mostradas na Tabela 2-2. Essas sequências são usadas no lugar dos caracteres que elas representam.

Por exemplo, esta linha atribui a **ch** o caractere de tabulação:

```
| ch = '\t';
```

O próximo exemplo atribui uma aspa simples a **ch**:

```
| ch = '\'';
```

Literais de strings

Java dá suporte a outro tipo de literal: o string. Um *string* é um conjunto de caracteres inserido em aspas duplas. Por exemplo,

```
| "this is a test"
```

é um string. Você viu exemplos de strings em muitas das instruções **println()** dos exemplos de programa anteriores.

Além dos caracteres comuns, um literal de string também pode conter uma ou mais das sequências de escape que acabamos de descrever. Por exemplo, considere o programa a seguir. Ele usa as sequências de escape `\n` e `\t`.

```
// Demonstra sequências de escape em strings.
class StrDemo {
    public static void main(String[] args) {
        System.out.println("First line\nSecond line");
        System.out.println("A\tB\tC");
        System.out.println("D\tE\tF");
    }
}
```

A saída é mostrada aqui:

```
First line
Second line
A      B      C
D      E      F
```

Observe como a sequência de escape `\n` é usada para gerar uma nova linha. Você não precisa usar várias instruções `println()` para obter uma saída de várias linhas. Apenas incorpore `\n` a um string mais longo nos pontos onde deseja que as novas linhas ocorram.

Verificação do progresso

1. Qual é o tipo do literal 10? E o do literal 10,0?
2. Como podemos especificar um literal **long**?
3. “x” é um string ou um literal de caractere?

Pergunte ao especialista

P Um string composto por um único caractere é o mesmo que um literal de caractere? Por exemplo, “k” é o mesmo que ‘k’?

R Não. Você não deve confundir strings com caracteres. Um literal de caractere representa uma única letra de tipo **char**. Um string contendo apenas uma letra continua sendo um string. Embora os strings sejam compostos por caracteres, eles não são do mesmo tipo.

UM EXAME MAIS DETALHADO DAS VARIÁVEIS

As variáveis foram introduzidas no Capítulo 1. Aqui, vamos examiná-las mais detalhadamente. Como você aprendeu, as variáveis são declaradas com o uso da seguinte forma de instrução,

tipo nome-var;

Respostas:

1. O literal 10 é um **int**, e o 10,0 é um **double**.
2. Um literal **long** é especificado com a inclusão do sufixo **L** ou **l**. Por exemplo, 100L.
3. O literal “x” é um string.

onde *tipo* é o tipo de dado da variável e *nome-var* é seu nome. Você pode declarar uma variável de qualquer tipo válido, inclusive os tipos simples que acabei de descrever. Quando declarar uma variável, estará criando uma instância de seu tipo. Logo, os recursos de uma variável são determinados por seu tipo. Por exemplo, uma variável de tipo **boolean** pode ser usada para armazenar valores verdadeiro/falso, mas não valores de ponto flutuante. Além disso, o tipo de uma variável não pode mudar durante seu tempo de vida. Uma variável **int** não pode virar uma variável **char**, por exemplo.

Em Java, todas as variáveis devem ser declaradas antes de seu uso. Isso é necessário porque o compilador tem que saber que tipo de dado uma variável contém antes de poder compilar apropriadamente qualquer instrução que use a variável. Também permite que Java execute uma rigorosa verificação de tipos.

Inicializando uma variável

Em geral, devemos dar um valor à variável antes de usá-la. Uma maneira de dar um valor a uma variável é por uma instrução de atribuição, como já vimos. Outra é dando um valor inicial quando ela é declarada. Para fazer isso, coloque um sinal de igualdade e o valor que está sendo atribuído após o nome da variável. A forma geral de inicialização é mostrada aqui:

tipo var = valor;

Nessa linha, *valor* é o valor dado a *var* quando *var* é criada. O valor deve ser compatível com o tipo especificado. Veja alguns exemplos:

```
int count = 10; // dá a count um valor inicial igual a 10
char ch = 'X'; // inicializa ch com a letra X
float f = 1.2F; // f é inicializada com 1,2
```

Ao declarar duas ou mais variáveis do mesmo tipo usando uma lista separada por vírgulas, você pode dar um valor inicial a uma ou mais dessas variáveis. Por exemplo:

```
int a, b = 8, c = 19, d; // b e c têm inicializações
```

Nesse caso, só **b** e **c** são inicializadas.

Inicialização dinâmica

Embora os exemplos anteriores só tenham usado constantes como inicializadores, Java permite que as variáveis sejam inicializadas dinamicamente, com o uso de qualquer expressão válida no momento em que a variável é declarada. Por exemplo, aqui está um programa curto que calcula o volume de um cilindro dado o raio de sua base e sua altura:

```
// Demonstra a inicialização dinâmica.
class DynInit {
    public static void main(String[] args) {
        double radius = 4, height = 5;
        double volume = 3.1416 * radius * radius * height; ←
        System.out.println("Volume is " + volume);
    }
}
```

volume é inicializada dinamicamente
no tempo de execução.

Nesse exemplo, três variáveis locais – **radius**, **height** e **volume** – são declaradas. As duas primeiras, **radius** e **height**, são inicializadas por constantes. No entanto, **volume** é inicializada dinamicamente com o volume do cilindro. O ponto-chave aqui é que a expressão de inicialização pode usar qualquer elemento válido no momento da inicialização, inclusive chamadas a métodos, a outras variáveis ou a literais.

ESCOPO E O TEMPO DE VIDA DAS VARIÁVEIS

Até agora, todas as variáveis que usamos foram declaradas no início do método **main()**. Porém, Java permite que as variáveis sejam declaradas dentro de qualquer bloco. Como explicado no Capítulo 1, um bloco começa com uma chave de abertura e termina com uma chave de fechamento. O bloco define um *escopo*. Logo, sempre que você iniciar um novo bloco, estará criando um novo escopo. Um escopo determina que objetos estarão visíveis para outras partes de seu programa. Também determina o tempo de vida desses objetos.

Outras linguagens de computador definem duas categorias gerais de escopos: global e local. Embora suportadas, essas não são as melhores maneiras de categorizar os escopos em Java. Os escopos mais importantes em Java são os definidos por uma classe e os definidos por um método. Uma discussão sobre o escopo das classes (e as variáveis declaradas dentro dele) será deixada para depois, quando as classes forem descritas no livro. Por enquanto, examinaremos apenas os escopos definidos por ou dentro de um método.

O escopo definido por um método começa com sua chave de abertura. No entanto, se esse método tiver parâmetros, eles também estarão incluídos dentro do escopo do método.

Como regra geral, as variáveis declaradas dentro de um escopo não podem ser vistas (isto é, acessadas) por um código definido fora desse escopo. Logo, quando você declarar uma variável dentro de um escopo, estará localizando essa variável e protegendo-a contra modificação e/ou acesso não autorizado. Na verdade, as regras de escopo fornecem a base do encapsulamento.

Os escopos podem ser aninhados. Por exemplo, sempre que você criar um bloco de código, estará criando um novo escopo aninhado. Quando isso ocorre, o escopo externo engloba o escopo interno. Ou seja, os objetos declarados no escopo externo poderão ser vistos por um código que estiver dentro do escopo interno. No entanto, o inverso não é verdadeiro. Objetos declarados dentro do escopo interno não podem ser vistos fora dele.

Para entender o efeito dos escopos aninhados, considere o programa a seguir:

```
// Demonstra o escopo de bloco.
class ScopeDemo {
    public static void main(String[] args) {
        int x; // conhecida pelo código dentro de main

        x = 10;
        if(x == 10) { // inicia novo escopo

            int y = 20; // conhecida apenas neste bloco

            // tanto x quanto y são conhecidas aqui.

            System.out.println("x and y: " + x + " " + y);
        }
    }
}
```

```

        x = y * 2;
    }
    // y = 100; // Erro! y não é conhecida aqui ← Aqui, y está fora de seu escopo.

    // x ainda é conhecida aqui.
    System.out.println("x is " + x);
}
}

```

Como os comentários indicam, a variável **x** é declarada no início do escopo de **main()** e pode ser acessada por qualquer código subsequente desse método. Dentro do bloco **if**, **y** é declarada. Já que um bloco define um escopo, **y** só pode ser vista por códigos desse bloco. É por isso que, fora de seu bloco, a linha **y = 100;** é desativada por um comentário. Se você remover o símbolo de comentário, um erro de compilação ocorrerá, porque **y** não pode ser vista fora de seu bloco. Dentro do bloco **if**, **x** pode ser usada porque o código de um bloco (isto é, de um escopo aninhado) tem acesso às variáveis declaradas por um escopo externo.

Dentro de um bloco, as variáveis podem ser declaradas em qualquer ponto, mas só são válidas após serem declaradas. Portanto, se você definir uma variável no início de um método, ela estará disponível para todo o código desse método. Inversamente, se declarar uma variável no fim de um bloco, ela não terá utilidade, porque nenhum código poderá acessá-la.

Aqui está outro ponto que deve ser lembrado: as variáveis são criadas quando alcançamos seu escopo, e destruídas quando saímos dele. Ou seja, uma variável não manterá seu valor quando tiver saído do escopo. Logo, as variáveis declaradas dentro de um método não manterão seus valores entre chamadas a esse método. Além disso, uma variável declarada dentro de um bloco perderá seu valor após o bloco ser deixado. Portanto, o tempo de vida de uma variável está confinado ao seu escopo.

Se a declaração de variável incluir um inicializador, essa variável será reinicializada sempre que entrarmos no bloco em que ela é declarada. Por exemplo, considere este programa:

```

// Demonstra o tempo de vida de uma variável.
class VarInitDemo {
    public static void main(String[] args) {
        int x;

        for(x = 0; x < 3; x++) {
            int y = -1; // y será inicializada sempre que entrarmos no bloco
            System.out.println("y is: " + y); // essa linha sempre exibe -1
            y = 100;
            System.out.println("y is now: " + y);
        }
    }
}

```

A saída gerada pelo programa é mostrada abaixo:

```

y is: -1
y is now: 100
y is: -1

```

```
y is now: 100
y is: -1
y is now: 100
```

Como você pode ver, `y` é reinicializada com `-1` sempre que entramos no laço `for`. Ainda que depois ela receba o valor `100`, esse valor é perdido.

Há uma peculiaridade nas regras de escopo Java que deve surpreendê-lo: embora os blocos possam ser aninhados, dentro de um método nenhuma variável declarada em um escopo interno pode ter o mesmo nome de uma variável declarada por um escopo externo. Por exemplo, o programa a seguir, que tenta declarar duas variáveis separadas com o mesmo nome, não será compilado.

```
/*
Este programa tenta declarar uma variável
em um escopo interno com o mesmo nome de uma
definida em um escopo externo.

*** O programa não será compilado. ***
*/
class NestVar {
    public static void main(String[] args) {
        int count; ←

        for(count = 0; count < 10; count = count+1) {
            System.out.println("This is count: " + count);

            int count; // inválido!!! ←———— Não pode declarar count novamente porque
                       // ela já foi declarada.

            for(count = 0; count < 2; count++)
                System.out.println("This program is in error!");
        }
    }
}
```

Em outras linguagens (principalmente C/C++), não há restrições para os nomes dados a variáveis declaradas em um escopo interno. Assim, em C/C++ a declaração de `count` dentro do bloco do laço `for` externo é perfeitamente válida, e esse tipo de declaração oculta a variável externa. Os projetistas de Java acharam que essa ocultação de nome poderia levar facilmente a erros de programação e não a permitiram.

Verificação do progresso

1. O que é um escopo? Como um escopo pode ser criado?
2. Onde em um bloco as variáveis podem ser declaradas?
3. Em um bloco, quando uma variável é criada? E quando é destruída?

Respostas:

1. Um escopo define a visibilidade e o tempo de vida de um objeto. Um bloco define um escopo.
2. Uma variável pode ser declarada em qualquer ponto dentro de um bloco.
3. Dentro de um bloco, uma variável é criada quando sua declaração é encontrada. Ela é destruída quando saímos do bloco.

OPERADORES

Java fornece um ambiente rico em operadores. Um *operador* é um símbolo que solicita ao compilador que execute uma operação matemática ou lógica específica ou algum outro tipo de operação. Java tem quatro classes gerais de operadores: aritmético, bitwise, relacional e lógico. Também define alguns operadores adicionais que tratam certas situações especiais. Este capítulo examinará os operadores aritméticos, relacionais e lógicos. Também examinaremos o operador de atribuição. O operador bitwise e outros operadores especiais serão examinados posteriormente.

OPERADORES ARITMÉTICOS

Um conjunto básico de operadores aritméticos foi introduzido no Capítulo 1. Este é o conjunto completo:

Operador	Significado
+	Adição (também mais unário)
-	Subtração (também menos unário)
*	Multiplicação
/	Divisão
%	Módulo
++	Incremento
--	Decremento

Os operadores +, -, * e / funcionam em Java da mesma maneira que em qualquer outra linguagem de computador (ou em álgebra). Eles podem ser aplicados a qualquer tipo de dado numérico interno. Também podem ser usados em objetos de tipo **char**.

Embora as ações dos operadores aritméticos sejam conhecidas por todos os leitores, algumas situações especiais pedem explicação. Primeiro, lembre-se de que quando / é aplicado a um inteiro, o resto gerado é truncado; por exemplo, 10/3 será igual a 3 na divisão de inteiros. Você pode obter o resto dessa divisão usando o operador de módulo %. Ele gera o resto de uma divisão de inteiros. Por exemplo, 10 % 3 é igual a 1. Em Java, o operador % pode ser aplicado a tipos inteiros e de ponto flutuante. Logo, 10,0 % 3,0 também é igual a 1. O programa a seguir demonstra o operador de módulo.

```
// Demonstra o operador %.
class ModDemo {
    public static void main(String[] args) {
        int iresult, irem;
        double dresult, drem;

        iresult = 10 / 3;
        irem = 10 % 3;

        dresult = 10.0 / 3.0;
        drem = 10.0 % 3.0;
```

```
    System.out.println("Result and remainder of 10 / 3: " +
                       iresult + " " + irem);
    System.out.println("Result and remainder of 10.0 / 3.0: " +
                       dresult + " " + drem);
}
```

A saída do programa é mostrada aqui:

```
|Result and remainder of 10 / 3: 3 1
|Result and remainder of 10.0 / 3.0: 3.333333333333335 1.0
```

Como você pode ver, o operador % gera um resto igual a 1 para operações de tipos inteiros e de ponto flutuante.

Incremento e decremento

Introduzidos no Capítulo 1, ++ e -- são os operadores Java de incremento e decremento. Como veremos, eles têm algumas propriedades especiais que os tornam muito interessantes. Comecemos examinando exatamente o que os operadores de incremento e decremento fazem.

O operador de incremento adiciona 1 a seu operando, e o de decremento subtrai

1. Logo,

```
|x = x + 1;
```

é o mesmo que

```
|x++;
```

e

```
|x = x - 1;
```

é o mesmo que

```
|x--;
```

Tanto o operador de incremento quanto o de decremento podem preceder (prefixar) ou vir após (posfixar) o operando. Por exemplo,

```
|x = x + 1;
```

pode ser escrito como

```
|++x; // forma prefixada
```

ou como

```
|x++; // forma pós-fixada
```

No exemplo anterior, não há diferença se o incremento é aplicado como um prefixo ou um posíxo. No entanto, quando um incremento ou decremento é usado como parte de uma expressão maior, há uma diferença importante. Quando um operador de incremento ou decremento precede seu operando, Java executa a operação correspondente antes de obter o valor do operando a ser usado pelo resto da expressão. Se

o operador vier após seu operando, Java obterá o valor do operando antes de ele ser incrementado ou decrementado. Considere o seguinte:

```
|x = 10;
|y = ++x;
```

Nesse caso, **y** será configurado com 11. No entanto, se o código for escrito como

```
|x = 10;
|y = x++;
```

então **y** será configurado com 10. Nos dois casos, **x** é configurado com 11; a diferença é quando isso ocorre. Em expressões aritméticas complicadas, há vantagens significativas em podermos controlar quando a operação de incremento ou decremento deve ocorrer.

OPERADORES RELACIONAIS E LÓGICOS

Nos termos *operador relacional* e *operador lógico*, *relacional* se refere aos relacionamentos que os valores podem ter uns com os outros, e *lógico* se refere às maneiras como os valores verdadeiro e falso podem estar conectados. Já que os operadores relacionais produzem resultados verdadeiros ou falsos, com frequência trabalham com os operadores lógicos. Portanto, eles serão discutidos juntos aqui.

Os operadores relacionais foram introduzidos no Capítulo 1. Por conveniência, vamos mostrá-los novamente:

Operador	Significado
==	Igual a
!=	Diferente de
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a

Os operadores lógicos são mostrados abaixo:

Operador	Significado
&	AND
	OR
^	XOR (exclusive OR)
	OR de curto-circuito
&&	AND de curto-circuito
!	NOT

O resultado dos operadores relacionais e lógicos é um valor **boolean**.

Em Java, podemos comparar todos os objetos para ver se são iguais ou diferentes com o uso de == e !=. No entanto, os operadores de comparação <, >, <= ou >=

só podem ser aplicados aos tipos que dão suporte a um relacionamento sequencial. Logo, os operadores relacionais podem ser aplicados a todos os tipos numéricos e ao tipo **char**. Porém, valores de tipo **boolean** só podem ser comparados quanto à igualdade ou diferença, já que os valores **true** e **false** não são sequenciais. Por exemplo, **true > false** não tem significado em Java.

Quanto aos operadores lógicos, os operandos devem ser de tipo **boolean** e o resultado de uma operação lógica é de tipo **boolean**. Os operadores lógicos **&**, **|**, **^** e **!** dão suporte às operações lógicas básicas AND, OR, XOR e NOT, de acordo com a tabela-verdade a seguir:

p	q	p & q	p q	p ^ q	!p
Falso	Falso	Falso	Falso	Falso	Verdadeiro
Verdadeiro	Falso	Falso	Verdadeiro	Verdadeiro	Falso
Falso	Verdadeiro	Falso	Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Verdadeiro	Verdadeiro	Verdadeiro	Falso	Falso

Como a tabela mostra, o resultado de uma operação exclusive OR é verdadeiro quando exatamente um e apenas um operando é verdadeiro.

Aqui está um programa que demonstra vários dos operadores relacionais e lógicos:

```
// Demonstra os operadores relacionais e lógicos.
class RelLogOps {
    public static void main(String[] args) {
        int i, j;
        boolean b1, b2;

        i = 10;
        j = 11;
        if(i < j) System.out.println("i < j");
        if(i <= j) System.out.println("i <= j");
        if(i != j) System.out.println("i != j");
        if(i == j) System.out.println("this won't execute");
        if(i >= j) System.out.println("this won't execute");
        if(i > j) System.out.println("this won't execute");

        b1 = true;
        b2 = false;
        if(b1 & b2) System.out.println("this won't execute");
        if(!(b1 & b2)) System.out.println("!(b1 & b2) is true");
        if(b1 | b2) System.out.println("b1 | b2 is true");
        if(b1 ^ b2) System.out.println("b1 ^ b2 is true");
    }
}
```

A saída do programa é mostrada abaixo:

```
i < j
i <= j
i != j
```

```
| ! (b1 & b2) is true
| b1 | b2 is true
| b1 ^ b2 is true
```

OPERADORES LÓGICOS DE CURTO-CIRCUITO

Java fornece versões especiais de *curto-circuito* de seus operadores lógicos AND e OR que podem ser usadas para produzir código mais eficiente. Para entender o porquê, considere o seguinte: em uma operação AND, se o primeiro operando for falso, o resultado será falso não importando o valor do segundo operando. Em uma operação OR, se o primeiro operando for verdadeiro, o resultado da operação será verdadeiro não importando o valor do segundo operando. Logo, nesses dois casos, não há necessidade de avaliar o segundo operando. Quando não avaliamos o segundo operando, economizamos tempo e um código mais eficiente é produzido.

O operador AND de curto-circuito é **&&**, e o operador OR de curto-circuito é **||**. Seus equivalentes comuns são **&** e **|**, respectivamente. A única diferença entre as versões comum e de curto-circuito é que a versão comum sempre avalia cada operando e a versão de curto-circuito só avalia o segundo operando quando necessário.

Aqui está um programa que demonstra o operador AND de curto-circuito. O programa determina se o valor de **d** é um fator de **n**. Ele faz isso executando uma operação de módulo. Se o resto de **n / d** for zero, então **d** é um fator. No entanto, já que a operação de módulo envolve uma divisão, a versão de curto-circuito de AND é usada para impedir a ocorrência de um erro de divisão por zero.

```
// Demonstra os operadores de curto-circuito.
class SCops {
    public static void main(String[] args) {
        int n, d, q;

        n = 10;
        d = 2;
        if(d != 0 && (n % d) == 0)
            System.out.println(d + " is a factor of " + n);

        d = 0; // configura d com zero

        // Já que d é igual a zero, o segundo operando não é avaliado.
        if(d != 0 && (n % d) == 0) ← O operador de curto-
            System.out.println(d + " is a factor of " + n); ← -circuito impede uma
                                                               divisão por zero.

        /* Tente a mesma coisa sem o operador de curto-circuito.
         * Isso causará um erro de divisão por zero.
         */
        if(d != 0 & (n % d) == 0) ← Agora as duas expressões
            System.out.println(d + " is a factor of " + n);
    }
}
```

Para impedir uma divisão por zero, primeiro a instrução **if** verifica se **d** é igual a zero. Se for, o operador AND de curto-círcuito será interrompido nesse ponto e não executará a operação de módulo. Portanto, no primeiro teste, **d** é igual a 2 e a operação de módulo é executada. O segundo teste falha porque **d** é configurado com zero, e a operação de módulo é ignorada, o que evita um erro de divisão por zero. Para concluir, o operador AND comum é usado. Isso faz os dois operandos serem avaliados, o que leva a um erro de tempo de execução quando ocorre a divisão por zero.

Uma última coisa: a especificação formal de Java chama os operadores de curto-círcuito de operadores *conditional-or* e *conditional-and*, mas normalmente é usado o termo “curto-círcuito”.

Verificação do progresso

1. O que faz o operador **%**? A que tipos ele pode ser aplicado?
2. Que tipo de valores podem ser usados como operandos dos operadores lógicos?
3. Um operador de curto-círcuito sempre avalia seus dois operandos?

Pergunte ao especialista

P Já que os operadores de curto-círcuito são, em alguns casos, mais eficientes do que seus equivalentes comuns, por que Java oferece os operadores AND e OR comuns?

R Em alguns casos, você pode querer que os dois operandos de uma operação AND ou OR sejam avaliados devido aos efeitos colaterais produzidos. Considere o seguinte:

```
// Os efeitos colaterais podem ser importantes.
class SideEffects {
    public static void main(String[] args) {
        int i;

        i = 0;

        /* Aqui, i é incrementada mesmo que
           a instrução if seja falsa. */
        if(false & (++i < 100))
            System.out.println("this won't be displayed");
        System.out.println("if statement executed: " + i); // exibe 1

        /* Nesse caso, i não é incrementada porque
           o operador de curto-círcuito ignora o incremento. */
    }
}
```

Respostas:

1. **%** é o operador de módulo, que retorna o resto de uma divisão de inteiros. Ele pode ser aplicado a todos os tipos numéricos.
2. Os operadores lógicos devem ter operandos de tipo **boolean**.
3. Não, um operador de curto-círcuito só avalia seu segundo operando se o resultado da operação não puder ser determinado apenas por seu primeiro operando.

```

    if(false && (++i < 100))
        System.out.println("this won't be displayed");
    System.out.println("if statement executed: " + i); // continua exibindo 1 !!
}
}

```

Como os comentários indicam, na primeira instrução **if**, **i** é incrementada sendo ou não a instrução bem-sucedida. No entanto, quando o operador de curto-círcuito é usado, a variável **i** não é incrementada quando o primeiro operando é falso. A lição aprendida aqui é a de que se seu código espera que o operando do lado direito de uma operação AND ou OR seja avaliado, você deve usar versões dessas operações Java que não sejam de curto-círcuito.

O OPERADOR DE ATRIBUIÇÃO

Você vem usando o operador de atribuição desde o Capítulo 1. Agora é hora de o examinarmos formalmente. O *operador de atribuição* é o sinal de igual simples, **=**. Esse operador funciona em Java do mesmo modo como em qualquer outra linguagem de computador. Ele tem esta forma geral:

var = expressão;

Aqui, o tipo de *var* deve ser compatível com o tipo de *expressão*.

O operador de atribuição tem uma propriedade interessante que talvez você não conheça: ele permite a criação de uma cadeia de atribuições. Por exemplo, considere este fragmento:

```

int x, y, z;
x = y = z = 100; // configura x, y e z com 100

```

Ele configura as variáveis **x**, **y** e **z** com 100 usando a mesma instrução. Isso funciona porque **=** é um operador que fornece o valor da expressão do lado direito. Logo, o valor de **z = 100** é 100, que é então atribuído a **y**, que por sua vez é atribuído a **x**. O uso de uma “cadeia de atribuição” é uma maneira fácil de configurar um grupo de variáveis com um valor comum.

ATRIBUIÇÕES ABREVIADAS

Java fornece operadores especiais de atribuição *abreviada* que simplificam a codificação de certas instruções de atribuição. Comecemos com um exemplo. A instrução de atribuição mostrada aqui

```
| x = x + 10;
```

pode ser escrita, com o uso da atribuição abreviada Java, como

```
| x += 10;
```

O par de operadores `+=` solicita ao compilador que atribua a `x` o valor de `x` mais 10. Veja outro exemplo. A instrução

```
| x = x - 100;
```

é igual a

```
| x -= 100;
```

As duas instruções atribuem a `x` o valor de `x` menos 100.

Essa atribuição abreviada funciona para todos os operadores binários em Java (isto é, os que requerem dois operandos). A forma geral da atribuição abreviada é

`var op = expressão;`

Logo, os operadores aritméticos e lógicos de atribuição abreviada são os seguintes:

<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>
<code>%=</code>	<code>&=</code>	<code> =</code>	<code>^=</code>

Como esses operadores combinam uma operação com uma atribuição, eles são formalmente chamados de *operadores de atribuição compostos*.

Os operadores de atribuição compostos fornecem duas vantagens. Em primeiro lugar, são mais compactos do que seus equivalentes “não abreviados”. Em segundo lugar, em alguns casos, um bytecode mais eficiente pode ser gerado. Portanto, é comum vermos os operadores de atribuição compostos sendo usados em programas Java escritos profissionalmente.

CONVERSÃO DE TIPOS EM ATRIBUIÇÕES

Em programação, é comum atribuir um tipo de variável a outro. Por exemplo, você poderia atribuir um valor `int` a uma variável `float`, como mostrado aqui:

```
int i;  
float f;  
  
i = 10;  
f = i; // atribui um int a um float
```

Quando tipos compatíveis são combinados em uma atribuição, o valor do lado direito é convertido automaticamente para o tipo do lado esquerdo. Logo, no fragmento anterior, o valor de `i` é convertido para um `float` e então atribuído a `f`. No entanto, devido à rigorosa verificação de tipos de Java, nem todos os tipos são compatíveis e, assim, nem todas as conversões de tipo são permitidas implicitamente. Por exemplo, `boolean` e `int` não são compatíveis.

Se um tipo de dado for atribuído a uma variável de outro tipo, uma *conversão de tipos automática* ocorrerá quando

- os dois tipos forem compatíveis;
- o tipo de destino for maior que o de origem.

Quando essas duas condições são atendidas, ocorre uma conversão ampliadora. Por exemplo, o tipo `int` é sempre suficientemente grande para conter todos os valores

byte válidos, e tanto **int** quanto **byte** são tipos inteiros, logo, uma conversão automática de **byte** para **int** pode ser aplicada.

Em conversões ampliadoras, os tipos numéricos, inclusive os tipos inteiro e de ponto flutuante, são compatíveis. Por exemplo, o programa a seguir é perfeitamente válido, já que a transformação de **long** em **double** é uma conversão ampliadora que é executada automaticamente.

```
// Demonstra a conversão automática de long para double.
class LtoD {
    public static void main(String[] args) {
        long longVar;
        double doubleVar;

        longVar = 100123285L;
        doubleVar = longVar; ← Conversão automática de long para double.

        System.out.println("longVar and doubleVar: " +
                           longVar + " " + doubleVar);
    }
}
```

Embora haja a conversão automática de **long** para **double**, não há conversão automática de **double** para **long**, já que essa não é uma conversão ampliadora. Logo, a versão a seguir do programa anterior é inválida.

```
// *** Esse programa não será compilado. ***
class DtoL {
    public static void main(String[] args) {
        long longVar;
        double doubleVar;

        doubleVar = 100123285.0;
        longVar = doubleVar; // Inválido!!! ← Não há conversão automática de double
                            para long.
        System.out.println("longVar and doubleVar: " +
                           longVar + " " + doubleVar);
    }
}
```

Não há conversões automáticas de tipos numéricos para **char** ou **boolean**. Além disso, **char** e **boolean** não são compatíveis. No entanto, um literal inteiro pode ser atribuído a **char**.

USANDO UMA COERÇÃO

Embora as conversões de tipos automáticas sejam úteis, elas não atendem todas as necessidades de programação, porque só se aplicam a conversões ampliadoras entre tipos compatíveis. Em todos os outros casos, você deve雇regar uma coerção (*cast*). A *coerção* é uma instrução dada ao compilador para a conversão de um tipo em outro. Logo, ela solicita uma conversão de tipos explícita. Uma coerção tem esta forma geral:

(tipo-destino) expressão

Aqui, *tipo-destino* indica o tipo para o qual queremos converter a expressão especificada. Por exemplo, se você quiser converter o tipo da expressão `x/y` para `int`, pode escrever

```
double x, y;
// ...
int z = (int) (x / y);
```

No exemplo, ainda que `x` e `y` sejam de tipo `double`, a coerção converterá o resultado da expressão para `int`. Os parênteses que delimitam `x/y` são necessários. Caso contrário, a coerção para `int` só seria aplicada a `x` e não ao resultado da divisão. Nesse caso, a coerção é necessária porque não há conversão automática de `double` para `int`.

Quando a coerção envolve uma *conversão redutora*, informações podem ser perdidas. Por exemplo, na coerção de um `long` para um `short`, informações serão perdidas se o valor de tipo `long` for maior do que o intervalo do tipo `short`, porque seus bits de ordem superior serão removidos. Quando um valor de ponto flutuante é convertido para um tipo inteiro, o componente fracionário também é perdido devido ao truncamento. Por exemplo, se o valor 1,23 for atribuído a um inteiro, o valor resultante será simplesmente 1. O componente 0,23 será perdido.

O programa a seguir demonstra algumas conversões de tipo que requerem coerção:

```
// Demonstra a coerção.
class CastDemo {
    public static void main(String[] args) {
        double x, y;
        byte b;
        int i;
        char ch;

        x = 10.0;
        y = 3.0;
        i = (int) (x / y); // faz a coerção de double para int
        System.out.println("Integer outcome of x / y: " + i);

        i = 100;
        b = (byte) i; // Não há perda de informações aqui. Um byte pode conter o valor 100.
        System.out.println("Value of b: " + b);

        i = 257;
        b = (byte) i; // Desta vez há perda de informações. Um byte não pode conter o valor 257.
        System.out.println("Value of b: " + b);

        b = 88; // código ASCII para X
        ch = (char) b; // Faz a coerção de byte para char.
        System.out.println("ch: " + ch);
    }
}
```

A saída do programa é mostrada aqui:

```
Integer outcome of x / y: 3
Value of b: 100
Value of b: 1
ch: X
```

No programa, a coerção de (**x / y**) para **int** resulta no truncamento do componente fracionário e informações são perdidas. Em seguida, não ocorre perda de informação quando **b** recebe o valor 100 porque um **byte** pode conter o valor 100. No entanto, quando é feita a tentativa de atribuir a **b** o valor 257, ocorre perda de informações porque 257 excede o valor máximo de um **byte**. Para concluir, nenhuma informação é perdida, mas uma coerção é necessária na atribuição de um valor **byte** a um **char**.

Verificação do progresso

1. O que é coerção?
2. Um **short** poder ser atribuído a um **int** sem coerção? E um **byte** a um **char**?
3. Como a instrução a seguir pode ser reescrita?

```
x = x + 23;
```

PRECEDÊNCIA DE OPERADORES

A Tabela 2-3 mostra a ordem de precedência de todos os operadores Java, da mais alta à mais baixa. Operadores que estão na mesma linha têm a mesma precedência. Essa tabela inclui vários operadores que serão discutidos posteriormente no livro. A precedência de um operador determina em que ponto ele será avaliado em uma ex-

Tabela 2-3 A precedência dos operadores Java

Mais alta						
++ (postfixo)	-- (postfixo)					
++ (prefixo)	-- (prefixo)	~	!	+ (unário)	- (unário)	<i>(coerção de tipo)</i>
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
 						
&&						
 						
?:						
=	<i>op=</i>					
Mais baixa						

Respostas:

1. Uma coerção é uma conversão explícita.
2. Sim. Não.
3. `x += 23;`

pressão. Um operador de precedência mais alta será avaliado antes de um operador de precedência mais baixa. Por exemplo, dada a expressão

$$10 - 4 * 2$$

o resultado será 2 e não 12, porque a multiplicação tem precedência mais alta do que a subtração. Exceto na atribuição, operadores com precedência igual são avaliados da esquerda para a direita. Uma cadeia de atribuições é avaliada da direita para a esquerda. Embora tecnicamente sejam chamados de delimitadores, se considerados como operadores, [], () e . terão a precedência mais alta.

TENTE ISTO 2-2 Exiba uma tabela-verdade para os operadores lógicos

`LogicalOpTable.java`

Neste projeto, você criará um programa para exibir a tabela-verdade dos operadores lógicos Java. As colunas da tabela devem ficar alinhadas. O projeto faz uso de vários recursos abordados neste capítulo, inclusive uma das sequências de escape Java e os operadores lógicos. Ele também ilustra as diferenças de precedência entre o operador aritmético + e os operadores lógicos.

PASSO A PASSO

1. Crie um novo arquivo chamado **LogicalOpTable.java**.
 2. A fim de assegurar que as colunas fiquem alinhadas, você usará a sequência de escape \t para embutir tabulações em cada string de saída. Por exemplo, esta instrução `println()` exibe o cabeçalho da tabela:
- ```
| System.out.println("P\tQ\tAND\tOR\tXOR\tNOT") ;
```
3. Cada linha subsequente da tabela usará tabulações para que o resultado de cada operação seja posicionado sob o título apropriado.
  4. Aqui está o programa **LogicalOpTable.java** inteiro. Insira-o agora.

```
// Tente isto 2-2: uma tabela-verdade para os operadores lógicos.
class LogicalOpTable {
 public static void main(String[] args) {

 boolean p, q;

 System.out.println("P\tQ\tAND\tOR\tXOR\tNOT") ;

 p = true; q = true;
 System.out.print(p + "\t" + q + "\t");
 System.out.print((p&q) + "\t" + (p|q) + "\t");
 System.out.println((p^q) + "\t" + (!p)) ;

 p = true; q = false;
 System.out.print(p + "\t" + q + "\t");
 System.out.print((p&q) + "\t" + (p|q) + "\t");
 System.out.println((p^q) + "\t" + (!p)) ;
 }
}
```

```

p = false; q = true;
System.out.print(p + "\t" + q +"\t");
System.out.print((p&q) + "\t" + (p|q) + "\t");
System.out.println((p^q) + "\t" + (!p));
}

p = false; q = false;
System.out.print(p + "\t" + q +"\t");
System.out.print((p&q) + "\t" + (p|q) + "\t");
System.out.println((p^q) + "\t" + (!p));
}
}

```

Observe os parênteses que delimitam as operações lógicas dentro das instruções **de exibição**. Eles são necessários devido à precedência dos operadores Java. O operador + tem precedência mais alta do que os operadores lógicos.

- Compile e execute o programa. A tabela a seguir será exibida.

| P     | Q     | AND   | OR    | XOR   | NOT   |
|-------|-------|-------|-------|-------|-------|
| true  | true  | true  | true  | false | false |
| true  | false | false | true  | true  | false |
| false | true  | false | true  | true  | true  |
| false | false | false | false | false | true  |

- Por conta própria, tente modificar o programa para que ele use e exiba uns e zeros em vez de true e false. Isso pode dar um pouco mais de trabalho do que o esperado!

## EXPRESSÕES

Os operadores, as variáveis e os literais são componentes das *expressões*. Quando uma expressão é encontrada em um programa, ela é avaliada. Você já deve estar compreendendo bem as expressões porque elas foram usadas nos programas anteriores. Além disso, as expressões Java são semelhantes às encontradas em álgebra. No entanto, alguns de seus aspectos serão discutidos agora.

### Conversão de tipos em expressões

Dentro de uma expressão, é possível usar dois ou mais tipos de dados diferentes, contanto que eles sejam compatíveis. Por exemplo, você pode usar **short** e **long** dentro de uma expressão porque os dois são tipos numéricos. Quando tipos de dados diferentes são usados em uma expressão, todos são convertidos para o mesmo tipo. Isso é feito com o uso das *regras de promoção de tipos* de Java.

Primeiro, todos os valores **char**, **byte** e **short** são promovidos a **int**. Em seguida, se um operando for **long**, a expressão inteira será promovida a **long**. Se um operando for **float**, a expressão inteira será promovida a **float**. Se algum dos operandos for **double**, o resultado será **double**.

É importante entender que as promoções de tipos só são aplicadas aos valores usados quando uma expressão é avaliada. Por exemplo, se o valor de uma variável

**byte** for promovido a **int** dentro de uma expressão, fora dela a variável continuará sendo **byte**. A promoção de tipos só afeta a avaliação de uma expressão.

No entanto, a promoção de tipos pode levar a resultados inesperados. Por exemplo, quando uma operação aritmética envolve dois valores **byte**, ocorre a seguinte sequência: primeiro, os operandos **byte** são promovidos a **int**. Depois ocorre a operação, gerando um resultado **int**. Logo, o resultado de uma operação que envolve dois valores **byte** será um **int**. Isso não era esperado. Considere o programa a seguir:

```
// O inesperado em uma promoção!
class PromDemo {
 public static void main(String[] args) {
 byte b;
 int i;
 b = 10; └────────── Não é necessária a coerção porque o resultado já é elevado a int.
 i = b * b; // Certo, não é necessária uma coerção
 b = 10; └────────── Aqui é necessária uma coerção para atribuir um int a um byte!
 b = (byte) (b * b); // coerção necessária!!
 System.out.println("i and b: " + i + " " + b);
 }
}
```

Mesmo parecendo errado, nenhuma coerção é necessária na atribuição de **b\*b** a **i**, porque **b** é promovido a **int** quando a expressão é avaliada. No entanto, quando você tentar atribuir **b\*b** a **b**, precisará de uma coerção – novamente para **byte**! Lembre-se disso se receber mensagens de erro inesperadas de incompatibilidade de tipos referentes a expressões que de outra forma estariam perfeitamente corretas.

Situações como essa também ocorrem em operações com **chars**. Por exemplo, no fragmento a seguir, a coerção novamente para **char** é necessária devido à promoção de **ch1** e **ch2** a **int** dentro da expressão:

```
char ch1 = 'a', ch2 = 'b';
ch1 = (char) (ch1 + ch2);
```

Sem a coerção, o resultado da soma de **ch1** e **ch2** seria de tipo **int**, que não pode ser atribuído a um **char**.

As coerções não são úteis apenas na conversão entre tipos em uma atribuição. Por exemplo, considere o programa abaixo. Ele usa uma coerção para **double** a fim de obter um componente fracionário de uma divisão que seria de inteiros.

```
// Usando uma coerção.
class UseCast {
 public static void main(String[] args) {
 int i;
 for(i = 0; i < 5; i++) {
```

```

 System.out.println(i + " / 3: " + i / 3);
 System.out.println(i + " / 3 with fractions: "
 + (double) i / 3);
 System.out.println();
 }
}
}

```

A saída do programa é mostrada aqui:

```

0 / 3: 0
0 / 3 with fractions: 0.0

1 / 3: 0
1 / 3 with fractions: 0.3333333333333333

2 / 3: 0
2 / 3 with fractions: 0.6666666666666666

3 / 3: 1
3 / 3 with fractions: 1.0

4 / 3: 1
4 / 3 with fractions: 1.3333333333333333

```

## Espaçamento e parênteses

Uma expressão em Java pode ter tabulações e espaços para torná-la mais legível. Por exemplo, as duas expressões a seguir são iguais, mas a segunda é mais fácil de ler:

```

x=10/y*(127/x);

x = 10 / y * (127/x);

```

Os parênteses aumentam a precedência das operações contidas dentro deles, como na álgebra. O uso de parênteses adicionais não causará erros nem retardará a execução da expressão. É recomendável o seu uso para que a ordem exata da avaliação fique mais clara, tanto para você quanto para as pessoas que precisarem entender seu programa posteriormente. Por exemplo, qual das duas expressões abaixo é mais fácil de ler?

```

x = y/3-34*temp+127;

x = (y/3) - (34*temp) + 127;

```

---

## EXERCÍCIOS

1. Por que Java especifica rigorosamente o intervalo e o comportamento de seus tipos primitivos?
2. Qual é o tipo de caractere usado em Java e em que ele é diferente do tipo de caractere usado por outras linguagens de programação?
3. Um valor **boolean** pode ter o valor que você quiser, já que qualquer valor diferente de zero é verdadeiro. Verdade ou mentira?
4. Dada esta saída,

|       |
|-------|
| One   |
| Two   |
| Three |

usando um único string, mostre a instrução **println()** que a produziu.

5. O que está errado neste fragmento?

```
for(i = 0; i < 10; i++) {
 int sum;

 sum = sum + i;
}
System.out.println("Sum is: " + sum);
```

6. Explique a diferença entre as formas prefixada e pós-fixada do operador de incremento.
7. Mostre como um AND de curto-circuito pode ser usado para impedir um erro de divisão por zero.
8. Em uma expressão, a que tipo são promovidos **byte** e **short**?
9. Em geral, quando uma coerção é necessária?
10. Escreva um programa que encontre todos os números primos entre 2 e 100.
11. O uso de parênteses adicionais afeta o desempenho do programa?
12. Um bloco define um escopo?
13. Em algumas linguagens, as variáveis podem conter valores de qualquer tipo. Por que Java não permite esse comportamento? Isto é, porque Java restringe as variáveis a conter valores de apenas um tipo, a saber, o tipo declarado para a variável?
14. Crie um programa que atribua o valor 50.000 a uma variável inteira **x**, atribua o valor de **x\*x** a uma variável inteira **y** e então exiba o valor de **y**. Obteve uma resposta estranha? Se sim, explique por quê.
15. No exemplo **BoolDemo**, aparece a linha de código a seguir:

```
| System.out.println("10 > 9 is " + (10 > 9));
```

O que seria exibido (se fosse caso) se os parênteses fossem removidos e a linha contivesse:

```
| System.out.println("10 > 9 is " + 10 > 9);
```

Explique sua resposta.

16. Quais das instruções de atribuição a seguir são válidas em Java? Explique o porquê para cada uma que não for válida.
- A. `int x = false;`
  - B. `int x = 3 > 4;`
  - C. `int x = (3 > 4);`
  - D. `int x = int y = 3;`
  - E. `int x = 3.14;`
  - F. `int x = 3.14L;`
  - G. `int x = 5,000,000;`
  - H. `int x = 5_000_000;`
  - I. `int x = '350';`
  - J. `int x = "350";`
  - K. `int x = '3';`
  - L. `boolean b = (boolean) 5;`
  - M. `byte b = (byte) 5;`
  - N. `double d = 1E3.5;`
  - O. `char c = '\\/';`
  - P. `char c = '\\\\';`
  - Q. `char c = 3;`
  - R. `char c = "3";`
17. Quais das expressões a seguir são válidas em Java? Se uma expressão não for válida, explique o porquê. Se for válida, forneça seu valor. Presuma que **x** é uma variável **int** de valor 5, **y** é uma variável **double** de valor 3.5 e **b** é uma variável **boolean** de valor **false**.
- A. `(3 + 4 / 5)/3`
  - B. `3 * 4 % 5 / 2 * 6`
  - C. `3 + x++`
  - D. `3 + ++x`
  - E. `0/0`
  - F. `y/x`
  - G. `'a' + 'b'`
  - H. `'a' + 'b' + "c"`
  - I. `"3" + 2 + 1`
  - J. `"3" + (2 + 1)`
  - K. `false < true`
  - L. `false == true`
  - M. `'c' == 99`
  - N. `(3+4 > 5) & (4=6) | b`
  - O. `!((3>4) | (5!=5)) & (3<(4*0))`
  - P. `!(3>4 | 5!=5&3<4*0)`

18. Suponha que **a**, **b** e **c** sejam variáveis **boolean**. Encontre um conjunto de valores para **a**, **b** e **c** que façam as expressões  $(a \& b \mid c)$  e  $(\neg a \mid \neg b \& c)$  serem verdadeiras.
19. Se **x** é uma variável de tipo **int** e seu valor é 5, qual será seu valor após a sequência de instruções a seguir ser executada?

```
| x += 4;
| x *= 2;
| x /= 3;
| x %= 4;
```

20. Se **x** fosse uma variável de tipo **boolean** e seu valor fosse **true**, qual seria seu valor após a sequência de instruções a seguir ser executada?

```
| x |= false;
| x &= true;
| x ^= true;
```

21. **Math.random()** é um método da biblioteca Java que encontra um valor **double** aleatório entre 0 e 1. Por exemplo, a instrução

```
| double x = Math.random();
```

atribui à variável **x** um **double** aleatório entre 0 e 1. Crie um programa que verifique se **Math.random()** funciona bem. Mais precisamente, escreva um programa que chame **Math.random()** 1.000 vezes para criar 1.000 valores, registrando quantos deles são maiores do que 0,5, e então exiba o resultado. Teoricamente seu programa deve exibir um número muito próximo de 500.

22. Escreva um programa que crie três variáveis **double** aleatórias **a**, **b** e **c** e atribua a elas valores entre 0 e 1 usando o método **Math.random()** mencionado no exercício anterior. Em seguida, ele deve executar todas as ações a seguir:
- Exibir os três valores.
  - Exibir “All are tiny” se todos os três valores forem menores do que 0,2.
  - Exibir “One is tiny” se exatamente um dos três valores for menor do que 0,2.

# 3

## Instruções de controle de programa

### PRINCIPAIS HABILIDADES E CONCEITOS

- Inserir caracteres a partir do teclado
- Saber a forma completa da instrução **if**
- Usar a instrução **switch**
- Saber a forma completa do laço **for**
- Usar o laço **while**
- Usar o laço **do-while**
- Usar **break** para sair de um laço
- Usar **break** como uma forma de goto
- Aplicar **continue**
- Aninhar laços

Neste capítulo, você aprenderá as instruções que controlam o fluxo de execução do programa. As instruções de controle de programa Java podem ser organizadas nas seguintes categorias:

- Instruções de seleção
- Instruções de iteração
- Instruções de salto

As instruções de seleção permitem que o programa selecione diferentes caminhos de execução. As instruções de iteração permitem que uma seção de código seja repetida. As instruções de salto transferem o controle do programa diretamente de um local para outro. As instruções de seleção Java são **if** e **switch**; as de iteração são **for**, **while** e **do-while**; e as instruções de salto são **break**, **continue** e **return**. Exceto por **return**, que será discutida no Capítulo 4, as outras instruções de controle, inclusive as instruções **if** e **for** sobre as quais você já teve uma pequena introdução, serão examinadas com detalhes aqui.

O capítulo não começará apresentando as instruções de controle e sim explicando como podemos fornecer algumas entradas simples a partir do teclado. Esse pequeno desvio permitirá que você comece criando programas interativos.

*Nota: O mecanismo Java de tratamento de exceções também pode afetar o fluxo de execução de programas. Ele será discutido no Capítulo 10.*

## CARACTERES DE ENTRADA DO TECLADO

Até o momento, os exemplos de programa deste livro exibiram informações *para* o usuário sem recebê-las *do* usuário. Logo, você tem usado a saída de console, mas não a entrada de console (teclado). A razão é principalmente porque muitos dos meios de entrada em Java dependem ou fazem uso de recursos que só serão discutidos posteriormente no livro. Além disso, vários programas e applets Java do mundo real são gráficos e baseados em janelas e não em console. Portanto, não será feito muito uso da entrada de console neste livro. Mas há um tipo de entrada de console que é relativamente fácil de usar: a leitura de um caractere a partir do teclado. Já que vários dos exemplos deste capítulo farão uso desse recurso, ele será discutido aqui.

Para ler um caractere a partir do teclado usaremos **System.in.read()**. **System.in** complementa **System.out**. É o objeto de entrada ligado ao teclado. O método **read()** espera até o usuário pressionar uma tecla e então retorna o resultado. O caractere é retornado como um inteiro, logo, deve ser convertido para um **char** para ser atribuído a uma variável **char**. Por padrão, a entrada de console usa um *buffer de linha*. Aqui, o termo *buffer* se refere a uma pequena parte da memória que é usada para armazenar os caracteres antes de serem lidos pelo programa. Nesse caso, o buffer armazena uma linha de texto completa. Já que a linha inteira está no buffer, você deve pressionar ENTER antes de qualquer caractere digitado ser enviado para o programa. A seguir, temos um programa que mostra como se dá a leitura de um caractere a partir do teclado.

```
// Lê um caractere do teclado.
class KbIn {
 public static void main(String[] args)
 throws java.io.IOException {

 char ch;

 System.out.print("Press a key followed by ENTER: ");

 ch = (char) System.in.read(); // obtém um char ← Lê um caractere do teclado.

 System.out.println("Your key is: " + ch);
 }
}
```

Aqui está um exemplo da execução:

```
| Press a key followed by ENTER: t
| Your key is: t
```

No programa, observe que **main( )** começa assim:

```
public static void main(String[] args)
 throws java.io.IOException {
```

Como **System.in.read( )** está sendo usado, o programa deve especificar a cláusula **throws java.io.IOException**. Essa linha é necessária para tratar erros de entrada. Ela faz parte do mecanismo de tratamento de exceções de Java, que é discutido no Capítulo 9. Por enquanto, não se preocupe com seu significado exato.

O fato de **System.in** usar um buffer de linha pode ser fonte de aborrecimentos. Quando pressionamos ENTER, uma sequência retorno de carro/alimentação de linha é inserida no fluxo de entrada. (Em alguns ambientes, só uma alimentação de linha é inserida.) Além disso, esses caracteres ficam pendentes no buffer de entrada até serem lidos. Logo, em alguns aplicativos, podemos ter de removê-los (lendo-os) antes da próxima operação de entrada. Veremos um exemplo posteriormente neste capítulo.

### Verificação do progresso

1. O que é **System.in**?
2. Como podemos ler um caractere digitado no teclado?

## A INSTRUÇÃO if

O Capítulo 1 introduziu a instrução **if**. Ela será examinada em detalhes agora. A forma completa da instrução **if** é

```
if(condição) instrução;
else instrução;
```

em que os alvos de **if** e **else** são instruções individuais. A cláusula **else** é opcional. Os alvos tanto de **if** quanto de **else** podem ser blocos de instruções. A forma geral de **if**, usando blocos de instruções, é

```
if(condição)
{
 sequência de instruções
}
else
{
 sequência de instruções
}
```

---

Respostas:

1. **System.in** é o objeto de entrada vinculado à entrada padrão, que geralmente é o teclado.
2. Para ler um caractere, chame **System.in.read( )**.

Se a expressão condicional for verdadeira, o alvo de **if** será executado; caso contrário, se houver, o alvo de **else** será executado. Nunca ambos serão executados. A expressão condicional que controla **if** deve produzir um resultado **boolean**. Para demonstrar **if** (e várias outras instruções de controle), criaremos e desenvolveremos um jogo de adivinhação computadorizado simples que seria apropriado para crianças. Na primeira versão do jogo, o programa pede ao jogador uma letra entre A e Z. Se o jogador pressionar a letra correta no teclado, o programa responderá exibindo a mensagem **\*\* Right \*\***. O código é mostrado abaixo:

```
// Adivinhe a letra do jogo.
class Guess {
 public static void main(String[] args)
 throws java.io.IOException {

 char ch, answer = 'K';

 System.out.println("I'm thinking of a letter between A and Z.");
 System.out.print("Can you guess it: ");

 ch = (char) System.in.read(); // lê um char no teclado

 if(ch == answer) System.out.println("** Right **");
 }
}
```

Esse programa interage com o jogador e então lê um caractere do teclado. Usando uma instrução **if**, ele compara o caractere com a resposta, que é K nesse caso. Se K for inserido, a mensagem será exibida. Quando você testar o programa, lembre-se de que o K deve ser inserido em maiúscula.

Para avançarmos um pouco mais no jogo de adivinhação, a próxima versão usa **else** para exibir uma mensagem quando a letra errada é escolhida.

```
// Adivinhe a letra do jogo, 2ª versão.
class Guess2 {
 public static void main(String[] args)
 throws java.io.IOException {

 char ch, answer = 'K';

 System.out.println("I'm thinking of a letter between A and Z.");
 System.out.print("Can you guess it: ");

 ch = (char) System.in.read(); // obtém um char

 if(ch == answer) System.out.println("** Right **");
 else System.out.println("...Sorry, you're wrong.");
 }
}
```

## Ifs ANINHADOS

Um **if aninhado** é uma instrução **if** que é alvo de outro **if** ou **else**. Os **ifs** aninhados são muito comuns em programação porque fornecem uma maneira de fazermos uma nova seleção baseada no resultado da seleção anterior. O importante a lembrar sobre **ifs** aninhados em Java é que uma instrução **else** será sempre referente à instrução **if** mais próxima que estiver dentro do mesmo bloco e ainda não estiver associada a um **else**. Aqui está um exemplo:

```
if(i == 10) {
 if(j < 20) a = b;
 if(k > 100) c = d;
 else a = c; // esse else é referente a if(k > 100)
}
else a = d; // esse else é referente a if(i == 10)
```

Como os comentários indicam, o **else** final não está associado a **if(j < 20)**, porque não está no mesmo bloco (ainda que esse seja o **if** mais próximo sem um **else**). Em vez disso, o **else** final está associado a **if(i == 10)**. O **else** interno é referente a **if(k > 100)**, porque esse é o **if** mais próximo dentro do mesmo bloco.

Você pode usar um **if** aninhado para melhorar ainda mais o jogo de adivinhação. Esse acréscimo fornece ao jogador uma explicação sobre um palpite errado.

```
// Adivinhe a letra do jogo, 3ª versão.
class Guess3 {
 public static void main(String[] args)
 throws java.io.IOException {

 char ch, answer = 'K';

 System.out.println("I'm thinking of a letter between A and Z.");
 System.out.print("Can you guess it: ");

 ch = (char) System.in.read(); // obtém um char

 if(ch == answer) System.out.println("** Right **");
 else {
 System.out.print("...Sorry, you're ");

 // um if aninhado
 if(ch < answer) System.out.println("too low");
 else System.out.println("too high");
 }
 }
}
```

Um exemplo da execução é mostrado aqui:

```
I'm thinking of a letter between A and Z.
Can you guess it: Z
...Sorry, you're too high
```

## A ESCADA if-else-if

Uma estrutura de programação comum que é baseada no **if** aninhado costuma ser chamada de *escada if-else-if*. Ela tem a seguinte aparência:

```
if(condição)
 instrução;
else if(condição)
 instrução;
else if(condição)
 instrução;
.
.
.
else
 instrução;
```

As expressões condicionais são avaliadas de cima para baixo. Assim que uma condição verdadeira é encontrada, a instrução associada a ela é executada e o resto da escada é ignorado. Se nenhuma das condições for verdadeira, a instrução **else** final será executada. Com frequência, o **else** final age como uma condição padrão, isto é, se todos os outros testes condicionais falharem, a última instrução **else** será executada. Se não houver um **else** final e todas as outras condições forem falsas, não ocorrerá ação alguma.

O programa a seguir demonstra a escada **if-else-if**:

```
// Demonstra uma escada if-else-if.
class Ladder {
 public static void main(String[] args) {
 int x;

 for(x=0; x<6; x++) {
 if(x==1)
 System.out.println("x is one");
 else if(x==2)
 System.out.println("x is two");
 else if(x==3)
 System.out.println("x is three");
 else if(x==4)
 System.out.println("x is four");
 else
 System.out.println("x is not between 1 and 4"); ← Esta é a instrução
 padrão.
 }
 }
}
```

O programa produz a saída abaixo:

```
x is not between 1 and 4
x is one
```

```

x is two
x is three
x is four
x is not between 1 and 4

```

Como você pode ver, o **else** padrão só é executado quando nenhuma das instruções **if** anteriores é bem-sucedida.

### Verificação do progresso

1. A condição que controla **if** deve ser de que tipo?
2. A que **if** um **else** está sempre associado?
3. O que é uma escada **if-else-if**?

## A INSTRUÇÃO switch

A segunda instrução de seleção Java é a **switch**. A instrução **switch** fornece uma ramificação com vários caminhos. Logo, ela permite que o programa faça uma seleção entre várias alternativas. Embora uma série de instruções **if** aninhadas possam executar testes com vários caminhos, em muitas situações **switch** é uma abordagem mais eficiente. Funciona desta forma: o valor de uma expressão é verificado sucessivamente em uma lista de constantes. Quando uma ocorrência é encontrada, a sequência de instruções associada a essa ocorrência é executada. A forma geral da instrução **switch** é

```

switch(expressão) {
 case constante1:
 sequência de instruções
 break;
 case constante2:
 statement sequence
 break;
 case constante3:
 sequência de instruções
 break;
 .
 .
}

```

Respostas:

1. A condição que controla **if** deve ser de tipo **boolean**.
2. Um **else** sempre está associado ao **if** mais próximo do mesmo bloco que ainda não estiver associado a um **else**.
3. Uma escada **if-else-if** é uma sequência de instruções **if-else** aninhadas.

```
 default:
 sequência de instruções
 }
```

Em versões de Java anteriores ao JDK 7, a *expressão* que controla **switch** deve ser de tipo **byte**, **short**, **int**, **char** ou uma enumeração. (As enumerações serão descritas no Capítulo 13.) A partir do JDK 7, a *expressão* também pode ser de tipo **String**. Ou seja, versões modernas de Java podem usar um **string** para controlar **switch**. (Essa técnica é demonstrada no Capítulo 5, quando **String** é descrito.) Com frequência, a expressão que controla **switch** é apenas uma variável e não uma expressão maior.

Cada valor especificado nas instruções **case** deve ser uma expressão de constante exclusiva (como um valor literal). Não são permitidos valores duplicados em **case**. O tipo de cada valor deve ser compatível com o tipo da expressão controladora.

A sequência de instruções **default** é executada quando nenhuma constante **case** coincide com a expressão. A instrução **default** é opcional: se não estiver presente, não ocorrerá ação alguma quando todas as comparações falharem. Quando uma ocorrência é encontrada, as instruções associadas a esse **case** são executadas até **break** ser alcançado ou, no caso de **default** ou do último **case**, até o fim de **switch** ser alcançado.

O programa a seguir demonstra **switch**:

```
// Demonstra switch.
class SwitchDemo {
 public static void main(String[] args) {
 int i;

 for(i=0; i<10; i++)
 switch(i) {
 case 0:
 System.out.println("i is zero");
 break;
 case 1:
 System.out.println("i is one");
 break;
 case 2:
 System.out.println("i is two");
 break;
 case 3:
 System.out.println("i is three");
 break;
 case 4:
 System.out.println("i is four");
 break;
 default:
 System.out.println("i is five or more");
 }
 }
}
```

A saída produzida por esse programa é mostrada aqui:

```
i is zero
i is one
i is two
i is three
i is four
i is five or more
```

Como você pode ver, a cada passagem pelo laço, as instruções associadas à constante **case** que corresponde a **i** são executadas. Todas as outras são ignoradas. Quando **i** é cinco ou maior, nenhuma instrução **case** apresenta correspondência, logo, a instrução **default** é executada.

Tecnicamente, a instrução **break** é opcional, embora seja usada na maioria das aplicações de **switch**. Quando encontrada dentro da sequência de instruções de um **case**, a instrução **break** faz o fluxo do programa sair da instrução **switch** e continuar na próxima instrução externa. No entanto, se uma instrução **break** não terminar a sequência de instruções associada a um **case**, tanto as instruções *pertencentes ao case certo quanto as posteriores* serão executadas até um **break** (ou o fim de **switch**) ser alcançado.

Por exemplo, estude o programa a seguir com cuidado. Antes de olhar a saída, consegue identificar o que será exibido?

```
// Demonstra switch sem instruções break.
class NoBreak {
 public static void main(String[] args) {
 int i;

 for(i=0; i<=5; i++) {
 switch(i) {
 case 0:
 System.out.println("i is less than one");
 case 1:
 System.out.println("i is less than two");
 case 2:
 System.out.println("i is less than three");
 case 3:
 System.out.println("i is less than four");
 case 4:
 System.out.println("i is less than five");
 }
 System.out.println();
 }
 }
}
```

Todas as instruções **case** são executadas.

Esse programa exibirá a saída abaixo:

```
i is less than one
i is less than two
i is less than three
i is less than four
i is less than five

i is less than two
i is less than three
i is less than four
i is less than five

i is less than three
i is less than four
i is less than five

i is less than four
i is less than five

i is less than five
```

Como o programa ilustra, a execução passará para o próximo **case** se não houver uma instrução **break** presente.

Você também pode ter **cases** vazios, como mostrado neste exemplo:

```
switch(i) {
 case 1:
 case 2:
 case 3: System.out.println("i is 1, 2 or 3");
 break;
 case 4: System.out.println("i is 4");
 break;
}
```

Nesse fragmento, se **i** tiver o valor 1, 2 ou 3, a primeira instrução **println( )** será executada. Se for igual a 4, a segunda instrução **println( )** será executada. O “empilhamento” de **cases**, como mostrado no exemplo, é comum quando vários **cases** compartilham o mesmo código.

## INSTRUÇÕES switch ANINHADAS

É possível um **switch** fazer parte da sequência de instruções de um **switch** externo. Isso é chamado de **switch aninhado**. Mesmo se as constantes **case** do **switch** interno e externo tiverem valores comuns, não ocorrerá conflito. Por exemplo, o fragmento de código a seguir é perfeitamente aceitável:

```
switch(ch1) {
 case 'A': System.out.println("This A is part of outer switch.");
 switch(ch2) {
 case 'A':
 System.out.println("This A is part of inner switch");
```

```

 break;
 case 'B': // ...
} // fim do switch interno
break;
case 'B': // ...

```

## Verificação do progresso

1. A expressão que controla **switch** pode ser de que tipo?
2. Quando a expressão de **switch** coincide com uma constante **case**, o que acontece?
3. O que acontece quando uma sequência **case** não terminar em **break**?

### TENTE ISTO 3-1 Comece a construção de um sistema de ajuda Java

`Help.java`

Este projeto constrói um sistema de ajuda simples que exibe a sintaxe das instruções de controle Java. No processo, ele mostra a instrução **switch** em ação. O programa exibe um menu contendo as instruções de controle e então espera que uma seja selecionada. Após a seleção, a sintaxe da instrução é exibida. Nessa primeira versão do programa, só há ajuda disponível para as instruções **if** e **switch**. As outras instruções de controle serão adicionadas em exemplos subsequentes.

#### PASSO A PASSO

1. Crie um arquivo chamado **Help.java**.
2. O programa começa exibindo o menu a seguir:

```

Help on:
 1. if
 2. switch
Choose one:

```

Para exibi-lo, você usará a sequência de instruções mostradas aqui:

```

System.out.println("Help on:");
System.out.println(" 1. if");

```

#### Respostas:

1. A expressão de **switch** pode ser de tipo **char**, **short**, **int**, **byte** ou uma enumeração. A partir do JDK 7, um **String** também pode ser usado.
2. Quando uma constante **case** coincidente é encontrada, a sequência de instruções associada a esse **case** é executada.
3. Se uma sequência **case** não terminar com **break**, a execução passará para a próxima sequência **case**, se existir uma.

```
| System.out.println(" 2. switch");
| System.out.print("Choose one: ");
```

3. Em seguida, o programa lerá a seleção do usuário chamando **System.in.read()**, como mostrado abaixo:

```
| choice = (char) System.in.read();
```

4. Uma vez que a seleção tiver sido lida, o programa usará a instrução **switch** mostrada a seguir para exibir a sintaxe da instrução selecionada.

```
switch(choice) {
 case '1':
 System.out.println("The if:\n");
 System.out.println("if(condition) statement;");
 System.out.println("else statement;");
 break;
 case '2':
 System.out.println("The switch:\n");
 System.out.println("switch(expression) {");
 System.out.println(" case constant:");
 System.out.println(" statement sequence");
 System.out.println(" break;");
 System.out.println(" // ...");
 System.out.println("}");
 break;
 default:
 System.out.print("Selection not found.");
}
```

Observe como a cláusula **default** captura escolhas inválidas. Por exemplo, se o usuário inserir 3, não haverá uma constante **case** correspondente, fazendo a sequência **default** ser executada.

5. Aqui está o programa **Help.java** inteiro:

```
/*
Tente isto 3-1

Um sistema de ajuda simples.
*/
class Help {
 public static void main(String[] args)
 throws java.io.IOException {
 char choice;

 System.out.println("Help on:");
 System.out.println(" 1. if");
 System.out.println(" 2. switch");
 System.out.print("Choose one: ");
 choice = (char) System.in.read();

 System.out.println("\n");

 switch(choice) {
```

```

 case '1':
 System.out.println("The if:\n");
 System.out.println("if(condition) statement;");
 System.out.println("else statement;");
 break;
 case '2':
 System.out.println("The switch:\n");
 System.out.println("switch(expression) { ");
 System.out.println(" case constant:");
 System.out.println(" statement sequence");
 System.out.println(" break; ");
 System.out.println(" // ...");
 System.out.println("} ");
 break;
 default:
 System.out.print("Selection not found.");
 }
 }
}
}

```

### 6. Veja um exemplo da execução.

```

Help on:
1. if
2. switch
Choose one: 1

The if:

if(condition) statement;
else statement;

```

## Pergunte ao especialista

**P** Sob que condições devo usar uma escada **if-else-if** em vez de um **switch** ao codificar uma ramificação com vários caminhos?

**R** Em geral, use uma escada **if-else-if** quando as condições que controlam o processo de seleção não dependerem de um único valor. Por exemplo, considere a sequência **if-else-if** a seguir:

```

if(x < 10) // ...
else if(y != 0) // ...
else if(!done) // ...

```

Essa sequência não pode ser recodificada com um **switch** porque todas as três condições envolvem variáveis diferentes – e tipos diferentes. Que variável controlaria o **switch**? Você também terá que usar uma escada **if-else-if** ao testar valores de ponto flutuante ou outros objetos que não sejam de tipos válidos em uma expressão **switch**.

## O LAÇO for

Você vem usando uma forma simples do laço **for** desde o Capítulo 1. Talvez fique surpreso ao ver como ele é poderoso e flexível. Examinemos o básico, começando com as formas mais tradicionais de **for**.

A forma geral do laço **for** para a repetição de uma única instrução é

for(*inicialização*; *condição*; *iteração*) *instrução*;

Para a repetição de um bloco, a forma geral é

for(*inicialização*; *condição*; *iteração*)  
{  
 *sequência de instruções*  
}

Geralmente, a *inicialização* é uma instrução de atribuição que configura o valor inicial da variável de controle de laço, a qual age como o contador que controla o laço. A *condição* é uma expressão booleana que determina se o laço será ou não repetido. A expressão de *iteração* define o valor segundo o qual a variável de controle de laço mudará sempre que o laço for repetido. Observe que essas três seções principais do laço devem ser separadas por ponto e vírgula. O laço **for** continuará a ser executado enquanto a condição for verdadeira. Quando a condição se tornar falsa, o laço terminará e a execução do programa será retomada na instrução posterior a ele.

O laço **for** é mais usado quando sabemos que um laço será executado um número predeterminado de vezes. Ele também é muito útil quando uma sequência de valores é requerida, já que com frequência sua variável de controle pode ser usada para produzir a sequência. Por exemplo, se quiséssemos exibir as raízes quadradas dos números entre 1 e 99, um laço **for** seria muito útil, como este programa ilustra.

```
// Exibe as raízes quadradas de 1 a 99.
class SqrRoot {
 public static void main(String[] args) {
 double num, sroot;

 for(num = 1.0; num < 100.0; num++) {
 sroot = Math.sqrt(num);
 System.out.println("Square root of " + num +
 " is " + sroot);
 }
 }
}
```

Aqui, a sequência de valores para os quais as raízes quadradas são obtidas é produzida pela variável de controle do laço **for**.

O laço **for** pode seguir em sentido positivo ou negativo e mudar a variável de controle de laço de acordo com qualquer valor. Por exemplo, o programa a seguir exibe os números 100 a -95 em decrementos de 5:

```
// Um laço for sendo executado em sentido negativo.
class DecrFor {
 public static void main(String[] args) {
 int x;

 for(x = 100; x > -100; x -= 5) ← A variável de controle de laço é sempre
 System.out.println(x); decrementada em 5 unidades.
 }
}
```

Um ponto importante sobre os laços **for** é que a expressão condicional é sempre testada no início do laço. Ou seja, o código de dentro do laço não será executado se a condição for falsa. Aqui está um exemplo:

```
for(count=10; count < 5; count++)
 x += count; // Essa instrução não será executada
```

Esse laço nunca será executado, porque sua variável de controle, **count**, é maior do que 5 quando entramos no laço pela primeira vez. Isso torna a expressão condicional, **count < 5**, falsa desde o início; logo, nem mesmo uma iteração ocorrerá no laço.

## ALGUMAS VARIAÇÕES DO LAÇO for

O laço **for** é uma das instruções mais versáteis da linguagem Java porque permite muitas variações. Uma das mais comuns é o uso de diversas variáveis de controle. Quando muitas variáveis de controle são usadas, as expressões de inicialização e iteração de cada variável são separadas por vírgulas. Veja um exemplo simples:

```
// Usa múltiplas variáveis de controle em um laço for.
class MultipleLoopVars {
 public static void main(String[] args) {
 int i, j;

 for(i=0, j=10; i < j; i++, j--) ← Observe as duas variáveis de controle de laço.
 System.out.println("i and j: " + i + " " + j);
 }
}
```

A saída do programa é mostrada aqui:

```
i and j: 0 10
i and j: 1 9
i and j: 2 8
i and j: 3 7
i and j: 4 6
```

Observe como as vírgulas separam as duas expressões de inicialização e as duas expressões de iteração. Quando o laço começa, tanto **i** quanto **j** são inicializadas na par-

te de inicialização. Sempre que o laço se repete, **i** é incrementada e **j** é decrementada. O laço termina quando **i** é igual ou maior que **j**. Em princípio, você pode ter qualquer número de variáveis de controle de laço, mas, na prática, mais de duas ou três tornam o laço **for** difícil de controlar.

Outra variação comum de **for** envolve a natureza da condição de controle do laço. Essa condição não precisa envolver a variável de controle de laço. Ela pode ser qualquer expressão booleana válida. No próximo exemplo, o laço continua a ser executado até o usuário digitar a letra S no teclado:

```
// Executa o laço até um S ser digitado.
class ForTest {
 public static void main(String[] args)
 throws java.io.IOException {

 int i;

 System.out.println("Press S to stop.");

 for(i = 0; (char) System.in.read() != 'S'; i++)
 System.out.println("Pass #" + i);
 }
}
```

## Partes ausentes

Algumas variações interessantes do laço **for** são criadas quando deixamos vazias partes da definição do laço. Em Java, podemos deixar algumas ou todas as partes referentes à inicialização, condição ou iteração do laço **for** em branco. Por exemplo, considere o programa a seguir:

```
// Partes de for podem estar vazias.
class Empty {
 public static void main(String[] args) {
 int i;

 for(i = 0; i < 10;) { ←————— A expressão de iteração está faltando.
 System.out.println("Pass #" + i);
 i++; // incrementa a variável de controle de laço
 }
 }
}
```

Aqui, a expressão de iteração de **for** está vazia. Em vez disso, a variável de controle **i** é incrementada dentro do corpo do laço. Ou seja, sempre que o laço é repetido, **i** é testada para vermos se é igual a 10, mas nenhuma outra ação ocorre. É claro que, como **i** é incrementada dentro do corpo do laço, este é executado normalmente, exibindo a saída abaixo:

```
Pass #0
Pass #1
Pass #2
```

```

Pass #3
Pass #4
Pass #5
Pass #6
Pass #7
Pass #8
Pass #9

```

No próximo exemplo, a parte de inicialização também é removida de **for**:

```

// Retira mais uma parte do laço for.
class Empty2 {
 public static void main(String[] args) {
 int i;
 ↓
 i = 0; // move a inicialização para fora do laço
 for(; i < 10;) {
 System.out.println("Pass #" + i);
 i++; // incrementa a variável de controle de laço
 }
 }
}

```

A expressão de inicialização  
é removida do laço.

Nessa versão, **i** é inicializada antes de o laço começar e não como parte de **for**. Normalmente, preferimos inicializar a variável de controle dentro de **for**. A inserção da inicialização fora do laço só costuma ocorrer quando o valor inicial é derivado de um processo cujo confinamento dentro da instrução **for** é inadequado.

## O laço infinito

Você pode criar um *laço infinito* (um laço que nunca termina) usando **for** se deixar a expressão condicional vazia. Por exemplo, o fragmento abaixo mostra como muitos programadores de Java criam um laço infinito:

```

for(;;) // laço intencionalmente infinito
{
 //...
}

```

Esse laço será executado infinitamente. Embora haja algumas tarefas de programação, como o processamento de comandos do sistema operacional, que precisam de um laço infinito, os “laços infinitos” são, em sua maioria, apenas laços com requisitos especiais de encerramento. Quase no fim deste capítulo você verá como interromper um laço desse tipo. (Dica: isso é feito com o uso da instrução **break**.)

## Laços sem corpo

Em Java, o corpo associado a um laço **for** (ou a qualquer outro laço) pode estar vazio. Isso ocorre porque uma *instrução nula* é sintaticamente válida. Laços sem corpo costumam ser úteis. Por exemplo, o programa abaixo usa um para somar os números de 1 a 5:

```

// O corpo de um laço pode estar vazio.
class Empty3 {
 public static void main(String[] args) {

```

```
int i;
int sum = 0;

// soma os números até 5
for(i = 1; i <= 5; sum += i++) ; ← Não há corpo neste laço!

System.out.println("Sum is " + sum);
}
```

A saída do programa é mostrada aqui:

```
| Sum is 15
```

Observe que o processo de soma é totalmente tratado dentro da instrução **for** e nenhum corpo é necessário. Preste atenção principalmente na expressão de iteração:

```
| sum += i++
```

Não se assuste com instruções assim. Elas são comuns em programas Java escritos profissionalmente e serão fáceis de entender se você as dividir em suas partes. Em outras palavras, essa instrução diz “atribua a **sum** o valor de **sum** mais **i** e depois incremente **i**”. Logo, seria o mesmo que esta sequência de instruções:

```
sum = sum + i;
i++;
```

## DECLARANDO VARIÁVEIS DE CONTROLE DE LAÇO DENTRO DA INSTRUÇÃO **for**

Geralmente a variável que controla um laço **for** só é necessária para fins do laço e não é usada em outro local. Quando for esse o caso, podemos declarar a variável dentro da parte de inicialização de **for**. Por exemplo, o programa a seguir calcula tanto a soma quanto o produto dos números de 1 a 5. Ele declara sua variável de controle de laço **i** dentro de **for**.

```
// Declara a variável de controle de laço dentro de for.
class ForVar {
 public static void main(String[] args) {
 int sum = 0;
 int product = 1;

 // calcula a soma e o produto dos números de 1 a 5
 for(int i = 1; i <= 5; i++) { ← A variável i é declarada
 sum += i; // i é conhecida em todo o laço dentro da instrução for.
 product *= i;
 }

 // mas i não é conhecida aqui

 System.out.println("Sum is " + sum);
```

```

 System.out.println("Product is " + product);
}
}

```

Quando você declarar uma variável dentro de um laço **for**, há algo importante a lembrar: o escopo dessa variável terminará quando terminar a instrução **for**. (Isto é, o escopo da variável é limitado ao laço **for**.) Fora do laço **for**, a variável deixará de existir. Portanto, no exemplo anterior, **i** não pode ser acessada fora do laço **for**. Se você tiver de usar a variável de controle de laço em outro lugar de seu programa, não poderá declará-la dentro de **for**.

Antes de prosseguir, se quiser, teste suas próprias variações do laço **for**. Como verá, é um laço fascinante.

## O LAÇO for MELHORADO

Há outro tipo de laço **for**, chamado *for melhorado*. O **for** melhorado fornece uma maneira otimizada de percorrer o conteúdo de um conjunto de objetos, como em um array. Ele será discutido no Capítulo 5, após os arrays serem introduzidos.

### Verificação do progresso

1. Partes de um laço **for** podem estar vazias?
2. Mostre como criar um laço infinito usando **for**.
3. Qual é o escopo de uma variável declarada dentro de uma instrução **for**?

## O LAÇO while

Outro laço suportado em Java é o **while**. A forma geral do laço **while** é

`while(condição) instrução;`

em que *instrução* pode ser uma única instrução ou um bloco de instruções, e *condição* define a condição que controla o laço e pode ser qualquer expressão booleana válida. O laço se repete enquanto a condição é verdadeira. Quando a condição se torna falsa, o controle do programa passa para a linha imediatamente posterior ao laço.

Aqui está um exemplo simples em que um **while** é usado para exibir o alfabeto:

```

// Demonstra o laço while.
class WhileDemo {
 public static void main(String[] args) {
 char ch;

```

**Respostas:**

1. Sim. Todas as três partes de **for** – inicialização, condição e iteração – podem estar vazias.
2. `for(;;)`
3. O escopo de uma variável declarada dentro de **for** fica limitado ao laço. Fora do laço, ela é desconhecida.

```
// exibe o alfabeto usando um laço while
ch = 'a';
while(ch <= 'z') {
 System.out.print(ch);
 ch++;
}
}
```

No exemplo, **ch** é inicializada com a letra a. A cada passagem pelo laço, **ch** é exibida e então incrementada. Esse processo continua até **ch** ser maior do que z.

Como no laço **for**, **while** verifica a expressão condicional no início do laço, ou seja, o código do laço pode não ser executado. Isso elimina a necessidade de execução de um teste separado antes do laço. O programa abaixo ilustra essa característica do laço **while**. Ele calcula as potências inteiras de 2, de 0 a 9.

```
// Calcula as potências inteiras de 2.
class Power {
 public static void main(String[] args) {
 int e;
 int result;

 for(int i=0; i < 10; i++) {
 result = 1;
 e = i;
 while(e > 0) {
 result *= 2;
 e--;
 }

 System.out.println("2 to the " + i +
 " power is " + result);
 }
 }
}
```

A saída do programa é mostrada aqui:

```
2 to the 0 power is 1
2 to the 1 power is 2
2 to the 2 power is 4
2 to the 3 power is 8
2 to the 4 power is 16
2 to the 5 power is 32
2 to the 6 power is 64
2 to the 7 power is 128
2 to the 8 power is 256
2 to the 9 power is 512
```

Observe que o laço **while** só é executado quando *e* é maior do que 0. Logo, quando *e* é igual a zero, como ocorre na primeira iteração do laço **for**, o laço **while** é ignorado.

### Pergunte ao especialista

**P** Dada a flexibilidade inerente a todos os laços Java, que critérios devo usar ao selecionar um laço? Isto é, como escolher o laço certo para uma tarefa específica?

**R** Use um laço **for** para executar um número conhecido de iterações. Use **do-while** quando precisar de um laço que execute sempre pelo menos uma iteração. O laço **while** é mais adequado quando o laço é repetido até alguma condição ser falsa.

## O LAÇO do-while

O último dos laços Java é **do-while**. Diferentemente dos laços **for** e **while**, em que a condição é testada no início do laço, o laço **do-while** verifica sua condição no fim do laço. Ou seja, um laço **do-while** será sempre executado pelo menos uma vez. A forma geral do laço **do-while** é

```
do {
 instruções;
} while(condição);
```

Embora as chaves não sejam necessárias quando há apenas uma instrução presente, elas são usadas com frequência para melhorar a legibilidade da estrutura **do-while**, evitando, assim, confusão com **while**. O laço **do-while** é executado enquanto a expressão condicional for verdadeira.

O programa a seguir demonstra **do-while** entrando em laço até o usuário inserir a letra q:

```
// Demonstra o laço do-while.
class DWDemo {
 public static void main(String[] args)
 throws java.io.IOException {

 char ch;

 do {
 System.out.print("Press a key followed by ENTER: ");
 ch = (char) System.in.read(); // obtém um char
 } while(ch != 'q');
 }
}
```

Observe que o corpo do laço **do-while** pede um pressionamento de tecla e então lê a tecla pressionada. Esse caractere é comparado com a letra q na expressão condicional. Se a tecla pressionada não for um q, o laço será repetido. Já que essa condição

é testada no fim do laço, o corpo do laço será executado pelo menos uma vez, o que assegura que o usuário seja solicitado a pressionar uma tecla.

Usando o laço **do-while**, podemos melhorar ainda mais o programa do jogo de adivinhação que vimos anteriormente neste capítulo. Dessa vez, o programa entrará em laço até você adivinhar a letra.

```
// Adivinhe a letra do jogo, 4ª versão.
class Guess4 {
 public static void main(String[] args)
 throws java.io.IOException {

 char ch, ignore, answer = 'K';

 do {
 System.out.println("I'm thinking of a letter between A and Z.");
 System.out.print("Can you guess it: ");

 // lê um caractere
 ch = (char) System.in.read();

 // descarta qualquer outro caractere do buffer de entrada
 do {
 ignore = (char) System.in.read();
 } while(ignore != '\n');

 if(ch == answer) System.out.println("** Right **");
 else {
 System.out.print("...Sorry, you're ");
 if(ch < answer) System.out.println("too low");
 else System.out.println("too high");
 System.out.println("Try again!\n");
 }
 } while(answer != ch);
 }
}
```

Aqui está um exemplo da execução:

```
I'm thinking of a letter between A and Z.
Can you guess it: A
...Sorry, you're too low
Try again!

I'm thinking of a letter between A and Z.
Can you guess it: Z
...Sorry, you're too high
Try again!

I'm thinking of a letter between A and Z.
Can you guess it: K
** Right **
```

Observe outra coisa interessante nesse programa. Há dois laços **do-while**. O primeiro entra em laço até o usuário adivinhar a letra. Sua operação e significado devem estar claros. O segundo laço **do-while**, mostrado novamente aqui, pede alguma explicação:

```
// descarta qualquer outro caractere do buffer de entrada
do {
 ignore = (char) System.in.read();
} while(ignore != '\n');
```

Como explicado anteriormente, a entrada do console fica em um buffer de linha – você tem que pressionar ENTER antes de os caracteres serem enviados. O pressionamento de ENTER faz uma sequência retorno de carro/alimentação de linha (nova linha) ser gerada. Esses caracteres ficam pendentes no buffer de entrada. Além disso, se você digitar mais de uma tecla antes de pressionar ENTER, elas também ficarão no buffer de entrada. O laço em questão descarta esses caracteres continuando a ler a entrada até o fim da linha ser alcançado. Se eles não fossem descartados, seriam enviados para o programa como palpites e não é o que queremos. (Para ver o efeito disso, tente remover o laço **do-while** interno.) No Capítulo 11, após você ter aprendido mais sobre Java, serão descritas outras maneiras de tratar entradas do console em um nível mais alto. No entanto, o uso de **read()** aqui dá uma ideia de como a base do sistema de I/O Java opera. E também mostra outro exemplo dos laços Java em ação.

## Verificação do progresso

1. Qual é a principal diferença entre os laços **while** e **do-while**?
2. A condição que controla **while** pode ser de qualquer tipo. Verdadeiro ou falso?

### TENTE ISTO 3-2 Melhore o sistema de ajuda Java

`Help2.java`

Este projeto expande o sistema de ajuda Java que foi criado na seção Tente isto 3-1. A versão atual adiciona a sintaxe dos laços **for**, **while** e **do-while**. Também mostra como um laço **do-while** pode ser usado na verificação da seleção do usuário no menu, entrando em laço até uma resposta válida ser inserida. Observe como a instrução **switch** facilita muito a inclusão de seleções.

#### PASSO A PASSO

1. Copie **Help.java** em um novo arquivo chamado **Help2.java**.

#### Respostas:

1. O laço **while** verifica sua condição no início do laço. O laço **do-while** verifica sua condição no fim. Logo, **do-while** sempre será executado pelo menos uma vez.
2. Falso. A condição deve ser de tipo **boolean**.

2. Altere a primeira parte de **main()** para que use um laço na exibição das opções, como mostrado aqui:

```
public static void main(String[] args)
 throws java.io.IOException {
 char choice, ignore;

 do {
 System.out.println("Help on:");
 System.out.println(" 1. if");
 System.out.println(" 2. switch");
 System.out.println(" 3. for");
 System.out.println(" 4. while");
 System.out.println(" 5. do-while\n");
 System.out.print("Choose one: ");

 choice = (char) System.in.read();

 do {
 ignore = (char) System.in.read();
 } while(ignore != '\n');
 } while(choice < '1' | choice > '5');
```

Observe que um laço **do-while** aninhado é usado para descartar qualquer caractere indesejado remanescente no buffer de entrada. Após essa alteração, o programa entrará em laço, exibindo o menu até o usuário inserir uma resposta entre 1 e 5.

3. Expanda a instrução **switch** para incluir os laços **for**, **while** e **do-while**, como mostrado a seguir:

```
switch(choice) {
 case '1':
 System.out.println("The if:\n");
 System.out.println("if(condition) statement;");
 System.out.println("else statement;");
 break;
 case '2':
 System.out.println("The switch:\n");
 System.out.println("switch(expression) {");
 System.out.println(" case constant:");
 System.out.println(" statement sequence");
 System.out.println(" break;");
 System.out.println(" // ...");
 System.out.println("}");
 break;
 case '3':
 System.out.println("The for:\n");
 System.out.println("for(init; condition; iteration)");
 System.out.println(" statement");
 break;
 case '4':
 System.out.println("The while:\n");
 System.out.println("while(condition) statement");
 break;
```

```

 case '5':
 System.out.println("The do-while:\n");
 System.out.println("do {");
 System.out.println(" statement;");
 System.out.println("} while (condition);");
 break;
 }
}

```

Observe que não há instrução **default** presente nessa versão de **switch**. Já que o laço do menu assegura que uma resposta válida seja inserida, não é mais necessário incluir uma instrução **default** para o tratamento de uma escolha inválida.

#### 4. Aqui está o programa **Help2.java** inteiro:

```

/*
 Tente isto 3-2

 Um sistema de ajuda melhorado que usa do-while
 para processar uma seleção no menu.
*/
class Help2 {
 public static void main(String[] args)
 throws java.io.IOException {
 char choice, ignore;

 do {
 System.out.println("Help on:");
 System.out.println(" 1. if");
 System.out.println(" 2. switch");
 System.out.println(" 3. for");
 System.out.println(" 4. while");
 System.out.println(" 5. do-while\n");
 System.out.print("Choose one: ");

 choice = (char) System.in.read();

 do {
 ignore = (char) System.in.read();
 } while(ignore != '\n');
 } while(choice < '1' | choice > '5');

 System.out.println("\n");

 switch(choice) {
 case '1':
 System.out.println("The if:\n");
 System.out.println("if(condition) statement;");
 System.out.println("else statement;");
 break;
 case '2':
 System.out.println("The switch:\n");
 System.out.println("switch(expression) { ");
 System.out.println(" case constant:");

```

```

 System.out.println(" statement sequence");
 System.out.println(" break;");
 System.out.println(" // ...");
 System.out.println("}");
 break;
 case '3':
 System.out.println("The for:\n");
 System.out.print("for(init; condition; iteration)");
 System.out.println(" statement;");
 break;
 case '4':
 System.out.println("The while:\n");
 System.out.println("while(condition) statement;");
 break;
 case '5':
 System.out.println("The do-while:\n");
 System.out.println("do {");
 System.out.println(" statement;");
 System.out.println("} while (condition);");
 break;
 }
}
}

```

## USE break PARA SAIR DE UM LAÇO

É possível forçar a saída imediata de um laço, ignorando o código restante em seu corpo e o teste condicional, com o uso da instrução **break**. Quando uma instrução **break** é encontrada dentro de um laço, este é encerrado e o controle do programa é retomado na instrução posterior ao laço. Veja um exemplo simples:

```

// Usando break para sair de um laço.
class BreakDemo {
 public static void main(String[] args) {
 int num;

 num = 100;

 // executa o laço enquanto i ao quadrado é menor do que num
 for(int i=0; i < num; i++) {
 if(i*i >= num) break; // encerra o laço se i*i >= 100
 System.out.print(i + " ");
 }
 System.out.println("Loop complete.");
 }
}

```

Esse programa gera a saída a seguir:

```
| 0 1 2 3 4 5 6 7 8 9 Loop complete.
```

Como você pode ver, embora o laço **for** tenha sido projetado para ir de 0 a **num** (que nesse caso é 100), a instrução **break** encerra-o prematuramente quando **i** ao quadrado é maior ou igual a **num**.

A instrução **break** pode ser usada com qualquer laço Java, inclusive os intencionalmente infinitos. Por exemplo, o programa abaixo apenas lê a entrada até o usuário digitar a letra q:

```
// Lê a entrada até um q ser recebido.
class Break2 {
 public static void main(String[] args)
 throws java.io.IOException {

 char ch;

 for(; ;) { ←
 ch = (char) System.in.read(); // obtém um char ←
 if(ch == 'q') break; ← } ←
 System.out.println("You pressed q!");
 }
}
```

Este laço “infinito” é encerrado por **break**.

Quando usada dentro de um conjunto de laços aninhados, a instrução **break** encerra apenas o laço mais interno. Por exemplo:

```
// Usando break com laços aninhados.
class Break3 {
 public static void main(String[] args) {

 for(int i=0; i<3; i++) {
 System.out.println("Outer loop count: " + i);
 System.out.print(" Inner loop count: ");

 int t = 0;
 while(t < 100) {
 if(t == 10) break; // encerra o laço se t for 10
 System.out.print(t + " ");
 t++;
 }
 System.out.println();
 }
 System.out.println("Loops complete.");
 }
}
```

Esse programa gera a saída a seguir:

```
Outer loop count: 0
 Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 1
 Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 2
 Inner loop count: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

Como ficou claro, a instrução **break** do laço mais interno causa o encerramento apenas deste laço. O laço externo não é afetado.

Há mais dois fatos sobre **break** que devemos lembrar. Em primeiro lugar, mais de uma instrução **break** pode aparecer em um laço. No entanto, tome cuidado. Muitas instruções **break** podem desestruturar o código. Em segundo lugar, o **break** que termina uma instrução **switch** só afeta a instrução **switch** e não os laços externos.

## USE break COMO UMA FORMA DE goto

Além de seus usos com a instrução **switch** e os laços, a instrução **break** pode ser empregada individualmente para fornecer uma forma “civilizada” da instrução goto. Java não tem uma instrução goto, porque ela fornece uma maneira desestruturada de alterar o fluxo de execução do programa. Geralmente programas que fazem amplo uso de goto são de compreensão e manutenção difíceis. No entanto, há alguns locais em que goto é um artifício útil e legítimo. Por exemplo, goto pode ser útil na saída de um conjunto de laços profundamente aninhado. Para tratar essas situações, Java define uma forma expandida da instrução **break**. Usando essa forma de **break**, você pode, por exemplo, sair de um ou mais blocos de código. Esses blocos não precisam fazer parte de um laço ou de um **switch**. Podem ser qualquer bloco. Além disso, você pode especificar exatamente onde a execução continuará, porque essa forma de **break** funciona com um rótulo. A instrução **break** fornece os benefícios de um goto sem alguns de seus problemas.

A forma geral da instrução **break** rotulada é mostrada aqui:

```
break rótulo;
```

Nesse caso, *rótulo* é o nome que identifica uma instrução ou um bloco de código. Quando essa forma de **break** é executada, o controle é transferido para fora da instrução ou do bloco rotulado. A instrução ou bloco rotulado deve incluir a instrução **break**, mas não precisa ser o bloco imediatamente externo. Ou seja, você pode usar uma instrução **break** rotulada para sair de um conjunto de blocos aninhados, por exemplo, mas não pode usá-la para transferir o controle para um bloco de código que não a inclua.

Para nomear uma instrução ou bloco, insira um rótulo no início dele. Um *rótulo* é qualquer identificador Java válido seguido por dois pontos. Uma vez que você tiver rotulado uma instrução ou bloco, poderá usar esse rótulo como alvo de uma instrução **break**. Isso fará a execução ser retomada no *fim* da instrução ou bloco. Por exemplo, o programa a seguir mostra três blocos aninhados:

```
// Usando break com um rótulo.
class Break4 {
 public static void main(String[] args) {
```

```
int i;

for(i=1; i<4; i++) {
one: {
two: {
three: {
System.out.println("\ni is " + i);
if(i==1) break one; ← Break com um rótulo.
if(i==2) break two;
if(i==3) break three;

// essa parte nunca será alcançada
System.out.println("won't print");
}
System.out.println("After block three.");
}
System.out.println("After block two.");
}
System.out.println("After block one.");
}
System.out.println("After for.");
}
```

A saída do programa é mostrada aqui:

```
i is 1
After block one.

i is 2
After block two.
After block one.

i is 3
After block three.
After block two.
After block one.
After for.
```

Examinemos o programa com mais cuidado para entender exatamente por que essa saída é produzida. Quando **i** é igual a 1, a primeira instrução **if** é bem-sucedida, causando uma parada no fim do bloco de código definido pelo rótulo **one**. Isso faz **After block one**, ser exibido. Quando **i** é igual a 2, o segundo **if** é bem-sucedido, fazendo o controle ser transferido para o fim do bloco rotulado com **two**. Isso faz as mensagens **After block two**, e **After block one**, serem exibidas, nessa ordem. Quando **i** é igual a 3, o terceiro **if** é bem-sucedido e o controle é transferido para o fim do bloco rotulado com **three**. Agora, as três mensagens são exibidas.

Vejamos outro exemplo. Dessa vez, **break** está sendo usado para saltar para fora de uma série de laços **for** aninhados. Quando a instrução **break** do laço interno é executada, o controle do programa salta para o fim do bloco definido pelo laço **for** externo, que foi rotulado com **done**. Isso faz os outros três laços serem ignorados.

```
// Outro exemplo do uso de break com um rótulo.
class Break5 {
 public static void main(String[] args) {

done:
 for(int i=0; i<10; i++) {
 for(int j=0; j<10; j++) {
 for(int k=0; k<10; k++) {
 System.out.println(k + " ");
 if(k == 5) break done; // salta para done
 }
 System.out.println("After k loop"); // não será executado
 }
 System.out.println("After j loop"); // não será executado
 }
 System.out.println("After i loop");
}
}
```

A saída do programa é mostrada abaixo:

```
0
1
2
3
4
5
After i loop
```

É muito importante o local exato onde um rótulo é inserido – principalmente no trabalho com laços. Por exemplo, considere o programa a seguir:

```
// É importante onde o rótulo é inserido.
class Break6 {
 public static void main(String[] args) {
 int x=0, y=0;

// aqui, insere o rótulo antes da instrução for.
stop1: for(x=0; x < 5; x++) {
 for(y = 0; y < 5; y++) {
 if(y == 2) break stop1;
 System.out.println("x and y: " + x + " " + y);
 }
}

System.out.println();

// agora, insere o rótulo imediatamente antes de {
for(x=0; x < 5; x++)
stop2: {
 for(y = 0; y < 5; y++) {
```

```
 if(y == 2) break stop2;
 System.out.println("x and y: " + x + " " + y);
 }
}
}
```

A saída do programa é esta:

```
x and y: 0 0
x and y: 0 1

x and y: 0 0
x and y: 0 1
x and y: 1 0
x and y: 1 1
x and y: 2 0
x and y: 2 1
x and y: 3 0
x and y: 3 1
x and y: 4 0
x and y: 4 1
```

No programa, os dois conjuntos de laços aninhados são iguais exceto por uma coisa. No primeiro conjunto, o rótulo precede a instrução **for** externa. Nesse caso, quando **break** é executado, transfere o controle para o fim do bloco **for** inteiro, saltando as outras iterações do laço externo. No segundo conjunto, o rótulo precede a chave de abertura do **for** externo. Logo, quando **break stop2** é executado, o controle é transferido para o fim do bloco **for** externo e não para o fim do laço. Isso faz a próxima iteração ocorrer.

Lembre-se de que você não pode usar a instrução **break** com um rótulo que não foi definido para uma instrução ou bloco que a inclua. Por exemplo, o programa abaixo é inválido e não será compilado:

```
// Este programa contém um erro.
class BreakErr {
 public static void main(String[] args) {
 one: for(int i=0; i<3; i++) {
 System.out.print("Pass " + i + ": ");
 }

 for(int j=0; j<100; j++) {
 if(j == 10) break one; // ERRADO
 System.out.print(j + " ");
 }
 }
}
```

Como o laço **for** rotulado com **one** não inclui a instrução **break** do segundo laço **for**, não é possível transferir o controle para esse rótulo.

## Pergunte ao especialista

**P** Você diz que **goto** é desestruturado e que **break** com um rótulo oferece uma alternativa melhor. Mas, convenhamos, usar **break** com um rótulo, que pode resultar em muitas linhas de código e níveis de aninhamento sendo removidos por **break**, também não desestrutura o código?

**R** Uma resposta rápida seria sim! No entanto, nos casos em que uma mudança drástica é necessária no fluxo do programa, usar **break** com um rótulo ainda mantém alguma estrutura porque você só pode saltar para fora de uma instrução ou bloco externo rotulado. Não pode saltar para qualquer instrução ou bloco arbitrário. Em contrapartida, **goto** basicamente não tem estrutura!

## USE continue

É possível forçar uma iteração antecipada de um laço, ignorando sua estrutura de controle normal. Isso é feito com o uso de **continue**. A instrução **continue** força a ocorrência da próxima iteração do laço e qualquer código existente entre ela e a expressão condicional que controla o laço é ignorado. Logo, **continue** é basicamente o complemento de **break**. Por exemplo, o programa a seguir usa **continue** para ajudar a exibir os números pares entre 0 e 100:

```
// Usa continue.
class ContDemo {
 public static void main(String[] args) {
 int i;

 // exibe os números pares entre 0 e 100
 for(i = 0; i<=100; i++) {
 if((i%2) != 0) continue; // iterate
 System.out.println(i);
 }
 }
}
```

Só números pares são exibidos, porque um número ímpar faria o laço iterar antecipadamente, ignorando a chamada a **println( )**. Isso é feito com o uso do operador **%**, que retorna o resto de uma divisão. Se o número for par, o resto de uma divisão por 2 será zero e **if** falhará. Se o número for ímpar, o resto será 1, fazendo **if** executar a instrução **continue**.

Em laços **while** e **do-while**, uma instrução **continue** faria o controle ir diretamente para a expressão condicional e então continuar o processo de execução do laço. No caso de **for**, a expressão de iteração do laço é avaliada, a expressão condicional é executada e o laço continua.

Uma instrução **continue** pode especificar um rótulo para descrever qual laço externo deve prosseguir. Aqui está um exemplo de programa que usa **continue** com um rótulo:

```
// Usa continue com um rótulo.
class ContToLabel {
```

```

public static void main(String[] args) {
outerloop:
 for(int i=1; i < 10; i++) {
 System.out.print("\nOuter loop pass " + i +
 ", Inner loop: ");
 for(int j = 1; j < 10; j++) {
 if(j == 5) continue outerloop; // laço externo de continue
 System.out.print(j);
 }
 }
}
}

```

A saída do programa é mostrada abaixo:

```

Outer loop pass 1, Inner loop: 1234
Outer loop pass 2, Inner loop: 1234
Outer loop pass 3, Inner loop: 1234
Outer loop pass 4, Inner loop: 1234
Outer loop pass 5, Inner loop: 1234
Outer loop pass 6, Inner loop: 1234
Outer loop pass 7, Inner loop: 1234
Outer loop pass 8, Inner loop: 1234
Outer loop pass 9, Inner loop: 1234

```

Como a saída mostra, quando **continue** é executado, o controle passa para o laço externo, saltando o restante do laço interno.

Bons usos para **continue** são raros. Uma das razões é a linguagem Java fornecer um rico conjunto de instruções de laço que atende à maioria das aplicações. No entanto, para circunstâncias especiais em que a iteração antecipada é necessária, a instrução **continue** fornece uma maneira estruturada de a executarmos.

## Verificação do progresso

1. Dentro de um laço, o que ocorre quando um **break** (sem rótulo) é executado?
2. O que ocorre quando um **break** com rótulo é executado?
3. O que **continue** faz?

Respostas:

1. Dentro de um laço, um **break** sem rótulo causa o encerramento imediato do laço. A execução é retomada na primeira linha de código após o laço.
2. Quando um **break** rotulado é executado, a execução é retomada na primeira linha de código após a instrução ou bloco rotulado.
3. A instrução **continue** faz um laço iterar imediatamente, ignorando o restante do código. Se **continue** incluir um rótulo, o laço rotulado será executado.

### TENTE ISTO 3-3 Termine o sistema de ajuda Java

`Help3.java`

Este projeto dá os toques finais no sistema de ajuda Java que foi criado nos projetos anteriores. Essa versão adiciona a sintaxe de **break** e **continue**. Também permite que o usuário solicite a sintaxe de mais de uma instrução. Ela faz isso adicionando um laço externo que é executado até o usuário inserir **q** como seleção no menu.

#### PASSO A PASSO

1. Copie **Help2.java** em um novo arquivo chamado **Help3.java**.
2. Inclua todo o código do programa em um laço **for** infinito. Saia desse laço, usando **break**, quando uma letra **q** for inserida. Como o laço engloba todo o código do programa, sair dele faz o programa terminar.
3. Altere o laço do menu como mostrado aqui:

```
do {
 System.out.println("Help on:");
 System.out.println(" 1. if");
 System.out.println(" 2. switch");
 System.out.println(" 3. for");
 System.out.println(" 4. while");
 System.out.println(" 5. do-while");
 System.out.println(" 6. break");
 System.out.println(" 7. continue\n");
 System.out.print("Choose one (q to quit): ");

 choice = (char) System.in.read();

 do {
 ignore = (char) System.in.read();
 } while(ignore != '\n');
} while(choice < '1' | choice > '7' & choice != 'q');
```

Observe que agora esse laço inclui as instruções **break** e **continue**. Ele também aceita a letra **q** como opção válida.

4. Expanda a instrução **switch** para incluir as instruções **break** e **continue**, como mostrado abaixo:

```
case '6':
 System.out.println("The break:\n");
 System.out.println("break; or break label;");
 break;
case '7':
 System.out.println("The continue:\n");
 System.out.println("continue; or continue label;");
 break;
```

**5.** Este é o programa **Help3.java** inteiro:

```
/*
Tente isto 3-3

O sistema de ajuda em instruções Java que
processa várias solicitações terminado.

*/
class Help3 {
 public static void main(String[] args)
 throws java.io.IOException {
 char choice, ignore;

 for(;;) {
 do {
 System.out.println("Help on:");
 System.out.println(" 1. if");
 System.out.println(" 2. switch");
 System.out.println(" 3. for");
 System.out.println(" 4. while");
 System.out.println(" 5. do-while");
 System.out.println(" 6. break");
 System.out.println(" 7. continue\n");
 System.out.print("Choose one (q to quit): ");

 choice = (char) System.in.read();

 do {
 ignore = (char) System.in.read();
 } while(ignore != '\n');
 } while(choice < '1' | choice > '7' & choice != 'q');

 if(choice == 'q') break;

 System.out.println("\n");

 switch(choice) {
 case '1':
 System.out.println("The if:\n");
 System.out.println("if(condition) statement;");
 System.out.println("else statement;");
 break;
 case '2':
 System.out.println("The switch:\n");
 System.out.println("switch(expression) { ");
 System.out.println(" case constant:");
 System.out.println(" statement sequence");
 System.out.println(" break; ");
 System.out.println(" // ...");
 System.out.println("} ");
 break;
 case '3':
 System.out.println("The for:\n");
 System.out.print("for(init; condition; iteration)");
 }
 }
 }
}
```

```
 System.out.println(" statement;");
 break;
 case '4':
 System.out.println("The while:\n");
 System.out.println("while(condition) statement;");
 break;
 case '5':
 System.out.println("The do-while:\n");
 System.out.println("do {");
 System.out.println(" statement;");
 System.out.println("} while (condition);");
 break;
 case '6':
 System.out.println("The break:\n");
 System.out.println("break; or break label;");
 break;
 case '7':
 System.out.println("The continue:\n");
 System.out.println("continue; or continue label;");
 break;
 }
 System.out.println();
}
}
}
```

**6.** Aqui está um exemplo da execução:

```
Help on:
1. if
2. switch
3. for
4. while
5. do-while
6. break
7. continue

Choose one (q to quit): 1

The if:

if(condition) statement;
else statement;
Help on:
1. if
2. switch
3. for
4. while
5. do-while
6. break
7. continue
```

```
Choose one (q to quit): 6
```

The break:

```
break; or break label;
```

Help on:

1. if
2. switch
3. for
4. while
5. do-while
6. break
7. continue

```
Choose one (q to quit): q
```

## LAÇOS ANINHADOS

Como vimos em alguns dos exemplos anteriores, um laço pode ser aninhado dentro de outro. Os laços aninhados são usados para resolver uma grande variedade de problemas de programação e são parte essencial do ato de programar. Portanto, antes de encerrarmos o tópico das instruções de laço Java, examinemos mais um exemplo de laço aninhado. O programa a seguir usa um laço **for** aninhado para encontrar todos os fatores (exceto 1 e o próprio número) dos números de 2 a 100. Observe que o laço externo produz os números cujos fatores serão obtidos. O laço interno determina os fatores dos números.

```
/*
 Usa laços aninhados para encontrar os fatores dos números
 de 2 a 100.
*/
class FindFac {
 public static void main(String[] args) {

 for(int i=2; i <= 100; i++) {
 System.out.print("Factors of " + i + ": ");
 for(int j = 2; j < i; j++)
 if((i%j) == 0) System.out.print(j + " ");
 System.out.println();
 }
 }
}
```

Aqui está uma parte da saída produzida pelo programa:

```
Factors of 2:
Factors of 3:
Factors of 4: 2
```

```
Factors of 5:
Factors of 6: 2 3
Factors of 7:
Factors of 8: 2 4
Factors of 9: 3
Factors of 10: 2 5
Factors of 11:
Factors of 12: 2 3 4 6
Factors of 13:
Factors of 14: 2 7
Factors of 15: 3 5
Factors of 16: 2 4 8
Factors of 17:
Factors of 18: 2 3 6 9
Factors of 19:
Factors of 20: 2 4 5 10
```

No programa, o laço externo executa **i** de 2 a 100. O laço interno testa sucessivamente todos os números de 2 a **i**, exibindo aqueles cuja divisão por **i** é exata. Observe o uso do operador % para determinar quando um valor gera uma divisão exata por outro. Se o resultado for zero, o divisor é um fator. Um desafio adicional: o programa anterior pode ser mais eficiente. Consegue ver como? (Dica: o número de iterações do laço interno pode ser reduzido.)

---

## EXERCÍCIOS

1. Escreva um programa que leia caracteres do teclado até um ponto ser recebido. Faça-o contar o número de espaços. Relate o total no fim do programa.
2. Mostre a forma geral da escada **if-else-if**.
3. Dado o código

```
if(x < 10)
 if(y > 100) {
 if(!done) x = z;
 else y = z;
 }
 else System.out.println("error"); // que if?
```

a que **if** o último **else** está associado?

4. Mostre a instrução **for** de um laço que conte de 1.000 a 0 em intervalos de -2.
5. O fragmento a seguir é válido?

```
for(int i = 0; i < num; i++)
 sum += i;

count = i;
```

6. Explique o que **break** faz. Certifique-se de explicar suas duas formas.  
 7. No fragmento a seguir, após a instrução **break** ser executada, o que é exibido?

```
| for(i = 0; i < 10; i++) {
 while(running) {
 if(x<y) break;
 // ...
 }
 System.out.println("after while");
}
System.out.println("After for");
```

8. O que o fragmento abaixo exibe?

```
| for(int i = 0; i<10; i++) {
 System.out.print(i + " ");
 if((i%2) == 0) continue;
 System.out.println();
}
```

9. Nem sempre a expressão de iteração de um laço **for** tem de alterar a variável de controle de laço adicionando ou subtraindo um valor fixo. Em vez disso, a variável de controle pode mudar de qualquer maneira arbitrária. Usando esse conceito, escreva um programa que use um laço **for** para gerar e exibir a progressão 1, 2, 4, 8, 16, 32 e assim por diante.
10. As letras minúsculas ASCII ficam separadas das maiúsculas por um intervalo igual a 32. Logo, para converter uma letra minúscula em maiúscula, temos de subtrair 32 dela. Use essa informação para escrever um programa que leia caracteres do teclado. Faça-o converter todas as letras minúsculas em maiúsculas e todas as letras maiúsculas em minúsculas, exibindo o resultado. Não faça alterações em outros caracteres. O programa será encerrado quando o usuário pressionar o ponto. No fim, ele deve exibir quantas alterações ocorreram na caixa das letras.
11. O que é um laço infinito?
12. No uso de **break** com um rótulo, este deve estar em uma instrução ou bloco que contenha **break**?
13. Qual é a diferença entre os três valores literais a seguir: 5, ‘5’, “5”?
14. Suponha que **c** seja uma variável de tipo **char**. Como você verificararia se o valor de **c** é o caractere de aspas simples?
15. A classe **ContDemo** deste capítulo mostra uma maneira de usarmos um laço **for** para exibir os números pares de 0 a 100. Crie programas que exibam essa mesma saída, mas como descrito a seguir:
- Usando um laço **for** que incremente a variável de controle em duas unidades a cada iteração.
  - Usando um laço **for** cuja variável de controle vá de 0 a 50.

- C. Usando um laço **for** cuja variável de controle retroceda de 100 a 0.
  - D. Usando um laço **for** infinito sem expressão condicional e saindo do laço com uma instrução **break**.
  - E. Usando um laço **while**.
  - F. Usando um laço **do-while**.
16. Crie um programa que use um laço para exibir as potências de 3, de  $3^0$  até e incluindo  $3^9$ .
17. Crie um programa que use um laço para exibir uma lista de 100 números composta por 1s e -1s alternados, começando com 1.
18. A classe **FindFac** discutida neste capítulo exibe os fatores de todos os números de 1 a 100. Modifique essa classe para que, em vez de parar em 100, ela continue até encontrar um número com exatamente nove fatores.
19. Crie um programa que leia caracteres do teclado até ler um caractere de alimentação de linha '\n'. Em seguida, faça-o exibir o número de vogais, de consoantes, de dígitos e de outros caracteres. Inclua o caractere final de alimentação de linha na contagem dos outros caracteres.
20. O programa **StarPattern** abaixo exibe o padrão de asteriscos que aparece logo após. Modifique o programa para que exiba os outros padrões usando laços aninhados.

```
class StarPattern {
 public static void main(String[] args) {

 for(int i = 1; i <= 5; i++) {
 for(int j = 1; j <= i; j++)
 System.out.print('*');
 System.out.println();
 }
 }
}
```

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

A. \*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

B. \*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

C. \*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*  
\*\*

21. Como mencionado no texto, um identificador Java é composto por um ou mais caracteres. O primeiro caractere deve ser uma letra maiúscula ou minúscula do alfabeto, um sublinhado (\_) ou um cifrão (\$). Cada caractere restante deve ser uma letra maiúscula ou minúscula do alfabeto, um dígito de 0 a 9, um sublinhado ou um cifrão. Crie um programa Java que leia uma linha de caracteres e exiba se ela é um identificador Java válido.
22. Infelizmente, os valores Unicode dos caracteres ‘0’-‘9’ não correspondem aos seus valores inteiros. Isto é, os valores Unicode de ‘0’-‘9’ são 48-57 e não 0-9. Mas podemos converter facilmente esses caracteres em seus valores inteiros subtraindo 48. Especificamente, se **c** for uma variável de tipo **char** contendo um dígito de ‘0’-‘9’, então podemos criar uma variável **x** de tipo **int** com os valores inteiros correspondentes como descrito a seguir:

```
| int x = c - 48;
```

Use essa técnica de conversão em um programa que leia três dígitos, convertê-los em um inteiro com os três dígitos, dobre o valor do inteiro e então exiba o resultado. Por exemplo, se a entrada for ‘3’, ‘4’ e ‘5’, a saída será 690.

23. Se você dividir 1 por 2, obterá 0,5. Se dividir novamente por 2, obterá 0,25. Crie um programa que calcule e exiba quantas vezes temos que dividir 1 por 2 para obter um valor menor do que um décimo de milésimo (0,0001).

# 4

## Introdução a classes, objetos e métodos

### PRINCIPAIS HABILIDADES E CONCEITOS

- Saber os fundamentos da classe
- Entender como os objetos são criados
- Entender como as variáveis de referência são atribuídas
- Criar um método
- Usar a palavra-chave **return**
- Retornar um valor de um método
- Adicionar parâmetros a um método
- Utilizar construtores
- Criar construtores parametrizados
- Entender **new**
- Entender a coleta de lixo e os finalizadores
- Usar a palavra-chave **this**

Antes de ir adiante em seu estudo de Java, você precisa conhecer a classe. A classe é a essência de Java. Ela é a estrutura lógica sobre a qual a linguagem Java é construída, porque define a natureza de um objeto. Como tal, forma a base da programação orientada a objetos em Java. Dentro de uma classe, são definidos dados e o código que age sobre eles. O código fica contido em métodos. Como classes, objetos e métodos são fundamentais para Java, eles serão introduzidos neste capítulo. Ter um entendimento básico desses recursos permitirá que você escreva programas mais sofisticados e compreenda melhor certos elementos-chave de Java descritos no próximo capítulo.

### FUNDAMENTOS DAS CLASSES

Já que toda atividade dos programas Java ocorre dentro de uma classe, temos usado classes desde o início deste livro. É claro que só foram usadas classes extremamente simples, e não nos beneficiamos da maioria de seus recursos. Como você verá, elas são significativamente mais poderosas do que as classes limitadas apresentadas até agora.

Comecemos examinando o básico. Uma classe é um modelo que define a forma de um objeto. Ela especifica tanto os dados quanto o código que operará sobre eles. Java usa uma especificação de classe para construir *objetos*. Os objetos são *instâncias* de uma classe. Logo, uma classe é basicamente um conjunto de planos que especifica como construir um objeto. É importante deixar uma questão bem clara: uma classe é uma abstração. Só quando um objeto dessa classe é criado é que existe uma representação física dela na memória.

Outro ponto: lembre-se de que os métodos e variáveis que compõem uma classe são chamados de *membros* da classe. Os membros de dados associados à instância de uma classe também são chamados de *variáveis de instância*.

## Forma geral de uma classe

Quando definimos uma classe, declaramos sua forma e natureza exatas. Fazemos isso especificando as variáveis de instância que ela contém e os métodos que operam sobre elas. Embora classes muito simples possam conter apenas métodos ou apenas variáveis de instância, a maioria das classes do mundo real contém ambos.

Uma classe é criada com o uso da palavra-chave **class**. Uma forma geral simplificada de uma definição **class** é mostrada aqui:

```
class nome_da_classe {
 // declara variáveis de instância
 tipo var1;
 tipo var2;
 // ...
 tipo varN;

 // declara métodos
 tipo método1(parâmetros) {
 // corpo do método
 }
 tipo método2(parâmetros) {
 // corpo do método
 }
 // ...
 tipo métodoN(parâmetros) {
 // corpo do método
 }
}
```

Embora não haja essa regra sintática, uma classe bem projetada deve definir apenas uma entidade lógica. Por exemplo, normalmente uma classe que armazena nomes e números de telefone não armazena também informações sobre o mercado de ações, a média pluviométrica, os ciclos das manchas solares ou outros dados não relacionados. Ou seja, uma classe bem projetada deve agrupar informações logicamente conectadas. A inserção de informações não relacionadas na mesma classe desestruturará rapidamente seu código!

Até o momento, as classes que usamos tinham apenas um método: **main( )**. Você verá como criar outros em breve. No entanto, observe que a forma geral de uma classe não especifica um método **main( )**. O método **main( )** só é necessário quando a classe é o ponto de partida do programa. Alguns tipos de aplicativos Java, como os applets, também não precisam de **main( )**.

## Definindo uma classe

Para ilustrar as classes, desenvolveremos uma classe que encapsula informações sobre veículos, como carros, furgões e caminhões. Essa classe se chamará **Vehicle** e conterá três informações sobre um veículo: o número de passageiros que ele pode levar, a capacidade de armazenamento de combustível e o consumo médio de combustível (em milhas por galão).

A primeira versão de **Vehicle** é mostrada a seguir. Ela define três variáveis de instância: **passengers**, **fuelCap** e **mpg**. Observe que **Vehicle** não contém métodos. Logo, atualmente é uma classe só de dados. (Seções subsequentes adicionarão métodos a ela.)

```
class Vehicle {
 int passengers; // número de passageiros
 int fuelCap; // capacidade de armazenamento de combustível em galões
 int mpg; // consumo de combustível em milhas por galão
}
```

Uma definição **class** cria um novo tipo de dado. Nesse caso, ele se chama **Vehicle**. Você usará esse nome para declarar objetos de tipo **Vehicle**. Lembre-se de que uma declaração **class** é só uma descrição de tipo; ela não cria um objeto real. Logo, o código anterior não faz um objeto de tipo **Vehicle** passar a existir.

Para criar realmente um objeto **Vehicle**, você usará uma instrução como a mostrada abaixo:

```
| Vehicle minivan = new Vehicle(); // cria um objeto Vehicle chamado minivan
```

Após essa instrução ser executada, **minivan** será uma instância de **Vehicle**. Portanto, terá realidade “física”. Por enquanto, não se preocupe com os detalhes da instrução.

Sempre que você criar uma instância de uma classe, estará criando um objeto contendo sua própria cópia de cada variável de instância definida pela classe. Assim, todos os objetos **Vehicle** conterão suas próprias cópias das variáveis de instância **passengers**, **fuelCap** e **mpg**. Para acessar essas variáveis, você usará o que é normalmente chamado de operador ponto (**.**). O *operador ponto* vincula o nome de um objeto ao nome de um membro. A forma geral do operador ponto é mostrada aqui:

*objeto.membro*

Portanto, o objeto é especificado à esquerda e o membro é inserido à direita. Por exemplo, para atribuir o valor 16 à variável **fuelCap** de **minivan**, use a instrução a seguir:

```
| minivan.fuelCap = 16;
```

Em geral, podemos usar o operador ponto para acessar tanto variáveis de instância quanto métodos.

Este é um programa completo que usa a classe **Vehicle**:

```
/* Um programa que usa a classe Vehicle.

Chame este arquivo de VehicleDemo.java
*/
class Vehicle {
 int passengers; // número de passageiros
 int fuelCap; // capacidade de armazenamento de combustível em galões
 int mpg; // consumo de combustível em milhas por galão
}

// Esta classe declara um objeto de tipo Vehicle.
class VehicleDemo {
 public static void main(String[] args) {
 Vehicle minivan = new Vehicle();
 int range;

 // atribui valores a campos de minivan
 minivan.passengers = 7;
 minivan.fuelCap = 16; ← Observe o uso do operador ponto
 minivan.mpg = 21; para o acesso a um membro.

 // calcula a autonomia presumindo um tanque cheio de gasolina
 range = minivan.fuelCap * minivan.mpg;
 System.out.println("Minivan can carry " + minivan.passengers +
 " with a range of " + range);
 }
}
```

O arquivo que contém o programa deve ser chamado de **VehicleDemo.java**, porque o método **main()** está na classe chamada **VehicleDemo** e não na classe chamada **Vehicle**. Quando compilar esse programa, você verá que dois arquivos **.class** foram criados, um para **Vehicle** e um para **VehicleDemo**. O compilador Java insere automaticamente cada classe em seu próprio arquivo **.class**. Não é necessário as classes **Vehicle** e **VehicleDemo** estarem no mesmo arquivo-fonte. Você pode inserir cada classe em seu próprio arquivo, chamados **Vehicle.java** e **VehicleDemo.java**, respectivamente.

Para executar o programa, você deve executar **VehicleDemo.java**. A saída a seguir é exibida:

```
|Minivan can carry 7 with a range of 336
```

Antes de avançar, examinemos um princípio básico: cada objeto tem suas próprias cópias das variáveis de instância definidas por sua classe. Logo, o conteúdo das variáveis de um objeto pode diferir do conteúdo das variáveis de outro. Não há conexão entre os dois objetos exceto pelo fato de serem do mesmo tipo. Por exemplo, se você tiver dois objetos **Vehicle**, cada um terá sua própria cópia de **passengers**, **fuelCap** e **mpg**, e o conteúdo dessas variáveis será diferente entre os dois objetos. O programa a seguir demonstra esse fato. (Observe que a classe que tem **main()** agora se chama **TwoVehicles**.)

```

// Este programa cria dois objetos Vehicle.

class Vehicle {
 int passengers; // número de passageiros
 int fuelCap; // capacidade de armazenamento de combustível em galões
 int mpg; // consumo de combustível em milhas por galão
}

// Esta classe declara dois objetos de tipo Vehicle.
class TwoVehicles {
 public static void main(String[] args) {
 Vehicle minivan = new Vehicle(); Lembre-se de que
 Vehicle sportscar = new Vehicle(); minivan e sportscar
 referenciam objetos
 separados.

 int range1, range2;

 // atribui valores a campos de minivan
 minivan.passengers = 7;
 minivan.fuelCap = 16;
 minivan.mpg = 21;

 // atribui valores a campos de sportscar
 sportscar.passengers = 2;
 sportscar.fuelCap = 14;
 sportscar.mpg = 12;

 // calcula a autonomia presumindo um tanque cheio de gasolina
 range1 = minivan.fuelCap * minivan.mpg;
 range2 = sportscar.fuelCap * sportscar.mpg;

 System.out.println("Minivan can carry " + minivan.passengers +
 " with a range of " + range1);

 System.out.println("Sportscar can carry " + sportscar.passengers +
 " with a range of " + range2);
 }
}

```

A saída produzida por esse programa é mostrada aqui:

```

Minivan can carry 7 with a range of 336
Sportscar can carry 2 with a range of 168

```

Como você pode ver, os dados de **minivan** são totalmente diferentes dos contidos em **sportscar**. A ilustração a seguir mostra essa situação.

|           |   |           |    |
|-----------|---|-----------|----|
| minivan   | → | passenger | 7  |
|           |   | fuelCap   | 16 |
|           |   | mpg       | 21 |
| sportscar | → | passenger | 2  |
|           |   | fuelCap   | 14 |
|           |   | mpg       | 12 |

### Verificação do progresso

1. Quais são os dois elementos que uma classe contém?
2. O que é usado quando acessamos membros de uma classe por intermédio de um objeto?
3. Cada objeto tem suas próprias cópias das \_\_\_\_\_ da classe.

## COMO OS OBJETOS SÃO CRIADOS

Nos programas anteriores, a linha abaixo foi usada para declarar um objeto de tipo **Vehicle**:

```
| Vehicle minivan = new Vehicle();
```

Essa declaração faz duas coisas. Em primeiro lugar, ela declara uma variável chamada **minivan** da classe **Vehicle**. Essa variável não define um objeto. Em vez disso, ela pode apenas *referenciar* um objeto. Em segundo lugar, a declaração cria uma cópia física do objeto e atribui à **minivan** uma referência a ele. Isso é feito com o uso do operador **new**.

O operador **new** aloca dinamicamente (isto é, aloca no tempo de execução) memória para um objeto e retorna uma referência a ele. Essa referência é, basicamente, o endereço do objeto na memória alocado por **new**. A referência é então armazenada em uma variável. Logo, em Java, todos os objetos de uma classe devem ser alocados dinamicamente.

As duas etapas da instrução anterior podem ser reescritas desta forma para mostrarmos cada etapa individualmente:

```
| Vehicle minivan; // declara uma referência ao objeto
| minivan = new Vehicle(); // aloca um objeto Vehicle
```

A primeira linha declara **minivan** como referência a um objeto de tipo **Vehicle**. Portanto, **minivan** é uma variável que pode referenciar um objeto, mas não é um objeto. Por enquanto, **minivan** não referencia um objeto. A próxima linha cria um novo objeto **Vehicle** e atribui à **minivan** uma referência a ele. Agora, **minivan** está vinculada a um objeto.

## AS VARIÁVEIS DE REFERÊNCIA E A ATRIBUIÇÃO

Em uma operação de atribuição, variáveis de referência de objeto podem agir diferentemente do esperado. Para entender o porquê, primeiro considere o que ocorre quando a atribuição se dá entre variáveis de tipo primitivo. Supondo duas variáveis **int** chamadas **x** e **y**, a instrução **x = y** significa que **x** recebe uma *cópia* do valor contido em **y**. Logo, após a atribuição, tanto **x** quanto **y** contêm suas próprias cópias independentes do valor. A alteração de uma não afeta a outra.

**Respostas:**

1. Código e dados. Em Java, isso significa métodos e variáveis de instância.
2. O operador ponto.
3. variáveis de instância

Quando a atribuição se dá entre variáveis de referência de objeto, a situação é um pouco mais complicada, porque estamos atribuindo referências. Ou seja, estamos alterando o objeto para o qual a variável de referência aponta em vez de fazer uma cópia desse objeto. Inicialmente, o efeito dessa diferença pode parecer inesperado. Por exemplo, considere o fragmento a seguir:

```
| Vehicle car1 = new Vehicle();
| Vehicle car2 = car1;
```

À primeira vista, é fácil achar que **car1** e **car2** referenciam objetos diferentes, mas não é esse o caso porque não foi feita uma cópia do objeto. Em vez disso, **car2** recebe uma cópia da *referência* de **car1**. Como resultado, tanto **car1** quanto **car2** referenciação o *mesmo* objeto. Em outras palavras, a atribuição de **car1** a **car2** simplesmente faz **car2** referenciar o mesmo objeto que **car1**. Logo, **car1** ou **car2** podem atuar sobre o objeto. Por exemplo, após a atribuição

```
| car1.mpg = 26;
ser executada, estas duas instruções println()
| System.out.println(car1.mpg);
| System.out.println(car2.mpg);
```

exibirão o mesmo valor: 26.

Embora tanto **car1** quanto **car2** referenciem o mesmo objeto, não há outro tipo de vinculação entre elas. Por exemplo, uma atribuição subsequente a **car2** alteraria apenas o objeto que **car2** referencia, como mostrado abaixo:

```
| Vehicle car1 = new Vehicle();
| Vehicle car2 = car1;
| Vehicle car3 = new Vehicle();

| car2 = car3; // agora car2 e car3 referenciam o mesmo objeto.
```

Após essa sequência ser executada, **car2** referenciará o mesmo objeto que **car3**. O objeto referenciado por **car1** permanece inalterado.

## Verificação do progresso

1. Explique o que ocorre quando uma variável de referência é atribuída a outra.
2. Supondo uma classe chamada **MyClass**, mostre como um objeto chamado **ob** é criado.

### Respostas:

1. Quando uma variável de referência é atribuída a outra variável de referência, as duas variáveis referenciam o mesmo objeto. Não é feita uma cópia do objeto.
2. **Myclass ob = new MyClass( );**

## MÉTODOS

Como explicado, as variáveis de instância e os métodos são componentes das classes. Até agora, a classe **Vehicle** contém dados, mas não métodos. Embora classes só de dados sejam perfeitamente válidas, a maioria das classes terá métodos. Os métodos são sub-rotinas que tratam os dados definidos pela classe e, em muitos casos, controlam o acesso a esses dados. Quase sempre, outras partes do programa interagem com uma classe por seus métodos.

Um método contém as instruções que definem suas ações. Em um código Java bem escrito, cada método executa apenas uma tarefa. Cada método tem um nome, o qual é usado para chamar o método. Em geral, podemos dar o nome que quisermos, contanto que ele seja um identificador válido. No entanto, as boas práticas de programação preconizam que devemos usar nomes descritivos. Lembre-se de que **main()** está reservado para o método que começa a execução do programa. Além disso, não use palavras-chave Java para nomear métodos.

Para representar métodos no texto, este livro tem usado e continuará usando uma convenção que se tornou comum quando se escreve sobre Java: o método terá parênteses após seu nome. Por exemplo, se o nome de um método for **getVal**, ele será escrito na forma **getVal()** quando seu nome for usado em uma frase. Essa notação ajudará a distinguir nomes de variáveis de nomes de métodos no livro.

A forma geral de um método é mostrada abaixo:

```
tipo-ret nome (lista-parâmetros) {
 // corpo do método
}
```

Aqui, *tipo-ret* especifica o tipo de dado retornado pelo método. Ele pode ser qualquer tipo válido, inclusive os tipos de classe que você criar. Se o método não retornar um valor, seu tipo de retorno deve ser **void**. O nome do método é especificado por *nome*. Ele pode ser qualquer identificador válido exceto os já usados por outros itens do escopo atual. A *lista-parâmetros* é uma sequência de pares separados por vírgulas compostos por tipo e identificador. Os parâmetros são basicamente variáveis que recebem o valor dos *argumentos* passados para o método quando ele é chamado. Se o método não tiver parâmetros, a lista estará vazia.

### Adicionando um método à classe Vehicle

Como acabei de explicar, normalmente os métodos de uma classe tratam e dão acesso aos dados da classe. Com isso em mente, lembre-se de que o método **main()** dos exemplos anteriores calculava a autonomia de um veículo multiplicando seu consumo pela capacidade de armazenamento de combustível. Embora tecnicamente correta, essa não é a melhor maneira de fazer o cálculo. O cálculo da autonomia de um veículo é realizado mais adequadamente pela própria classe **Vehicle**. É fácil entender o porquê: a autonomia de um veículo depende da capacidade do tanque de combustível e da taxa de consumo e esses dois valores são encapsulados por **Vehicle**. Ao adicionar à classe **Vehicle** um método que calcule a autonomia, você estará melhorando sua estrutura orientada a objetos. Para adicionar um método a **Vehicle**, especifique-o dentro da declaração da classe. Por exemplo, a versão a seguir de **Vehicle** contém um método chamado **range()** que exibe a autonomia do veículo.

```
// Adiciona range a Vehicle.

class Vehicle {
 int passengers; // número de passageiros
 int fuelCap; // capacidade de armazenamento de combustível em galões
 int mpg; // consumo de combustível em milhas por galão

 // Exibe a autonomia.
 void range() { ←
 System.out.println("Range is " + fuelCap * mpg);
 }
}

Observe que fuelCap e mpg são usadas diretamente, sem o operador ponto.

class AddMeth {
 public static void main(String[] args) {
 Vehicle minivan = new Vehicle();
 Vehicle sportscar = new Vehicle();

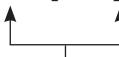
 int range1, range2;

 // atribui valores a campos de minivan
 minivan.passengers = 7;
 minivan.fuelCap = 16;
 minivan.mpg = 21;

 // atribui valores a campos de sportscar
 sportscar.passengers = 2;
 sportscar.fuelCap = 14;
 sportscar.mpg = 12;

 System.out.print("Minivan can carry " + minivan.passengers + ". ");
 minivan.range(); // exibe a autonomia de minivan
 System.out.print("Sportscar can carry " + sportscar.passengers +
". ");
 sportscar.range(); // exibe a autonomia de sportscar.
 }
}
```

O método **range()** está contido na classe **Vehicle**.



Esse programa gera a saída abaixo:

```
| Minivan can carry 7. Range is 336
| Sportscar can carry 2. Range is 168
```

Examinemos os elementos-chave do programa, começando com o método **range()**. A primeira linha de **range()** é

```
| void range() {
```

Essa linha declara um método chamado **range** que não tem parâmetros. Seu tipo de retorno é **void**. Logo, **range( )** não retorna um valor para o chamador. A linha termina com a chave de abertura do corpo do método.

O corpo de **range( )** é composto apenas pela linha a seguir:

```
| System.out.println("Range is " + fuelCap * mpg);
```

Essa instrução exibe a autonomia do veículo multiplicando **fuelCap** por **mpg**. Já que cada objeto de tipo **Vehicle** tem sua própria cópia de **fuelCap** e **mpg**, quando **range( )** é chamado, o cálculo da autonomia usa as cópias dessas variáveis pertencentes ao objeto chamador.

O método **range( )** termina quando sua chave de fechamento é alcançada. Isso faz o controle do programa ser transferido novamente para o chamador.

Agora, olhe atentamente para a seguinte linha de código que fica dentro de **main()**:

```
| minivan.range();
```

Essa instrução chama o método **range( )** em **minivan**. Isto é, ela chama **range( )** em relação ao objeto **minivan**, usando o nome do objeto seguido do operador ponto. Quando um método é chamado, o controle do programa é transferido para ele. Quando o método termina, o controle é transferido novamente para o chamador e a execução é retomada na linha de código posterior à chamada.

Nesse caso, a chamada a **minivan.range( )** exibe a autonomia do veículo definido por **minivan**. Da mesma forma, a chamada a **sportscar.range( )** exibe a autonomia do veículo definido por **sportscar**. Sempre que **range( )** é chamado, exibe a autonomia do objeto especificado.

Há algo muito importante a se observar dentro do método **range( )**: as variáveis de instância **fuelCap** e **mpg** são referenciadas diretamente, sem ser precedidas por um nome de objeto ou o operador ponto. Quando um método usa uma variável de instância definida por sua classe, ele faz isso diretamente, sem referência explícita a um objeto e sem o uso do operador ponto. Se você pensar bem, é fácil de entender. Um método sempre é chamado em relação a algum objeto de sua classe. Uma vez que essa chamada ocorre, o objeto é conhecido. Portanto, dentro de um método, não precisamos especificar o objeto novamente. Ou seja, as variáveis **fuelCap** e **mpg** existentes dentro de **range( )** referenciam implicitamente cópias dessas variáveis encontradas no objeto em que **range( )** é chamado.

## RETORNANDO DE UM MÉTODO

Em geral, há duas condições que fazem um método retornar – a primeira, como o método **range( )** do exemplo anterior mostra, é quando a chave de fechamento do método é alcançada. A segunda é quando uma instrução **return** é executada. Há duas formas de **return** – uma para uso em métodos **void** (métodos que não retornam valor) e outra para o retorno de valores. A primeira forma será examinada aqui. A próxima seção explicará como retornar valores.

Você pode causar o encerramento imediato de um método **void** usando esta forma de **return**:

```
return;
```

Quando essa instrução é executada, o controle do programa volta para o chamador, saltando qualquer código restante no método. Por exemplo, considere o seguinte método:

```
void myMeth() {
 for(int i=0; i < 10; i++) {
 if(i == 5) return; // para em 5
 System.out.println(i);
 }
}
```

Aqui, o laço **for** só será executado de 0 a 5, porque quando **i** for igual a 5, o método retornará. Podemos ter várias instruções **return** em um método, principalmente se houver duas ou mais saídas dele. Por exemplo:

```
void myMeth() {
 // ...
 if(done) return;
 // ...
 if(error) return;
 // ...
}
```

Nesse caso, o método retorna ao terminar ou se um erro ocorrer. No entanto, tome cuidado, porque a existência de muitos pontos de saída em um método pode desestruturar o código; por isso evite usá-los casualmente. Um método bem projetado tem pontos de saída bem definidos.

Resumindo: um método **void** pode retornar de uma entre duas maneiras – sua chave de fechamento é alcançada ou uma instrução **return** é executada.

## RETORNANDO UM VALOR

Embora não sejam raros métodos com tipo de retorno **void**, a maioria dos métodos retorna um valor. Na verdade, a possibilidade de retornar um valor é um dos recursos mais úteis dos métodos. Você já viu um exemplo de valor de retorno: quando usamos a função **sqrt()** para obter a raiz quadrada.

Os valores de retorno são usados para vários fins em programação. Em alguns casos, como em **sqrt()**, o valor de retorno contém o resultado de um cálculo. Em outros, pode simplesmente indicar sucesso ou falha. Em outros ainda, pode conter um código de status. Qualquer que seja a finalidade, o uso de valores de retorno é parte integrante da programação Java.

Os métodos retornam um valor para a rotina chamadora usando esta forma de **return**:

```
return valor;
```

Aqui, *valor* é o valor retornado. Essa forma de **return** só pode ser usada com métodos que tenham tipo de retorno diferente de **void**. Além disso, um método não **void** deve retornar um valor usando essa versão de **return**.

Você pode usar um valor de retorno para melhorar a implementação de **range()**. Em vez de exibir a autonomia, uma abordagem melhor seria **range()** calcular

a autonomia e retornar o valor. Uma das vantagens dessa abordagem é o valor poder ser usado em outros cálculos. O exemplo a seguir modifica **range()** para retornar a autonomia em vez de exibi-la.

```
// Usa um valor de retorno.

class Vehicle {
 int passengers; // número de passageiros
 int fuelCap; // capacidade de armazenamento de combustível em galões
 int mpg; // consumo de combustível em milhas por galão

 // Retorna a autonomia.
 int range() {
 return mpg * fuelCap; ← Retorna a autonomia de um determinado veículo.
 }
}

class RetMeth {
 public static void main(String[] args) {
 Vehicle minivan = new Vehicle();
 Vehicle sportscar = new Vehicle();

 int range1, range2;

 // atribui valores a campos de minivan
 minivan.passengers = 7;
 minivan.fuelCap = 16;
 minivan.mpg = 21;

 // atribui valores a campos de sportscar
 sportscar.passengers = 2;
 sportscar.fuelCap = 14;
 sportscar.mpg = 12;

 // obtém as autonomias
 range1 = minivan.range(); [] Atribui o valor retornado a uma variável.
 range2 = sportscar.range(); [] Atribui o valor retornado a uma variável.

 System.out.println("Minivan can carry " + minivan.passengers +
 " with range of " + range1 + " miles");

 System.out.println("Sportscar can carry " + sportscar.passengers +
 " with range of " + range2 + " miles");
 }
}
```

A saída é mostrada aqui:

```
Minivan can carry 7 with range of 336 miles
Sportscar can carry 2 with range of 168 miles
```

No programa, observe que quando **range()** é chamado, ele é inserido no lado direito de uma instrução de atribuição. À esquerda, temos uma variável que receberá o valor retornado por **range()**. Portanto, após

```
| range1 = minivan.range();
```

ser executado, a autonomia do objeto **minivan** será armazenada em **range1**. Em outras palavras, a chamada a **minivan.range()** resulta no cálculo da autonomia de **minivan**. O resultado é então retornado via instrução **return** de **range()**. Em seguida, esse valor é atribuído a **range1**. Logo, o valor retornado por **range()** passa a ser o valor da chamada de método. Nesse caso, é como se você tivesse escrito **range1 = 336** porque 336 é o valor retornado por **minivan.range()**.

Observe que agora **range()** tem tipo de retorno **int**, ou seja, retornará um valor inteiro para o chamador. O tipo de retorno de um método é importante porque o tipo de dado retornado deve ser compatível com o tipo de retorno especificado. Logo, se você quiser que um método retorne dados de tipo **double**, seu tipo de retorno deve ser **double**.

Embora o programa anterior esteja correto, não foi escrito de maneira tão eficiente quanto poderia ser. Especificamente, não precisamos das variáveis **range1** ou **range2**. Uma chamada a **range()** pode ser usada na instrução **println()** diretamente, como mostrado aqui:

```
| System.out.println("Minivan can carry " + minivan.passengers +
| " with range of " + minivan.range() + " miles");
```

Nesse caso, quando **println()** for executado, **minivan.range()** será chamado automaticamente e seu valor de retorno será passado para **println()**. Além disso, você pode usar uma chamada a **range()** sempre que a autonomia de um objeto **Vehicle** for necessária. Por exemplo, esta instrução compara as autonomias de dois veículos:

```
| if(v1.range() > v2.range()) System.out.println("v1 has greater range");
```

## USANDO PARÂMETROS

Podemos passar um ou mais valores para um método quando ele é chamado. Lembre-se de que um valor passado para um método se chama *argumento*. Dentro do método, a variável que recebe o argumento se chama *parâmetro*. Os parâmetros são declarados dentro dos parênteses que vêm após o nome do método. A sintaxe de declaração de parâmetros é a mesma usada para variáveis. Um parâmetro faz parte do escopo de seu método e, exceto pela tarefa especial de receber um argumento, ele age como qualquer variável local.

Aqui está um exemplo simples que usa um parâmetro. Dentro da classe **ChkNum**, o método **isEven()** retorna **true** quando o valor passado é par. Caso contrário, retorna **false**. Logo, **isEven()** tem tipo de retorno **boolean**.

```
// Um exemplo simples que usa um parâmetro.

class ChkNum {

 // Retorna true se x for par.
```

```

boolean isEven(int x) { ←————— Aqui, x é um parâmetro inteiro de isEven().
 if((x%2) == 0) return true;
 else return false;
}

class ParmDemo {
 public static void main(String[] args) {
 ChkNum e = new ChkNum();
 ↓————— Passa argumentos para isEven().
 if(e.isEven(10)) System.out.println("10 is even.");
 if(e.isEven(9)) System.out.println("9 is even.");
 if(e.isEven(8)) System.out.println("8 is even.");
 }
}

```

Esta é a saída produzida pelo programa:

```

10 is even.
8 is even.

```

No programa, **isEven()** é chamado três vezes e a cada vez um valor diferente é passado. Examinemos esse processo em detalhes. Primeiro, observe como **isEven()** é chamado. O argumento é especificado entre os parênteses. Quando **isEven()** é chamado pela primeira vez, recebe o valor 10. Portanto, quando ele começa a ser executado, o parâmetro **x** recebe o valor 10. Na segunda chamada, 9 é o argumento e, então, **x** tem o valor 9. Na terceira chamada, o argumento é 8, que é o valor que **x** recebe. Logo, o valor passado como argumento quando **isEven()** é chamado é o valor recebido por seu parâmetro, **x**.

Agora que vimos um parâmetro em ação, um conceito importante precisa ser mencionado. Os parâmetros são essenciais na programação Java porque proporcionam um meio de fornecermos os dados com os quais um método operará. Isso permite que os métodos sejam mais úteis, e mais genéricos. Por exemplo, se o método **isEven()** que acabamos de mostrar não tivesse um parâmetro e só retornasse o resultado da verificação do valor 19, ele teria uso muito limitado. No entanto, com a passagem do valor a ser verificado, a utilidade de **isEven()** aumentou bastante porque agora ele pode verificar qualquer valor. Logo, com a parametrização de um método, permitimos que ele aborde o caso *geral* em vez de apenas uma situação *específica*.

Um ponto-chave que devemos entender sobre a passagem de argumentos é que o tipo do argumento deve ser compatível com o tipo do parâmetro que o recebe. Isso significa, por exemplo, que seria um erro tentar chamar **isEven()** com um argumento **boolean**. Como um valor **boolean** não pode ser convertido em um valor **int**, o compilador Java relatará um erro e não compilará o programa.

Um método pode ter mais de um parâmetro. Simplesmente declare cada parâmetro, separando um do outro com uma vírgula. Por exemplo, a classe **Factor**

define um método chamado **isFactor( )** que determina se o primeiro parâmetro é um fator do segundo.

```
class Factor {
 // Retorna true se a for fator de b.
 boolean isFactor(int a, int b) { ← Este método tem dois parâmetros.
 if((b % a) == 0) return true;
 else return false;
 }
}

class IsFact {
 public static void main(String[] args) {
 Factor x = new Factor();
 ↓ Passa dois argumentos para isFactor().
 if(x.isFactor(2, 20)) System.out.println("2 is factor");
 if(x.isFactor(3, 20)) System.out.println("this won't be displayed");
 }
}
```

Observe que quando **isFactor( )** é chamado, os argumentos também são separados por vírgulas.

Quando são usados vários parâmetros, cada parâmetro especifica seu próprio tipo, que pode diferir dos outros. Por exemplo, isto é perfeitamente válido:

```
int myMeth(int a, double b, float c) {
// ...
```

### Adicionando um método parametrizado a Vehicle

Você pode usar um método parametrizado para adicionar um novo recurso à classe **Vehicle**: a possibilidade de calcular a quantidade de combustível necessária para cobrir uma determinada distância. Esse novo método se chama **fuelNeeded( )**. Ele recebe o número de milhas que você quer percorrer e retorna quantos galões de gasolina são necessários. O método **fuelNeeded( )** é definido assim:

```
double fuelNeeded(int miles) {
 return (double) miles / mpg;
}
```

Observe que esse método retorna um valor de tipo **double**. Isso é útil, já que a quantidade de combustível necessária para cobrir uma determinada distância pode não ser um número inteiro. A classe **Vehicle** completa com a inclusão de **fuelNeeded( )** é mostrada aqui:

```
/*
 Adiciona um método parametrizado que calcula o
 combustível necessário para cobrir uma determinada distância.
*/
class Vehicle {
```

```

int passengers; // número de passageiros
int fuelCap; // capacidade de armazenamento de combustível em galões
int mpg; // consumo de combustível em milhas por galão

// Retorna a autonomia.
// int range() {
// return mpg * fuelCap;
// }

// Calcula o combustível necessário para cobrir uma determinada
// distância.
double fuelNeeded(int miles) {
 return (double) miles / mpg;
}

class CompFuel {
 public static void main(String[] args) {
 Vehicle minivan = new Vehicle();
 Vehicle sportscar = new Vehicle();
 double gallons;
 int dist = 252;

 // atribui valores a campos de minivan
 minivan.passengers = 7;
 minivan.fuelCap = 16;
 minivan.mpg = 21;

 // atribui valores a campos de sportscar
 sportscar.passengers = 2;
 sportscar.fuelCap = 14;
 sportscar.mpg = 12;

 gallons = minivan.fuelNeeded(dist);

 System.out.println("To go " + dist + " miles minivan needs " +
 gallons + " gallons of fuel.");

 gallons = sportscar.fuelNeeded(dist);

 System.out.println("To go " + dist + " miles sportscar needs " +
 gallons + " gallons of fuel.");
 }
}

```

A saída do programa é a seguinte:

```
| To go 252 miles minivan needs 12.0 gallons of fuel.
| To go 252 miles sportscar needs 21.0 gallons of fuel.
```

## Verificação do progresso

1. Quando uma variável de instância ou um método deve ser acessado por intermédio de uma referência de objeto com o uso do operador ponto? Quando uma variável ou método pode ser usado diretamente?
2. Explique a diferença entre um argumento e um parâmetro.
3. Explique as duas maneiras pelas quais um método pode retornar para seu chamador.

### TENTE ISTO 4-1 Criando uma classe de ajuda

`HelpClassDemo.java`

Se alguém tentasse resumir a essência da classe em uma frase, ela poderia ser esta: uma classe encapsula funcionalidade. É claro que às vezes o truque é saber onde uma funcionalidade termina e outra começa. Como regra geral, você vai querer que suas classes sejam os blocos de construção do aplicativo final. Para que isso ocorra, cada classe deve representar uma única unidade funcional executando ações claramente delimitadas. Portanto, você vai querer que suas classes sejam tão pequenas quanto possível – mas não menores do que isso! Ou seja, classes que contêm funcionalidade demais confundem e desestruturam o código, mas classes que contêm muito pouca funcionalidade são fragmentadas. Qual é o equilíbrio? É nesse ponto que a ciência da programação se torna a *arte* de programar. Felizmente, a maioria dos programadores descobre que esse ato de equilíbrio se torna mais fácil com a experiência.

Para começar a ganhar essa experiência, você converterá o sistema de ajuda da seção Tente isto 3-3 do capítulo anterior em uma classe Help. Vejamos por que essa é uma boa ideia. Em primeiro lugar, o sistema de ajuda define apenas uma unidade lógica. Ele simplesmente exibe a sintaxe das instruções de controle Java. Logo, sua funcionalidade é compacta e bem definida. Em segundo lugar, inserir a ajuda em uma classe é uma abordagem esteticamente amigável. Sempre que você quiser oferecer o sistema de ajuda a um usuário, só terá de instanciar um objeto de sistema de ajuda. Para concluir, já que a ajuda está encapsulada, pode ser atualizada ou alterada sem causar efeitos colaterais indesejados nos programas que a usarem.

#### Respostas:

1. Quando uma variável de instância for acessada por um código que não faz parte da classe em que ela foi definida, isso deve ser feito por intermédio de um objeto, com o uso do operador ponto. Quando uma variável de instância for acessada por um código que faz parte de sua classe, ela pode ser referenciada diretamente. O mesmo se aplica aos métodos.
2. Um argumento é um valor que é passado para um método quando este é chamado. Um parâmetro é uma variável definida por um método que recebe o valor do argumento.
3. Podemos fazer um método retornar com o uso da instrução **return**. Se o método tiver tipo de retorno **void**, também retornará quando sua chave de fechamento for alcançada. Métodos não **void** devem retornar um valor, logo, o retorno pela chegada na chave de fechamento não é uma opção.

### PASSO A PASSO

1. Crie um novo arquivo chamado **HelpClassDemo.java**. Para evitar digitação, você pode copiar o arquivo da seção Tente isto 3-3, **Help3.java**, para **HelpClassDemo.java**.
2. Para converter o sistema de ajuda em uma classe, primeiro você deve determinar precisamente o que compõe o sistema. Por exemplo, em **Help3.java**, há código para a exibição de um menu, a inserção da escolha do usuário, a procura de uma resposta válida e a exibição de informações sobre o item selecionado. O programa também entra em laço até a letra q ser pressionada. Se você pensar bem, está claro que o menu, a procura por uma resposta válida e a exibição de informações são parte integrante do sistema de ajuda, mas o modo como a entrada do usuário é obtida e decidir se solicitações repetidas devem ser processadas não são. Logo, você criará uma classe que exibirá as informações de ajuda, exibirá o menu e procurará uma seleção válida. Seus métodos se chamarão **helpOn()**, **showMenu()** e **isValid()**, respectivamente.
3. Crie o método **helpOn()** como mostrado aqui:

```
// Exibe a ajuda.
void helpOn(int what) {
 switch(what) {
 case '1':
 System.out.println("The if:\n");
 System.out.println("if(condition) statement;");
 System.out.println("else statement;");
 break;
 case '2':
 System.out.println("The switch:\n");
 System.out.println("switch(expression) {");
 System.out.println(" case constant:");
 System.out.println(" statement sequence");
 System.out.println(" break;");
 System.out.println(" // ...");
 System.out.println("}");
 break;
 case '3':
 System.out.println("The for:\n");
 System.out.print("for(init; condition; iteration)");
 System.out.println(" statement;");
 break;
 case '4':
 System.out.println("The while:\n");
 System.out.println("while(condition) statement;");
 break;
 case '5':
 System.out.println("The do-while:\n");
 System.out.println("do {");
 System.out.println(" statement;");
 System.out.println("} while (condition);");
 break;
 case '6':
 System.out.println("The break:\n");
 }
}
```

```

 System.out.println("break; or break label;");
 break;
 case '7':
 System.out.println("The continue:\n");
 System.out.println("continue; or continue label;");
 break;
 }
 System.out.println();
}

```

**4.** Em seguida, crie o método **showMenu()**:

```

// Exibe o menu.
void showMenu() {
 System.out.println("Help on:");
 System.out.println(" 1. if");
 System.out.println(" 2. switch");
 System.out.println(" 3. for");
 System.out.println(" 4. while");
 System.out.println(" 5. do-while");
 System.out.println(" 6. break");
 System.out.println(" 7. continue\n");
 System.out.print("Choose one (q to quit): ");
}

```

**5.** Crie o método **isValid()**, mostrado aqui:

```

// Retorna true se ch for uma seleção válida.
boolean isValid(int ch) {
 if(ch < '1' | ch > '7' & ch != 'q') return false;
 else return true;
}

```

**6.** Reúna os métodos anteriores na classe **Help**, listada abaixo:

```

class Help {

 // Exibe a ajuda.
 void helpOn(int what) {
 switch(what) {
 case '1':
 System.out.println("The if:\n");
 System.out.println("if(condition) statement;");
 System.out.println("else statement;");
 break;
 case '2':
 System.out.println("The switch:\n");
 System.out.println("switch(expression) { ");
 System.out.println(" case constant:");
 System.out.println(" statement sequence");
 System.out.println(" break; ");
 System.out.println(" // ...");
 System.out.println("} ");
 break;
 }
 }
}

```

```
case '3':
 System.out.println("The for:\n");
 System.out.print("for(init; condition; iteration)");
 System.out.println(" statement;");
 break;
case '4':
 System.out.println("The while:\n");
 System.out.println("while(condition) statement;");
 break;
case '5':
 System.out.println("The do-while:\n");
 System.out.println("do {");
 System.out.println(" statement;");
 System.out.println("} while (condition);");
 break;
case '6':
 System.out.println("The break:\n");
 System.out.println("break; or break label;");
 break;
case '7':
 System.out.println("The continue:\n");
 System.out.println("continue; or continue label;");
 break;
}
System.out.println();
}

// Exibe o menu.
void showMenu() {
 System.out.println("Help on:");
 System.out.println(" 1. if");
 System.out.println(" 2. switch");
 System.out.println(" 3. for");
 System.out.println(" 4. while");
 System.out.println(" 5. do-while");
 System.out.println(" 6. break");
 System.out.println(" 7. continue\n");
 System.out.print("Choose one (q to quit): ");
}

// Retorna true se ch for uma seleção válida.
boolean isValid(int ch) {
 if(ch < '1' | ch > '7' & ch != 'q') return false;
 else return true;
}
}
```

7. Para concluir, reescreva o método **main()** da seção Tente isto 3-3, para que ele use a nova classe **Help**. Chame essa classe de **HelpClassDemo.java**. A listagem completa de **HelpClassDemo.java** é mostrada a seguir:

```
/*
Tente isto 4-1

Converte o sistema de ajuda da seção Tente isto 3-3 em
uma classe Help.

*/

class Help {

 // Exibe a ajuda.
 void helpOn(int what) {
 switch(what) {
 case '1':
 System.out.println("The if:\n");
 System.out.println("if(condition) statement;");
 System.out.println("else statement;");
 break;
 case '2':
 System.out.println("The switch:\n");
 System.out.println("switch(expression) { ");
 System.out.println(" case constant:");
 System.out.println(" statement sequence");
 System.out.println(" break; ");
 System.out.println(" // ...");
 System.out.println("} ");
 break;
 case '3':
 System.out.println("The for:\n");
 System.out.print("for(init; condition; iteration)");
 System.out.println(" statement;");
 break;
 case '4':
 System.out.println("The while:\n");
 System.out.println("while(condition) statement;");
 break;
 case '5':
 System.out.println("The do-while:\n");
 System.out.println("do {");
 System.out.println(" statement;");
 System.out.println("} while (condition);");
 break;
 case '6':
 System.out.println("The break:\n");
 System.out.println("break; or break label;");
 break;
 }
 }
}
```

```
 case '7':
 System.out.println("The continue:\n");
 System.out.println("continue; or continue label;");
 break;
 }
 System.out.println();
 }

// Exibe o menu.
void showMenu() {
 System.out.println("Help on:");
 System.out.println(" 1. if");
 System.out.println(" 2. switch");
 System.out.println(" 3. for");
 System.out.println(" 4. while");
 System.out.println(" 5. do-while");
 System.out.println(" 6. break");
 System.out.println(" 7. continue\n");
 System.out.print("Choose one (q to quit): ");
}

// Retorna true se ch for uma seleção válida.
boolean isValid(int ch) {
 if(ch < '1' | ch > '7' & ch != 'q') return false;
 else return true;
}

class HelpClassDemo {
 public static void main(String[] args)
 throws java.io.IOException {

 char choice, ignore;
 Help hlpobj = new Help();

 for(;;) {
 do {
 hlpobj.showMenu();

 choice = (char) System.in.read();

 do {
 ignore = (char) System.in.read();
 } while(ignore != '\n');

 } while(!hlpobj.isValid(choice));
 }
 }
}
```

```
 if(choice == 'q') break;

 System.out.println("\n");

 hlpobj.helpOn(choice);
 }
}
```

Quando você testar o programa, verá que ele é funcionalmente o mesmo de antes. A vantagem dessa abordagem é que agora você tem um componente de sistema de ajuda que pode ser reutilizado sempre que necessário.

# CONSTRUTORES

Nos exemplos anteriores, as variáveis de instância de cada objeto **Vehicle** tiveram de ser configuradas manualmente com o uso de uma sequência de instruções, como:

```
| minivan.passengers = 7;
| minivan.fuelCap = 16;
| minivan.mpg = 21;
```

Uma abordagem como essa nunca seria usada em um código Java escrito profissionalmente. Além de ser propensa a erros (você pode se esquecer de configurar um dos campos), há uma maneira melhor de executar essa tarefa: o construtor.

Um *construtor* inicializa um objeto quando este é criado. Ele tem o mesmo nome de sua classe e é sintaticamente semelhante a um método. No entanto, os construtores não têm um tipo de retorno explícito. Normalmente, usamos um construtor para fornecer valores iniciais para as variáveis de instância definidas pela classe ou para executar algum outro procedimento de inicialização necessário à criação de um objeto totalmente formado.

Todas as classes têm construtores, mesmo quando não definimos um, porque Java fornece automaticamente um construtor padrão. No entanto, quando definimos nosso próprio construtor, o construtor padrão não é mais usado.

Aqui está um exemplo simples que usa um construtor:

```
// Um construtor simples.

class MyClass {
 int x;

 MyClass() { ← Este é o construtor de MyClass.
 x = 10;
 }
}

class ConsDemo {
 public static void main(String[] args) {
```

```

 MyClass t1 = new MyClass();
 MyClass t2 = new MyClass();

 System.out.println(t1.x + " " + t2.x);
}
}

```

Nesse exemplo, o construtor de **MyClass** é

```

MyClass() {
 x = 10;
}

```

Esse construtor atribui o valor 10 à variável de instância **x** de **MyClass**. Ele é chamado por **new** quando um objeto é criado. Por exemplo, na linha

```
| MyClass t1 = new MyClass();
```

o construtor **MyClass()** é chamado e o objeto resultante é atribuído a **t1**, com **t1.x** recebendo o valor 10. O mesmo ocorre para **t2**. Após a construção, **t2.x** tem o valor 10. Portanto, a saída do programa é

```
| 10 10
```

## CONSTRUTORES PARAMETRIZADOS

No exemplo anterior, um construtor sem parâmetros foi usado. Embora isso seja adequado em algumas situações, quase sempre você precisará de um construtor que aceite um ou mais parâmetros. Os parâmetros são adicionados a um construtor do mesmo modo como são adicionados a um método: apenas declare-os dentro de parênteses após o nome do construtor. Por exemplo, aqui, **MyClass** recebe um construtor parametrizado:

```

// Um construtor parametrizado.

class MyClass {
 int x;

 MyClass(int i) { ←———— Este construtor tem um parâmetro.
 x = i;
 }
}

class ParmConsDemo {
 public static void main(String[] args) {
 MyClass t1 = new MyClass(10);
 MyClass t2 = new MyClass(88);

 System.out.println(t1.x + " " + t2.x);
 }
}

```

A saída desse programa é mostrada abaixo:

```
| 10 88
```

Nessa versão do programa, o construtor **MyClass( )** define um parâmetro chamado **i**, que é usado para inicializar a variável de instância **x**. Logo, quando a linha

```
| MyClass t1 = new MyClass(10);
```

é executada, o valor 10 é passado para **i**, que é então atribuído a **x**.

### Pergunte ao especialista

**P** Se eu não inicializar uma variável de instância, que valor ela terá?

**R** Se você não inicializar uma variável de instância, ela receberá um valor padrão. Para tipos numéricos, o padrão é zero. Para tipos de referência, o padrão é **null**, que indica que nenhum objeto está sendo referenciado, e para variáveis **boolean**, o padrão é **false**.

### Adicionando um construtor à classe Vehicle

Podemos melhorar a classe **Vehicle** adicionando um construtor que inicialize automaticamente os campos **passengers**, **fuelCap** e **mpg** quando um objeto for construído. Preste bastante atenção em como os objetos **Vehicle** são criados.

```
// Adiciona um construtor.

class Vehicle {
 int passengers; // número de passageiros
 int fuelCap; // capacidade de armazenamento de combustível em galões
 int mpg; // consumo de combustível em milhas por galão

 // Esse é um construtor para Vehicle.
 Vehicle(int p, int f, int m) { ←———— Construtor de Vehicle.
 passengers = p;
 fuelCap = f;
 mpg = m;
 }

 // Retorna a autonomia.
 int range() {
 return mpg * fuelCap;
 }

 // Calcula o combustível necessário para cobrir uma determinada distância.
 double fuelNeeded(int miles) {
 return (double) miles / mpg;
 }
}
```

```

class VehConsDemo {
 public static void main(String[] args) {

 // constrói veículos completos
 Vehicle minivan = new Vehicle(7, 16, 21);
 Vehicle sportscar = new Vehicle(2, 14, 12);

 double gallons;
 int dist = 252;

 gallons = minivan.fuelNeeded(dist);

 System.out.println("To go " + dist + " miles minivan needs " +
 gallons + " gallons of fuel.");

 gallons = sportscar.fuelNeeded(dist);

 System.out.println("To go " + dist + " miles sportscar needs " +
 gallons + " gallons of fuel.");
 }
}

```

Tanto **minivan** quanto **sportscar** são inicializadas pelo construtor **Vehicle()** quando são criadas. Cada objeto é inicializado como especificado nos parâmetros de seu construtor. Por exemplo, na linha a seguir,

```
| Vehicle minivan = new Vehicle(7, 16, 21);
```

os valores 7, 16 e 21 são passados para o construtor **Vehicle()** quando **new** cria o objeto. Logo, a cópia de **passengers**, **fuelCap** e **mpg** de **minivan** conterá os valores 7, 16 e 21, respectivamente. A saída desse programa é igual à da versão anterior.

### Verificação do progresso

1. Quando um construtor é executado?
2. Um construtor tem um tipo de retorno?

**Respostas:**

1. Um construtor é executado quando um objeto de sua classe é instanciado. O construtor é usado para inicializar o objeto que está sendo criado.
2. Não.

## O OPERADOR new REVISITADO

Agora que você sabe mais sobre as classes e seus construtores, examinemos com detalhes o operador **new**. No contexto de uma atribuição, o operador **new** tem esta forma geral:

```
var-classe = new nome-classe(lista-arg);
```

Aqui, *var-classe* é uma variável do tipo de classe que está sendo criada. *Nome-classe* é o nome da classe que está sendo instanciada. O nome da classe seguido por uma lista de argumentos entre parênteses (que pode estar vazia) especifica o construtor da classe. Se uma classe não definir seu próprio construtor, **new** usará o construtor padrão fornecido por Java. Logo, **new** pode ser usado para criar um objeto de qualquer tipo de classe. O operador **new** retorna uma referência ao objeto recém-criado, que (nesse caso) é atribuído a *var-classe*.

Já que a memória é finita, é possível que **new** não consiga alocar memória para um objeto por não existir memória suficiente. Se isso ocorrer, haverá uma exceção de tempo de execução. (Conheceremos as exceções no Capítulo 10.) Para os exemplos de programa deste livro, não precisamos nos preocupar em ficar sem memória, mas temos que considerar essa possibilidade em programas do mundo real que escrevermos.

### Pergunte ao especialista

**P** Por que não preciso usar **new** para variáveis de tipos primitivos, como **int** ou **float**?

**R** Os tipos primitivos da linguagem Java não são implementados como objetos. Em vez disso, devido a preocupações com a eficiência, eles são implementados como variáveis “comuns”. Uma variável de tipo primitivo contém diretamente o valor que damos a ela. Como explicado, uma variável de referência contém uma referência ao objeto. Essa camada de endereçamento indireto (e outros recursos dos objetos) adiciona sobrecarga a um objeto que é evitada por um tipo primitivo.

## COLETA DE LIXO E FINALIZADORES

Como vimos, os objetos são alocados dinamicamente em uma porção de memória livre com o uso do operador **new**. Conforme explicado, a memória não é infinita e o espaço livre pode se extinguir. Portanto, é possível que **new** falhe por não haver memória livre suficiente para a criação do objeto desejado. Logo, um componente-chave de qualquer esquema de alocação dinâmica é a recuperação de memória livre de objetos não usados, com a disponibilização dessa memória para realocações subsequentes. Em algumas linguagens de programação, a liberação de memória já alocada é realizada manualmente. (Por exemplo, em C++, usamos o operador **delete** para liberar memória que foi alocada.) No entanto, Java usa uma abordagem diferente, que apresenta menos problemas: a *coleta de lixo*.

O sistema de coleta de lixo de Java reclama objetos automaticamente – ocorrendo de maneira transparente em segundo plano, sem nenhuma intervenção do programador. Funciona assim: quando não existe referência a um objeto, ele não é mais considerado necessário e a memória ocupada é liberada. Essa memória reciclada pode então ser usada para uma alocação subsequente.

A coleta de lixo só ocorre esporadicamente durante a execução do programa. Ela não ocorrerá só porque existem um ou mais objetos que não são mais usados. A título de eficiência, geralmente o coletor de lixo só é executado quando duas condições são atendidas: há objetos a serem reciclados e há necessidade de reciclar-los. Lembre-se, a coleta de lixo é demorada, por isso, o sistema de tempo de execução Java só a executa quando apropriado. Portanto, não temos como saber exatamente quando ela ocorrerá.

## O método `finalize()`

É possível definir um método, conhecido como *finalizador*, para ser chamado imediatamente antes da destruição final de um objeto pelo coletor de lixo. Esse método se chama **`finalize()`** e pode ser usado em casos muito específicos para assegurar que um objeto seja totalmente eliminado. Por exemplo, você pode usar **`finalize()`** para assegurar que algum recurso do sistema não gerenciado pelo tempo de execução Java seja liberado apropriadamente. Embora a grande maioria dos programas Java não precise de finalizadores, o tópico será abordado aqui como complemento e porque um finalizador será usado na demonstração do mecanismo Java de coleta de lixo.

Para adicionar um finalizador a uma classe, você deve definir o método **`finalize()`**. O sistema de tempo de execução Java chamará esse método sempre que estiver para reciclar um objeto dessa classe. Dentro do método **`finalize()`**, você especificará as ações que devem ser executadas antes de um objeto ser destruído.

O método **`finalize()`** tem a seguinte forma geral:

```
protected void finalize()
{
 // parte onde entra o código de finalização
}
```

Aqui, a palavra-chave **protected** é um modificador que controla o acesso a **`finalize()`** por um código definido fora de sua classe. Esse e outros modificadores de acesso serão explicados no Capítulo 6.

É importante entender que **`finalize()`** é chamado imediatamente antes da coleta de lixo. Ele não é chamado quando um objeto sai de escopo, por exemplo. Ou seja, não temos como saber quando – ou até mesmo se – **`finalize()`** será executado. Por exemplo, se o programa terminar antes da coleta de lixo ocorrer, **`finalize()`** não será executado. Portanto, ele deve ser usado somente como um procedimento “reserva” para assegurar o tratamento apropriado de algum recurso ou para aplicações de uso especial, e não como um artifício para o programa usar em sua operação normal. Resumindo, **`finalize()`** é um método especializado que raramente é usado.

## TENTE ISTO 4-2 Demonstre a coleta de lixo

GCDemo.java

Como a coleta de lixo é executada esporadicamente em segundo plano, não é fácil vê-la em ação. No entanto, uma maneira de fazê-lo é com o uso do método **finalize()**. Lembre-se de que **finalize()** é chamado quando um objeto está para ser reciclado. Como explicado, os objetos não são necessariamente reciclados assim que não são mais necessários. Em vez disso, o coletor de lixo espera até poder executar sua coleta de maneira eficiente, geralmente quando há muitos objetos não usados. Logo, para demonstrar a coleta de lixo via método **finalize()**, temos de criar e destruir vários objetos – e é exatamente o que faremos neste projeto.

### PASSO A PASSO

1. Crie um novo arquivo chamado **GCDemo.java**.
2. Crie a classe **MyClass** mostrada aqui:

```
class MyClass {
 int x;

 MyClass(int i) {
 x = i;
 }

 // Chamado quando o objeto é reciclado.
 protected void finalize() {
 System.out.println("Finalizing " + x);
 }

 // Gera um objeto que é imediatamente abandonado.
 void generate(int i) {
 MyClass o = new MyClass(i);
 }
}
```

O construtor configura a variável de instância **x** com um valor conhecido. Nesse exemplo, **x** é usada como uma identificação de objeto. O método **finalize()** exibe o valor de **x** quando um objeto é reciclado. De especial interesse é **generator()**, método que cria e então abandona imediatamente um objeto **MyClass**. Isso torna esse objeto sujeito à coleta de lixo. Você verá como ele é usado na próxima etapa.

3. Crie a classe **GCDemo**, mostrada abaixo:

```
class GCDemo {
 public static void main(String[] args) {
 MyClass ob = new MyClass(0);

 /* Agora, gere um grande número de objetos. Em algum
 momento, a coleta de lixo ocorrerá.
```

```

 Nota: você pode ter de aumentar o número
 de objetos gerados para forçar
 a coleta de lixo. */

 for(int count=1; count < 1000000; count++)
 ob.generate(count);
 }
}
}

```

Essa classe cria um objeto **MyClass** inicial chamado **ob**. Em seguida, usando **ob**, ela cria 1.000.000 de objetos chamando **generator()** em **ob**. Como resultado, 1.000.000 de objetos são criados e descartados. Em vários pontos no meio do processo, a coleta de lixo ocorrerá. Muitos fatores vão influenciar exatamente com que frequência ou quando, como a quantidade inicial de memória livre e a carga atual de tarefas do sistema operacional. No entanto, em algum momento, você começará a ver as mensagens geradas por **finalize()**. Se não conseguir vê-las, tente aumentar o número de objetos que estão sendo gerados elevando a contagem no laço **for**.

4. Aqui está o programa **GCDemo.java** inteiro:

```

/*
 Tente isto 4-2

 Demonstra a coleta de lixo e o método finalize().
*/

class MyClass {
 int x;

 MyClass(int i) {
 x = i;
 }

 // Chamado quando o objeto é reciclado.
 protected void finalize() {
 System.out.println("Finalizing " + x);
 }

 // Gera um objeto que é imediatamente abandonado.
 void generate(int i) {
 MyClass o = new MyClass(i);
 }
}

class GCDemo {
 public static void main(String[] args) {
 MyClass ob = new MyClass(0);

 /* Agora, gere um grande número de objetos. Em
 algum momento, a coleta de lixo ocorrerá.
 */
 }
}

```

Nota: você pode ter de aumentar o número de objetos gerados para forçar a coleta de lixo. \*/

```
 for(int count=1; count < 1000000; count++)
 ob.generate(count);
 }
}
```

## A PALAVRA-CHAVE this

Antes de concluirmos este capítulo, é necessário introduzir **this**. Quando um método é chamado, ele recebe automaticamente um *argumento implícito*, que é uma referência ao objeto chamador (isto é, o objeto em que o método é chamado). Essa referência se chama **this**. Para entender **this**, primeiro considere um programa que cria uma classe chamada **Power** para calcular o resultado de um número elevado a alguma potência inteira:

```

 System.out.println(y.b + " raised to the " + y.e +
 " power is " + y.getPwr());
 System.out.println(z.b + " raised to the " + z.e +
 " power is " + z.getPwr());
 }
}

```

A saída do programa é mostrada abaixo:

```

4.0 raised to the 2 power is 16.0
2.5 raised to the 1 power is 2.5
5.7 raised to the 0 power is 1.0

```

Em cada caso, o valor da variável **b** de **Power** é elevado à potência passada para **e**.

Como você sabe, dentro de um método, os outros membros de uma classe podem ser acessados diretamente, sem nenhuma qualificação de objeto ou classe. Logo, dentro de **getPwr()**, a instrução

```
| return val;
```

significa que a cópia de **val** associada ao objeto chamador será retornada. No entanto, a mesma instrução também pode ser escrita assim:

```
| return this.val;
```

Aqui, **this** referencia o objeto em que **getPwr()** foi chamado. Portanto, **this.val** referencia a cópia de **val** pertencente a esse objeto. Por exemplo, se na instrução anterior **getPwr()** tivesse sido chamado em **x**, **this** referenciaria **x**. Escrever a instrução sem usar **this** é apenas uma forma de abreviar.

Esta é a classe **Power** inteira escrita com o uso da referência **this**:

```

class Power {
 double b;
 int e;
 double val;

 Power(double base, int exp) {
 this.b = base;
 this.e = exp;

 this.val = 1;
 if(exp==0) return;
 for(; exp>0; exp--) this.val = this.val * base;
 }

 double getPwr() {
 return this.val;
 }
}

```

Se no programa anterior você usasse essa versão de **Power**, os mesmos resultados seriam produzidos, porque as duas versões de **Power** são funcionalmente equivalentes.

Na verdade, nenhum programador de Java criaria **Power** como acabamos de mostrar, porque nada é ganho e a forma padrão é mais fácil. No entanto, **this** tem algumas aplicações importantes. Por exemplo, a sintaxe Java permite que o nome de um parâmetro ou de uma variável local seja igual ao nome de uma variável de instância. Quando isso ocorre, o nome local *oculta* a variável de instância. Você pode ganhar acesso à variável de instância oculta referenciando-a com **this**. Por exemplo, o código a seguir é uma maneira sintaticamente válida de escrever o construtor **Power()**.

```
Power(double b, int e) {
 this.b = b; This referencia a variável de
 this.e = e; instância b e não o parâmetro.

 val = 1;
 if(e==0) return;
 for(; e>0; e--) val = val * b;
}
```

Nessa versão, os nomes dos parâmetros são iguais aos nomes das variáveis de instância, ocultando-as. No entanto, **this** é usada para “expor” as variáveis de instância.

---

## EXERCÍCIOS

1. Qual é a diferença entre uma classe e um objeto?
2. Como uma classe é definida?
3. Cada objeto tem sua própria cópia de quê?
4. Usando duas instruções separadas, mostre como declarar uma variável de nome **counter** de uma classe chamada **MyCounter** e atribuir a ela um novo objeto dessa classe.
5. Mostre como um método chamado **myMeth()** será declarado se tiver um tipo de retorno **double** e dois parâmetros **int** chamados **a** e **b**.
6. Como um método deve retornar se um valor for retornado?
7. Que nome tem um construtor?
8. O que **new** faz?
9. O que é coleta de lixo e como ela funciona? O que é **finalize()**?
10. O que é **this**?
11. Um construtor pode ter um ou mais parâmetros?
12. Se um método não retornar um valor, qual deve ser seu tipo de retorno?
13. Crie uma classe **Die** com uma variável de instância inteira chamada **sideUp**. Forneça a ela um construtor, um método **getSideUp()** que retorne o valor de **sideUp** e um método **void roll()** que altere **sideUp** para um valor aleatório de 1 a 6. (Para ver como gerar um inteiro aleatório entre 1 e 6, examine o último

exercício do Capítulo 2.) Em seguida, crie uma classe **DieDemo** com um método principal que gere dois objetos **Die**, jogue-os e exiba a soma dos dois lados superiores.

14. Crie uma classe **Card** que represente a carta de um baralho. Ela deve ter uma variável de instância **int** chamada **rank** e uma variável **char** chamada **suit**. Forneça um construtor com dois parâmetros para inicializar as duas variáveis de instância e um método **getSuit()** e um **getRank()** que retornem os valores das variáveis. Agora, crie uma classe **CardTester** com um método principal que gere cinco **Cards** para compor um full house (isto é, três das cartas têm um mesmo valor e as outras duas têm outros dois valores iguais) e exiba os valores e os naipes dos cinco **Cards** usando os métodos **getSuit()** e **getRank()**.
15. Suponha que você tenha uma classe **MyClass** com uma variável de instância **x**. O que será exibido pelo fragmento de código a seguir? Explique sua resposta.

```
 MyClass c1 = new MyClass();
c1.x = 3;
MyClass c2 = c1;
c2.x = 4;
System.out.println(c1.x);
```

16. Suponha que uma classe tenha uma variável de instância **x** e um método com uma variável local **x**.
  - A. Se **x** for usada em um cálculo no corpo do método, que variável será referenciada?
  - B. Suponha que você precise que adicionar a variável local **x** à variável de instância **x** no corpo do método. Como o faria?
17. O método a seguir tem uma falha (na verdade, devido a essa falha ele não será compilado). Qual é a falha?

```
void displayAbsX(int x) {
 if (x > 0) {
 System.out.println(x);
 return;
 }
 else {
 System.out.println(-x);
 return;
 }
 System.out.println("Done");
}
```

18. Crie um método **max()** que tenha dois parâmetros inteiros **x** e **y** e retorne o maior dos dois.
19. Crie um método **max()** que tenha três parâmetros inteiros **x**, **y** e **z** e retorne o maior dos três. Faça-o de duas maneiras: uma vez usando uma escada **if-else-if** e a outra usando instruções **if** aninhadas.

20. Suponha que uma classe tenha que calcular um valor e depois exibi-lo. A fim de modularizar o código, o programador quer criar um novo método na classe para tratar dessa tarefa. Seria melhor o novo método calcular e exibir o valor ou apenas calcular e retornar o valor, deixando sua exibição para o código que o chama?
21. Encontre todos os erros (se houver algum) na declaração de classe a seguir:

```
Class MyCla$$ {
 integer x = 3.0;
 boolean b == false

 //construtor
 MyClass(boolean b) { b = b; }

 int doIt() {}

 int don'tDoIt() { return this; }
}
```

22. Crie uma classe **Swapper** com duas variáveis de instância inteiras **x** e **y** e um construtor com dois parâmetros que inicialize as duas variáveis. Inclua também três métodos: um método **getX()** que retorne **x**, um método **getY()** que retorna **y** e um método **void swap()** que troque os valores de **x** e **y**. Em seguida, crie uma classe **SwapperDemo** que teste todos os métodos.
23. Suponha que você esteja criando um programa de genealogia. Uma classe útil seria **Person**, na qual cada pessoa da árvore genealógica seria representada por um objeto **Person**. Liste pelo menos cinco variáveis de instância cuja inclusão nessa classe seria apropriada. Não se preocupe com o tipo das variáveis de instância.
24. Crie uma classe **USMoney** com duas variáveis de instância inteiras **dollars** e **cents**. Adicione um construtor com dois parâmetros para a inicialização de um objeto **USMoney**. O construtor deve verificar se o valor de **cents** está entre 0 e 99 e, se não estiver, transferir alguns dos **cents** para a variável **dollars** para que ela passe a ter entre 0 e 99. Adicione um método **plus** à classe que use um objeto **USMoney** como parâmetro. Ele deve criar e retornar um novo objeto **USMoney** representando a soma do objeto cujo método **plus()** está sendo chamado mais o parâmetro, sem modificar os valores dos dois objetos existentes. Também deve assegurar que o valor da variável de instância **cents** do novo objeto esteja entre 0 e 99. Por exemplo, se **x** for um objeto **USMoney** com 5 dólares e 80 cents e se **y** for um objeto **USMoney** com 1 dólar e 90 cents, **x.plus(y)** retornará um novo objeto **USMoney** com 7 dólares e 70 cents. Crie também uma classe **USMoneyDemo** que teste a classe **USMoney**.
25. Crie uma classe **Date** com três variáveis de instância inteiras chamadas **day**, **month** e **year**. Ela tem um construtor com três parâmetros para a inicialização das variáveis de instância e tem um método chamado **daysSinceJan1()**. A classe calcula e retorna o número de dias desde 1º de janeiro do mesmo ano, incluindo o dia 1º de janeiro e o dia do objeto **Date**. Por exemplo, se **day** for um

objeto **Date** com **day** = 1, **month** = 3 e **year** = 2000, a chamada **date.daysSinceJan1()** deve retornar 61 desde que haja 61 dias entre as datas de 1º de janeiro de 2000 e 1º de março de 2000, incluindo 1º de janeiro e 1º de março. Inclua uma classe **Date Demo** que teste a classe **Date**. Não esqueça os anos bissextos.

26. Qual é a diferença, caso haja, entre as duas implementações a seguir do método **doIt()**?

```
void doIt(int x) {
 if(x > 0)
 System.out.println("Pos");
 else
 System.out.println("Neg");
}

void doIt(int x) {
 if(x > 0) {
 System.out.println("Pos");
 return;
 }
 System.out.println("Neg");
}
```

# 5

# Mais tipos de dados e operadores

## PRINCIPAIS HABILIDADES E CONCEITOS

- Entender e criar arrays
- Criar arrays multidimensionais
- Criar arrays irregulares
- Saber a sintaxe alternativa de declaração de arrays
- Atribuir referências de arrays
- Usar o membro de array **length**
- Usar o laço **for** de estilo for-each
- Trabalhar com strings
- Aplicar argumentos de linha de comando
- Usar os operadores bitwise
- Aplicar o operador **?**

Este capítulo voltará ao assunto dos tipos de dados e operadores Java. Ele discutirá arrays, o tipo **String**, os operadores bitwise e o operador ternário **?**. Também abordará o laço **for** Java de estilo for-each. Ao avançarmos, os argumentos de linha de comando serão descritos.

## ARRAYS

É comum em programação haver um grupo de variáveis relacionadas. Por exemplo, você poderia querer uma lista das temperaturas máximas diárias do mês de abril. Embora pudesse usar 30 variáveis separadas para esse fim, essa solução seria ao mesmo tempo deselegante e ineficiente. Pense em como seria difícil calcular a média das temperaturas máximas. Primeiro você teria que somar todas as 30 variáveis individuais e então dividir esse valor por 30. Isso geraria uma expressão muito longa e monótona. Também geraria uma solução inflexível. Felizmente, Java dá suporte a uma maneira muito melhor de tratarmos grupos de variáveis relacionadas: o *array*.

Um *array* é um conjunto de variáveis do mesmo tipo, referenciadas por um nome comum. Em Java, os arrays podem ter uma ou mais dimensões, embora o array unidimensional seja o mais popular. Os arrays oferecem um meio conveniente de agrupar variáveis relacionadas. Por exemplo, usar um array para armazenar as tem-

peraturas máximas diárias durante um mês é muito melhor do que usar 30 valores separados. Outras coisas que você poderia armazenar em um array seriam uma lista de preços de ações, os títulos de sua coleção de livros de programação ou um inventário de produtos.

A principal vantagem de um array é que ele organiza os dados de tal forma que é fácil tratá-los. Por exemplo, se você tiver um array contendo uma lista de saldos bancários, será fácil calcular o valor total percorrendo-o. Os arrays também organizam os dados de forma que eles possam ser facilmente classificados.

Embora os arrays Java possam ser usados da mesma forma que os arrays de outras linguagens de programação, eles têm um atributo especial: são implementados como objetos. Essa é uma das razões para termos adiado a discussão dos arrays até os objetos serem introduzidos. Na implementação de arrays como objetos, muitas vantagens importantes são obtidas e uma delas, que não é menos importante, é que os arrays não usados podem ser alvo da coleta de lixo.

## Arrays unidimensionais

Um array unidimensional é uma lista de variáveis relacionadas. Essas listas são comuns em programação. Por exemplo, você pode usar um array unidimensional para armazenar os números de conta dos usuários ativos em uma rede. Para declarar um array unidimensional, você usará esta forma geral:

```
tipo[] nome-array = new tipo[tamanho];
```

Aqui, *tipo* declara o *tipo de elemento* do array. (O tipo de elemento também é chamado de tipo base.) O tipo de elemento determina o tipo de dado de cada elemento contido no array. O número de elementos que o array conterá é determinado por *tamanho*. Como os arrays são implementados como objetos, a criação de um array é um processo de duas etapas. Primeiro, você declara uma variável de referência de array. Depois, aloca memória para o array, atribuindo uma referência dessa memória à variável de array. Portanto, os arrays Java são alocados dinamicamente com o uso do operador **new**.

Veja um exemplo. A linha a seguir cria um array **int** de 10 elementos e o vincula a uma variável de referência de array chamada **sample**:

```
| int[] sample = new int[10];
```

Essa declaração funciona como uma declaração de objeto. A variável **sample** contém uma referência à memória alocada por **new**. Essa memória é suficientemente grande para conter 10 elementos de tipo **int**. Como ocorre com os objetos, é possível dividir a declaração anterior em duas. Por exemplo:

```
| int[] sample;
| sample = new int[10];
```

Nesse caso, quando **sample** é criada, ela não referencia um objeto físico. Só após a segunda instrução ser executada, **sample** é vinculada a um array.

Um elemento individual de um array é acessado com o uso de um índice. Um *índice* descreve a posição de um elemento dentro de um array. Em Java, todos os arrays têm zero como o índice de seu primeiro elemento. Já que a variável **sample** tem 10 elementos, ela tem valores de índice que vão de 0 a 9. Para indexar um array, de-

vemos especificar o número do elemento desejado, inserido em colchetes. Portanto, o primeiro elemento de **sample** é **sample[0]** e o último é **sample[9]**.

Um ponto importante a ser entendido é que cada elemento do array é usado da mesma forma que uma variável “comum”. Por exemplo, você pode atribuir um valor a um elemento, como mostrado aqui:

```
| sample[0] = 3;
```

Após essa instrução ser executada, o primeiro elemento de **sample** terá o valor 3. Você pode obter o valor de um elemento para usar em uma expressão, como mostrado a seguir:

```
| 2 * sample[0]
```

Aqui, se **sample[0]** contiver 3, o resultado da expressão anterior será 6.

O programa a seguir demonstra **sample** carregando-o com os números de 0 a 9 e exibindo seu conteúdo:

```
// Demonstra um array unidimensional.
class ArrayDemo {
 public static void main(String[] args) {
 int[] sample = new int[10];
 int i;

 for(i = 0; i < 10; i = i+1) <-- sample[i] = i;
 for(i = 0; i < 10; i = i+1) <-- System.out.println("This is sample[" + i + "]: " + sample[i]);
 }
}
```

Os arrays são indexados a partir de zero.

A saída do programa é mostrada aqui:

```
This is sample[0]: 0
This is sample[1]: 1
This is sample[2]: 2
This is sample[3]: 3
This is sample[4]: 4
This is sample[5]: 5
This is sample[6]: 6
This is sample[7]: 7
This is sample[8]: 8
This is sample[9]: 9
```

Conceptualmente, o array **sample** tem a seguinte aparência:

|            |            |            |            |            |            |            |            |            |            |
|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|
| 0          | 1          | 2          | 3          | 4          | 5          | 6          | 7          | 8          | 9          |
| sample [0] | sample [1] | sample [2] | sample [3] | sample [4] | sample [5] | sample [6] | sample [7] | sample [8] | sample [9] |

Os arrays são comuns em programação porque nos permitem lidar facilmente com grandes quantidades de variáveis relacionadas. Por exemplo, o programa abaixo encontra o valor mínimo e máximo do array **nums** percorrendo o array com o uso de um laço **for**:

```
// Encontra o valor mínimo e máximo de um array.
class MinMax {
 public static void main(String[] args) {
 int[] nums = new int[10];
 int min, max;

 nums[0] = 99;
 nums[1] = -10;
 nums[2] = 100123;
 nums[3] = 18;
 nums[4] = -978;
 nums[5] = 5623;
 nums[6] = 463;
 nums[7] = -9;
 nums[8] = 287;
 nums[9] = 49;

 min = max = nums[0];
 for(int i=1; i < 10; i++) {
 if(nums[i] < min) min = nums[i];
 if(nums[i] > max) max = nums[i];
 }
 System.out.println("min and max: " + min + " " + max);
 }
}
```

A saída do programa é mostrada a seguir:

```
| min and max: -978 100123
```

Observe como o programa funciona. Primeiro ele fornece tanto a **min** quanto a **max** o valor de **nums[0]**. Em seguida, percorre o array **nums**, um elemento de cada vez, começando com o segundo elemento. Dentro do laço, se o valor de **nums[i]** for menor do que **min**, ele passará a ser o novo valor mínimo. Da mesma forma, se o valor de **nums[i]** for maior do que **max**, ele será o novo valor máximo. O processo continua até todos os elementos de **nums** serem testados. Como resultado, quando o laço terminar, **min** terá o menor valor do array e **max** terá o maior.

No programa anterior, o array **nums** recebeu valores manualmente, usando 10 instruções de atribuição separadas. Embora isso esteja perfeitamente correto, há uma maneira mais fácil de fazê-lo. Os arrays podem ser inicializados quando são criados. A forma geral de inicialização de um array unidimensional é esta:

```
tipo[] nome-array = {val1, val2, val3,..., valN};
```

Aqui, os valores iniciais são especificados por *val1* até *valN*. Eles são atribuídos em sequência, da esquerda para a direita, em ordem de índice. Java aloca automaticamente um array grande o suficiente para conter os inicializadores especificados. Não há

necessidade de usar o operador **new** explicitamente. Por exemplo, esta é uma maneira melhor de escrever o programa **MinMax**:

```
// Usa inicializadores de array.
class MinMax2 {
 public static void main(String[] args) {
 int[] nums = { 99, -10, 100123, 18, -978,
 5623, 463, -9, 287, 49 }; ← Inicializadores de array.
 int min, max;

 min = max = nums[0];
 for(int i=1; i < 10; i++) {
 if(nums[i] < min) min = nums[i];
 if(nums[i] > max) max = nums[i];
 }
 System.out.println("Min and max: " + min + " " + max);
 }
}
```

Os limites do array são impostos rigorosamente em Java; é um erro de tempo de execução estar abaixo ou acima da extremidade de um array. Se quiser confirmar isso por sua própria conta, teste o programa a seguir, o qual intencionalmente excede um array:

```
// Demonstra uma situação que excede um array.
class ArrayErr {
 public static void main(String[] args) {
 int[] sample = new int[10];
 int i;

 // gera a transposição de um array
 for(i = 0; i < 100; i++)
 sample[i] = i;
 }
}
```

Assim que **i** alcançar 10, uma **ArrayIndexOutOfBoundsException** será gerada e o programa será encerrado.

### TENTE ISTO 5-1 Classificando um array

`Bubble.java`

Uma vez que um array unidimensional organiza os dados em uma lista linear que pode ser indexada, é fácil classificá-lo. Neste projeto, você aprenderá uma maneira simples de classificar um array. Como deve saber, há vários algoritmos de classificação. Há a classificação rápida, a classificação por troca e a classificação de shell, para citar apenas três. No entanto, a mais conhecida, simples e fácil de entender se chama classificação por bolha. Embora a classificação por bolha não seja muito eficiente – na verdade, geralmente seu desempenho é inaceitável para a classificação de arrays grandes –, ela pode ser usada de maneira eficaz na clas-

sificação de arrays pequenos. Porém, será usada aqui porque oferece um exemplo excelente que demonstra o poder dos arrays.

### PASSO A PASSO

1. Crie um arquivo chamado **Bubble.java**.
2. A classificação por bolha obtém seu nome da maneira como executa a operação de classificação. Ela usa a comparação repetida e, se necessário, a troca de elementos adjacentes do array. Nesse processo, valores pequenos se movem em direção a uma extremidade e os maiores em direção à outra. A classificação por bolha funciona percorrendo várias vezes o array e trocando os elementos que estiverem fora do lugar quando preciso.

Aqui está o código que forma a base da classificação por bolha. O array que está sendo classificado se chama **nums**.

```
// Esta é a classificação por bolha.
for(a=1; a < size; a++)
 for(b=size-1; b >= a; b--) {
 if(nums[b-1] > nums[b]) { // se fora de ordem
 // troca elementos
 t = nums[b-1];
 nums[b-1] = nums[b];
 nums[b] = t;
 }
 }
```

Observe que a classificação se baseia em dois laços **for**. O laço interno verifica os elementos adjacentes do array, procurando elementos fora de ordem. Quando um par de elementos fora de ordem é encontrado, os dois elementos são trocados. A cada passagem, o menor dos elementos restantes se move para o local apropriado. O laço externo faz esse processo se repetir até o array inteiro ser classificado.

3. Aqui está o programa **Bubble** inteiro:

```
/*
Tente isto 5-1

Demonstra a classificação por bolha.
*/

class Bubble {
 public static void main(String[] args) {
 int[] nums = { 99, -10, 100123, 18, -978,
 5623, 463, -9, 287, 49 };
 int a, b, t;
 int size;

 size = 10; // número de elementos a serem classificados

 // exibe o array original
 System.out.print("Original array is:");
 for(int i=0; i < size; i++)
```

```

 System.out.print(" " + nums[i]);
 System.out.println();

 // Esta é a classificação por bolha.
 for(a=1; a < size; a++) {
 for(b=size-1; b >= a; b--) {
 if(nums[b-1] > nums[b]) { // se fora de ordem
 // troca elementos
 t = nums[b-1];
 nums[b-1] = nums[b];
 nums[b] = t;
 }
 }

 // exibe o array classificado
 System.out.print("Sorted array is:");
 for(int i=0; i < size; i++)
 System.out.print(" " + nums[i]);
 System.out.println();
 }
 }
}

```

A saída do programa é mostrada abaixo:

```

Original array is: 99 -10 100123 18 -978 5623 463 -9 287 49
Sorted array is: -978 -10 -9 18 49 99 287 463 5623 100123

```

4. Como mencionado, embora a classificação por bolha possa ser útil em arrays pequenos, ela não é eficiente quando usada em arrays maiores. Um dos melhores algoritmos de classificação de uso geral é a classificação rápida (*Quicksort*). No entanto, a classificação rápida depende de recursos Java que você ainda não aprendeu.

## ARRAYS MULTIDIMENSIONAIS

Apesar do array unidimensional ser o mais usado em programação, os arrays multidimensionais (arrays de duas ou mais dimensões) certamente não são raros. Em Java, o array multidimensional é um array composto por arrays.

### Arrays bidimensionais

A forma mais simples de array multidimensional é o array bidimensional. Um array bidimensional é, na verdade, uma lista de arrays unidimensionais. Ele é semelhante a quando criamos uma tabela de dados, com os dados organizados por linha e coluna. Um dado individual é acessado com a especificação da posição de sua linha e coluna.

Para declarar um array bidimensional, você deve especificar o tamanho das duas dimensões. Por exemplo, aqui, **table** é declarado para ser um array bidimensional de tipo **int** e tamanho 10 por 20:

```
| int[][] table = new int[10][20];
```

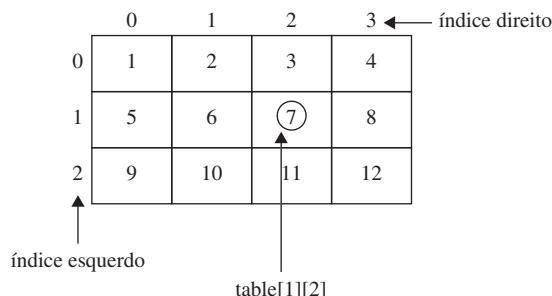
Preste atenção na declaração. Diferentemente de outras linguagens de computador, que usam vírgulas para separar as dimensões do array, Java insere cada dimensão em seu próprio conjunto de colchetes. Da mesma forma, para acessar o ponto 3, 5 do array **table**, usaríamos **table[3][5]**.

No próximo exemplo, um array bidimensional é carregado com os números de 1 a 12

```
// Demonstra um array bidimensional.
class TwoD {
 public static void main(String[] args) {
 int t, i;
 int[][] table = new int[3][4];

 for(t=0; t < 3; ++t) {
 for(i=0; i < 4; ++i) {
 table[t][i] = (t*4)+i+1;
 System.out.print(table[t][i] + " ");
 }
 System.out.println();
 }
 }
}
```

Nesse exemplo, **table[0][0]** terá o valor 1, **table[0][1]** o valor 2, **table[0][2]** o valor 3, e assim por diante. O valor de **table[2][3]** será 12. Conceitualmente, o array ficaria parecido com o mostrado na Figura 5-1. Observe como os dados estão organizados na forma tabular.



**Figura 5-1** Visão conceitual do array **table** criado pelo programa **TwoD**.

## Arrays irregulares

Quando alocamos memória para um array multidimensional, só temos de especificar a memória da primeira dimensão (a da extrema esquerda). As outras dimensões podem ser alocadas separadamente. Por exemplo, o código a seguir aloca memória para

a primeira dimensão do array **table** quando este é declarado. A segunda dimensão é alocada manualmente.

```
int[][] table = new int[3][];
table[0] = new int[4];
table[1] = new int[4];
table[2] = new int[4];
```

Embora não haja vantagens em alocar individualmente os arrays da segunda dimensão nessa situação, pode haver em outras. Por exemplo, quando alocamos as dimensões separadamente, não precisamos alocar o mesmo número de elementos para cada índice. Uma vez que os arrays multidimensionais são implementados como arrays compostos por arrays, temos o controle do tamanho de cada array.

Por exemplo, suponhamos que estivéssemos escrevendo um programa para armazenar o número de passageiros que pegam um ônibus no aeroporto. Se o ônibus faz o transporte dez vezes ao dia durante a semana e duas vezes ao dia no sábado e no domingo, poderíamos usar o array **riders** mostrado no programa abaixo para armazenar as informações. Observe que o tamanho da segunda dimensão para os primeiros cinco índices é 10 e para os dois últimos índices é 2.

```
// Aloca manualmente segundas dimensões de tamanhos diferentes.
class Ragged {
 public static void main(String[] args) {
 int[][] riders = new int[7][];
 riders[0] = new int[10]; ←
 riders[1] = new int[10];
 riders[2] = new int[10];
 riders[3] = new int[10];
 riders[4] = new int[10]; ←
 riders[5] = new int[2]; ←
 riders[6] = new int[2]; ←
 // Aqui, as segundas dimensões têm 10 elementos.
 // Mas, aqui, elas têm 2 elementos.

 int i, j;

 // forja alguns dados
 for(i=0; i < 5; i++)
 for(j=0; j < 10; j++)
 riders[i][j] = i + j + 10;
 for(i=5; i < 7; i++)
 for(j=0; j < 2; j++)
 riders[i][j] = i + j + 10;

 System.out.println("Riders per trip during the week:");
 for(i=0; i < 5; i++) {
 for(j=0; j < 10; j++)
 System.out.print(riders[i][j] + " ");
 System.out.println();
 }
 System.out.println();
```

```

System.out.println("Riders per trip on the weekend:");
for(i=5; i < 7; i++) {
 for(j=0; j < 2; j++)
 System.out.print(riders[i][j] + " ");
 System.out.println();
}
}
}

```

A saída do programa é mostrada aqui:

```

Riders per trip during the week:
10 11 12 13 14 15 16 17 18 19
11 12 13 14 15 16 17 18 19 20
12 13 14 15 16 17 18 19 20 21
13 14 15 16 17 18 19 20 21 22
14 15 16 17 18 19 20 21 22 23

Riders per trip on the weekend:
15 16
16 17

```

O uso de arrays multidimensionais irregulares (ou desiguais) não é apropriado a todas as situações. Com frequência, um array bidimensional regular é a melhor opção. No entanto, os arrays irregulares podem ser muito eficazes em alguns casos, como no exemplo que acabamos de mostrar. Se precisarmos de um array bidimensional muito grande preenchido esparsamente (isto é, um array em que poucos elementos sejam usados), o array irregular pode ser a solução perfeita.

## Arrays de três ou mais dimensões

Java permite arrays com mais de duas dimensões. Aqui está a forma geral de uma declaração de array multidimensional:

```
tipo [] []...[] nome = new tipo[tamanho1][tamanho2]... [tamanhoN];
```

Por exemplo, a declaração a seguir cria um array tridimensional inteiro de  $4 \times 10 \times 3$ .

```
int [] [] [] multidim = new int [4] [10] [3] ;
```

Dado esse array, a instrução abaixo atribui o valor 10 ao elemento 2, 7, 1:

```
multidim[2] [7] [1] = 10;
```

## Inicializando arrays multidimensionais

Um array multidimensional pode ser inicializado com a inserção da lista de inicializadores de cada dimensão dentro de seu próprio conjunto de chaves. Por exemplo, a forma geral da inicialização de um array bidimensional é mostrada abaixo:

```

tipo [] [] nome_array =
 {val, val, val, ..., val},
 {val, val, val, ..., val},
 .
 .
 .

```

```
{val, val, val, ..., val},
};
```

Aqui, *val* indica um valor de inicialização. Cada bloco interno designa uma linha. Dentro de cada linha, o primeiro valor será armazenado na primeira posição do subarray, o segundo valor na segunda posição e assim por diante. Observe que vírgulas separam os blocos inicializadores e um ponto e vírgula vem após a chave de fechamento.

Por exemplo, o programa a seguir inicializa um array chamado **sqrs** com os números de 1 a 10 e seus quadrados:

```
// Inicializa um array bidimensional.
class Squares {
 public static void main(String[] args) {
 int[][] sqrs = {
 { 1, 1 }, ←
 { 2, 4 }, ←
 { 3, 9 }, ←
 { 4, 16 }, ←
 { 5, 25 }, ←
 { 6, 36 }, ←
 { 7, 49 }, ←
 { 8, 64 }, ←
 { 9, 81 }, ←
 { 10, 100 } ←
 };
 int i, j;

 for(i=0; i < 10; i++) {
 for(j=0; j < 2; j++)
 System.out.print(sqrs[i][j] + " ");
 System.out.println();
 }
 }
}
```

Observe como cada linha tem seu próprio conjunto de inicializadores.

Esta é a saída do programa:

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

## Verificação do progresso

1. Como cada dimensão é especificada em arrays multidimensionais?
2. Em um array bidimensional, que é um array composto por arrays, cada array pode ter um tamanho diferente?
3. Como os arrays multidimensionais são inicializados?

## SINTAXE ALTERNATIVA PARA A DECLARAÇÃO DE ARRAYS

Há uma segunda forma que pode ser usada na declaração de um array:

*tipo nome-var[ ];*

Aqui, os colchetes vêm depois do nome da variável de array e não do especificador de tipo. Por exemplo, as duas declarações a seguir são equivalentes:

```
| int counter[] = new int[3];
| int[] counter = new int[3];
```

As declarações abaixo também são equivalentes:

```
| char table[][] = new char[3][4];
| char[][] table = new char[3][4];
```

Essa declaração alternativa nos permite declarar variáveis do mesmo tipo, sejam elas de array ou não, na mesma declaração. Por exemplo,

```
| int alpha, beta[], gamma;
```

Aqui, **alpha** e **gamma** são de tipo **int**, mas **beta** é um array de inteiros.

## ATRIBUINDO REFERÊNCIAS DE ARRAYS

Como ocorre com os demais objetos, quando atribuímos uma variável de referência de array a outra variável de referência de array, estamos simplesmente alterando o objeto que a variável referencia. Não estamos criando uma cópia do array, nem copiando o conteúdo de um array para o outro. Por exemplo, considere o programa a seguir:

```
// Atribuindo variáveis de referência de array.
class AssignARef {
 public static void main(String[] args) {
 int i;
```

---

Respostas:

1. Cada dimensão é especificada dentro de seu próprio conjunto de colchetes.
2. Sim.
3. Os arrays multidimensionais são inicializados com a inserção dos inicializadores de cada subarray dentro de seu próprio conjunto de chaves.

```
int[] nums1 = new int[10];
int[] nums2 = new int[10];

for(i=0; i < 10; i++)
 nums1[i] = i;

for(i=0; i < 10; i++)
 nums2[i] = -i;

System.out.print("Here is nums1: ");
for(i=0; i < 10; i++)
 System.out.print(nums1[i] + " ");
System.out.println();

System.out.print("Here is nums2: ");
for(i=0; i < 10; i++)
 System.out.print(nums2[i] + " ");
System.out.println();

nums2 = nums1; // agora nums2 referencia nums1 ← Atribui uma referência
 de array.

System.out.print("Here is nums2 after assignment: ");
for(i=0; i < 10; i++)
 System.out.print(nums2[i] + " ");
System.out.println();

// opera com o array nums1 por intermédio de nums2
nums2[3] = 99;

System.out.print("Here is nums1 after change through nums2: ");
for(i=0; i < 10; i++)
 System.out.print(nums1[i] + " ");
System.out.println();
}

}
```

A saída do programa é mostrada aqui:

```
Here is nums1: 0 1 2 3 4 5 6 7 8 9
Here is nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Here is nums2 after assignment: 0 1 2 3 4 5 6 7 8 9
Here is nums1 after change through nums2: 0 1 2 99 4 5 6 7 8 9
```

Esse exemplo cria dois arrays e dá a eles valores iniciais. Logo, no início, **nums1** e **nums2** referenciam arrays separados e distintos. Em seguida, **nums1** é atribuída a **nums2**. Após essa atribuição, tanto **nums1** quanto **nums2** referenciam o mesmo array. Portanto, a alteração do array por intermédio de **nums2** (como faz o exemplo) também afeta o array referenciado por **nums1** porque ambas referenciam o mesmo array.

## USANDO O MEMBRO length

Já que os arrays são implementados como objetos, cada array tem uma variável de instância **length** associada que contém o número de elementos que ele pode conter. Em outras palavras, **length** contém o tamanho do array. Aqui está um programa que demonstra essa propriedade:

```
// Usa o membro de array length.
class LengthDemo {
 public static void main(String[] args) {
 int[] list = new int[10];
 int[] nums = { 1, 2, 3 };
 int[][] table = { // uma tabela de tamanho variável
 {1, 2, 3},
 {4, 5},
 {6, 7, 8, 9}
 };

 System.out.println("length of list is " + list.length);
 System.out.println("length of nums is " + nums.length);
 System.out.println("length of table is " + table.length);
 System.out.println("length of table[0] is " + table[0].length);
 System.out.println("length of table[1] is " + table[1].length);
 System.out.println("length of table[2] is " + table[2].length);
 System.out.println();

 // usa length para inicializar list
 for(int i=0; i < list.length; i++) ←
 list[i] = i * i;
 }

 System.out.print("Here is list: ");
 // agora usa length para exibir list
 for(int i=0; i < list.length; i++) ←
 System.out.print(list[i] + " ");
 System.out.println();
}
}
```

Esse programa exibe a saída abaixo:

```
length of list is 10
length of nums is 3
length of table is 3
length of table[0] is 3
length of table[1] is 2
length of table[2] is 4

Here is list: 0 1 4 9 16 25 36 49 64 81
```

Preste atenção na maneira como **length** é usado com o array bidimensional **table**. Como explicado, um array bidimensional é um array composto por arrays. Portanto, quando a expressão

```
|table.length|
```

é usada, ela obtém o número de arrays armazenado em **table**, que nesse caso é 3. Para obter o tamanho de qualquer array individual de **table**, você usará uma expressão como

```
|table[0].length|
```

que, aqui, obtém o tamanho do primeiro array.

Outra coisa a ser notada em **LengthDemo** é a maneira como **list.length** é usado pelos laços **for** para controlar o número de iterações. Uma vez que cada array carrega com ele seu tamanho, você pode usar essa informação em vez de controlar manualmente o tamanho de um array. Lembre-se de que o valor de **length** não tem ligação com o número de elementos que estão sendo usados. Ele contém o número de elementos que o array pode conter.

A inclusão do membro **length** simplifica os algoritmos ao tornar mais fácil – e seguro – executar certos tipos de operações com arrays. Por exemplo, o programa abaixo usa **length** para copiar um array para outro ao mesmo tempo em que impede que o limite do array seja excedido e que ocorra erro durante a execução.

```
// Usa a variável length para ajudar na cópia de um array.
class ACopy {
 public static void main(String[] args) {
 int i;
 int [] nums1 = new int [10];
 int [] nums2 = new int [10];

 for(i=0; i < nums1.length; i++)
 nums1[i] = i; Usa length para comparar tamanhos de arrays.

 // copia nums1 para nums2
 if(nums2.length >= nums1.length) {
 for(i = 0; i < nums1.length; i++)
 nums2[i] = nums1[i];

 for(i=0; i < nums2.length; i++)
 System.out.print(nums2[i] + " ");
 }
 }
}
```

Aqui, **length** ajuda a desempenhar duas funções importantes. Em primeiro lugar, é usado para confirmar se o array de destino é suficientemente grande para armazenar o conteúdo do array de origem. Em segundo lugar, fornece a condição de encerramento do laço **for** que faz a cópia. É claro que, nesse exemplo simples, os tamanhos dos arrays podem ser facilmente conhecidos, mas essa mesma abordagem pode ser aplicada a situações mais desafiadoras.

Mais uma coisa sobre **length**: ele é somente de leitura. Portanto, não pode receber um novo valor. Ou seja, você não pode alterar o tamanho de um array mudando o valor de **length**.

## TENTE ISTO 5-2 Uma classe de pilha simples

`SimpleStackDemo.java`

Um dos elementos básicos da programação é a *estrutura de dados*. As estruturas de dados só serão examinadas com mais detalhes na Parte III, quando as suportadas pela biblioteca Java forem descritas, mas você já deve conhecer o bastante sobre Java para ser apresentado ao conceito.

Em sua essência, uma estrutura de dados fornece um meio de organizar dados. A estrutura de dados mais simples é o array. Como você acabou de ver, um array é uma lista linear que dá suporte ao acesso aleatório aos seus elementos. Com frequência, os arrays são usados como base para estruturas de dados mais sofisticadas, uma delas sendo a pilha. Uma *pilha* é uma lista em que os elementos só podem ser acessados na ordem último a entrar, primeiro a sair (LIFO). Logo, uma pilha é como uma pilha de pratos em uma mesa – o primeiro de baixo para cima é o último a ser usado. Além disso, um novo prato só pode ser adicionado ao topo da pilha, e quando um prato é necessário, ele deve ser removido do topo.

Uma coisa que torna estruturas de dados como as pilhas particularmente interessantes é que elas combinam o *armazenamento* de informações com os métodos que as *acessam*. Essa combinação, obviamente, é uma ótima opção para o encapsulamento dentro de uma classe. Na verdade, é assim que normalmente essas estruturas de dados são implementadas.

Neste projeto você implementará uma pilha simples. A pilha foi escolhida por duas razões. Em primeiro lugar, uma pilha representa um dos exemplos fundamentais da programação orientada a objetos porque mostra uma aplicação compacta, porém realista. Em segundo lugar, já que nossa implementação é baseada em um array, ela fornece outro exemplo do tratamento de arrays. Aqui criaremos uma implementação inicial da pilha. Capítulos subsequentes expandirão e melhorarão seus recursos à medida que introduzirmos novos elementos Java. Para simplificar, a pilha desenvolvida armazena caracteres, mas pode ser facilmente adaptada para armazenar outros tipos de dados.

Em geral, as pilhas dão suporte a duas operações básicas, normalmente chamadas de *push* e *pop*. Cada operação push insere um novo elemento no topo da pilha. Cada operação pop recupera o elemento do topo da pilha. Logo, um novo elemento é adicionado à pilha pela inserção dele no topo e um elemento é removido quando é tirado de lá. Nenhum outro acesso aos elementos de uma pilha é suportado. Por exemplo, você não pode remover um elemento do meio de uma pilha. Além disso, quando um elemento é extraído da pilha, ele é eliminado. Em outras palavras, quando um elemento é recuperado, não pode ser recuperado novamente.

Uma pilha tem duas condições limite: cheia e vazia. Ela está cheia quando não há espaço disponível para armazenar outro item, e vazia quando todos os seus elementos foram removidos.

Uma última coisa: como você verá na Parte III, a biblioteca Java fornece uma classe de pilha poderosa e completa que faz parte do Collections Framework. Já que nossa classe de pilha é muito mais simples, a chamaremos de **SimpleStack**.

#### PASSO A PASSO

1. Crie um arquivo chamado **SimpleStackDemo.java**.
2. Como explicado, um array fornecerá o armazenamento subjacente para a pilha. Já que essa versão da pilha armazena caracteres, um array **char** é usado. Esse array é acessado por intermédio de um índice que indica o topo da pilha. Tendo isso em mente, comece a criar a classe **SimpleStack** com estas linhas:

```
class SimpleStack {
 char[] data; // esse array contém a pilha
 int tos; // índice do topo da pilha
```

Aqui, **data** referenciará o array que contém a pilha e **tos** é o índice do topo da pilha. Em nossa implementação, o topo da pilha indica o próximo local em que um novo item será armazenado.

3. Adicione o construtor de **SimpleStack** mostrado abaixo. Ele cria uma pilha vazia de tamanho específico.

```
// Constrói uma pilha vazia dado seu tamanho.
SimpleStack(int size) {
 data = new char[size]; // cria o array para armazenar a pilha
 tos = 0;
}
```

O construtor cria um array com o tamanho especificado para armazenar a pilha e atribui a **data** uma referência a esse array. Já que inicialmente a pilha está vazia, **tos** é inicializada com zero. Como acabei de explicar, **tos** é o índice em que o próximo item será armazenado.

4. **SimpleStack** usa quatro métodos: **push()**, **pop()**, **isFull()** e **isEmpty()**. Eles fornecem a funcionalidade básica da pilha: inserir um item na pilha, remover um item da pilha e determinar quando a pilha está cheia ou vazia. As próximas etapas os descreverão.
5. Adicione o método **push()**, que insere um novo elemento no topo da pilha. Ele é mostrado aqui:

```
// Insere um caractere na pilha.
void push(char ch) {
 if(isFull()) {
 System.out.println(" -- Stack is full.");
 return;
 }
```

```

 data[tos] = ch;
 tos++;
}

```

É assim que funciona. Primeiro, fazemos uma verificação para saber se a pilha não está cheia. Isso é feito com uma chamada ao método **isFull()**. Se a pilha estiver cheia, uma mensagem será exibida e o método retornará. Caso contrário, **ch** será adicionado ao topo da pilha com sua atribuição ao índice indicado por **tos** e então **tos** será incrementada. Como explicado, **tos** contém o índice em que o próximo item será armazenado.

6. Adicione o método **pop()**, mostrado a seguir. Ele remove e retorna o elemento do topo da pilha.

```

// Extrai um caractere da pilha.
char pop() {
 if(isEmpty()) {
 System.out.println(" -- Stack is empty.");
 return (char) 0; // um valor de espaço reservado
 }

 tos--;
 return data[tos];
}

```

Quando **pop()** é chamado, primeiro ele verifica se a pilha está vazia chamando **isEmpty()**. Se a pilha estiver vazia, uma mensagem será exibida e um valor de espaço reservado igual a 0 será retornado. Se a pilha não estiver vazia, **tos** será decrementada e o caractere desse índice será retornado.

7. Antes de avançarmos, é importante destacar que a exibição de uma mensagem dentro de **push()** e **pop()** quando ocorre uma condição de pilha cheia ou vazia é simplesmente para fins demonstrativos. Essa abordagem nunca seria usada em um aplicativo do mundo real. O mesmo se aplica a **pop()** retornar um valor de espaço reservado quando a pilha está vazia. Posteriormente, no capítulo 10, você aprenderá uma maneira melhor de tratar erros. Até então, essa abordagem é suficiente.
8. O método **isFull()**, mostrado aqui, retorna **true** quando a pilha está cheia. Adicione-o a **SimpleStack**.

```

// Retorna true se a pilha estiver cheia.
boolean isFull() {
 return tos==data.length;
}

```

A pilha está cheia quando **tos** é igual ao tamanho do array. Lembre-se, a indexação de arrays começa em zero, logo, **data.length** terá uma unidade acima do último elemento do array.

9. Conclua **SimpleStack** adicionando o método **isEmpty()**, mostrado a seguir. Ele retorna **true** quando a pilha está vazia.

```
// Retorna true se a pilha estiver vazia.
boolean isEmpty() {
 return tos==0;
}
```

A pilha está vazia quando **tos** é igual a zero. Isso só ocorrerá se nenhum elemento tiver sido adicionado à pilha ou se removermos o último elemento.

10. Aqui está a classe **SimpleStack** inteira com uma classe chamada **SimpleStackDemo.java** que a demonstra:

```
/*
 Tente isto 5-2

 Uma classe de pilha simples para caracteres.
*/

class SimpleStack {
 char[] data; // esse array contém a pilha
 int tos; // índice do topo da pilha

 // Constrói uma pilha vazia dado seu tamanho.
 SimpleStack(int size) {
 data = new char[size]; // cria o array para armazenar a pilha
 tos = 0;
 }

 // Insere um caractere na pilha.
 void push(char ch) {
 if(isFull()) {
 System.out.println(" -- Stack is full.");
 return;
 }

 data[tos] = ch;
 tos++;
 }

 // Extrai um caractere da pilha.
 char pop() {
 if(isEmpty()) {
 System.out.println(" -- Stack is empty.");
 return (char) 0; // um valor de espaço reservado
 }
 tos--;
 return data[tos];
 }
}
```

```
// Retorna true se a pilha estiver vazia.
boolean isEmpty() {
 return tos==0;
}

// Retorna true se a pilha estiver cheia.
boolean isFull() {
 return tos==data.length;
}
}

// Demonstra a classe SimpleStack.
class SimpleStackDemo {
 public static void main(String[] args) {
 int i;
 char ch;

 System.out.println("Demonstrate SimpleStack\n");

 // Constrói uma pilha vazia para 10 elementos.
 SimpleStack stack = new SimpleStack(10);

 System.out.println("Push 10 items onto a 10-element stack.");

 // Insere as letras A a J na pilha.
 System.out.print("Pushing: ");
 for(ch = 'A'; ch < 'K'; ch++) {
 System.out.print(ch);
 stack.push(ch);
 }

 System.out.println("\nPop those 10 items from stack.");

 // Agora, extrai os caracteres da pilha.
 // Observe que a ordem será o inverso da inserção.
 System.out.print("Popping: ");
 for(i=0; i < 10; i++) {
 ch = stack.pop();
 System.out.print(ch);
 }

 System.out.println("\n\nNext, use isEmpty() and isFull() " +
 "to fill and empty the stack.");

 // Insere as letras até a pilha ficar cheia.
 System.out.print("Pushing: ");
 for(ch = 'A'; !stack.isFull(); ch++) {
 System.out.print(ch);
 stack.push(ch);
 }

 System.out.println();
```

```

// Agora, extrai os caracteres da pilha até ela ficar vazia.
System.out.print("Popping: ");
while(!stack.isEmpty()) {
 ch = stack.pop();
 System.out.print(ch);
}

System.out.println("\n\nNow, use a 4-element stack to generate" +
 " some errors.");

// Gera alguns erros.
SimpleStack smallStack = new SimpleStack(4);

// Tenta inserir 5 caracteres em uma pilha de 4 caracteres
System.out.print("Pushing: ");
for(ch = '1'; ch < '6'; ch++) {
 System.out.print(ch);
 smallStack.push(ch);
}

// Tenta extrair 5 elementos de uma pilha de 4 caracteres.
System.out.print("Popping: ");
for(i=0; i < 5; i++) {
 ch = smallStack.pop();
 System.out.print(ch);
}
}
}

```

11. A saída produzida pelo programa é mostrada a seguir:

```

Demonstrate SimpleStack

Push 10 items onto a 10-element stack.
Pushing: ABCDEFGHIJ
Pop those 10 items from stack.
Popping: JIHGFEDCBA

Next, use isEmpty() and isFull() to fill and empty the stack.
Pushing: ABCDEFGHIJ
Popping: JIHGFEDCBA
Now, use a 4-element stack to generate some errors.
Pushing: 12345 -- Stack is full.
Popping: 4321 -- Stack is empty.

```

Observe na saída que os elementos são extraídos da pilha na ordem oposta em que são inseridos. Lembre-se, uma pilha usa a ordem último a entrar, primeiro a sair. Observe também como **isFull()** e **isEmpty()** podem ser usados no gerenciamento de uma pilha e evitar condições de erro de pilha cheia e vazia.

- 12.** Antes de avançar, você pode fazer um teste. Embora **SimpleStack** armazene caracteres, sua lógica funcionará com outros tipos de dados. Tente modificar **SimpleStack** para que armazene outro tipo de dado, como **int** ou **double**.

## O LAÇO for DE ESTILO FOR-EACH

No trabalho com arrays, é comum encontrarmos situações em que um array deve ser examinado do início ao fim, elemento a elemento. Por exemplo, para calcularmos a soma dos valores contidos em um array, cada elemento do array deve ser examinado. A mesma situação ocorre no cálculo de uma média, na busca de um valor, na cópia de um array e assim por diante. Como essas operações do tipo “início ao fim” são tão comuns, Java define uma segunda forma do laço **for** que as otimiza.

A segunda forma de **for** implementa um laço de estilo “for-each”. Um laço for-each percorre um conjunto de objetos, como um array, de maneira rigorosamente sequencial, do início ao fim. Nos últimos anos, os laços de estilo for-each ganharam popularidade tanto entre projetistas quanto entre programadores de linguagens de computador. Originalmente, Java não oferecia um laço de estilo for-each, mas ele foi adicionado com o lançamento do JDK 5. O estilo for-each de **for** também é chamado de *laço for melhorado*. Os dois termos são usados neste livro.

A forma geral do laço **for** de estilo for-each é mostrada abaixo:

*for(**tipo** **var-iter**: **conjunto**) bloco de instruções*

Aqui, *tipo* especifica o tipo e *var-iter* especifica o nome de uma *variável de iteração* que receberá os elementos de um conjunto, um de cada vez, do início ao fim. O conjunto que está sendo percorrido é especificado por *conjunto*. Há vários tipos de conjuntos que podem ser usados com **for**, mas o único tipo usado neste livro é o array. A cada iteração do laço, o próximo elemento do conjunto é recuperado e armazenado em *var-iter*. O laço se repete até todos os elementos do conjunto terem sido usados. Logo, na iteração por um array de tamanho *N*, o laço **for** melhorado obtém os elementos do array em ordem de índice, de 0 a *N* – 1.

Já que a variável de iteração recebe valores do conjunto, *tipo* deve ser o mesmo dos (ou compatível com) elementos armazenados no conjunto. Portanto, na iteração em arrays, *tipo* deve ser compatível com o tipo de elemento do array.

### Pergunte ao especialista

**P** Além dos arrays, que outros tipos de conjuntos o laço **for** de estilo for-each percorre?

**R** Um dos mais importantes usos do laço **for** de estilo for-each é para percorrer o conteúdo de um conjunto definido pelo Collections Framework. O Collections Framework é um conjunto de classes que implementa várias estruturas de dados, como listas, vetores, conjuntos e mapas. A discussão do Collections Framework pode ser encontrada no Capítulo 25.

Para entender o porquê da existência de um laço de estilo for-each, considere o tipo de laço **for** que é projetado para executar substituições. O fragmento a seguir usa um laço **for** tradicional para calcular a soma dos valores de um array:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

Para calcularmos a soma, **nums** é lido em ordem, do início ao fim, elemento a elemento. Portanto, o array inteiro é lido em ordem rigorosamente sequencial, o que é feito com a indexação manual do array **nums** por **i**, a variável de controle de laço. Além disso, o valor inicial e final da variável de controle de laço e seu incremento devem ser explicitamente especificados.

O laço **for** de estilo for-each automatiza o laço anterior. Especificamente, ele elimina a necessidade de estabelecermos um contador de laço, determinarmos o valor inicial e final e indexarmos manualmente o array. Ele percorre o array inteiro automaticamente, obtendo um elemento de cada vez, em sequência, do início ao fim. Por exemplo, aqui está o fragmento anterior reescrito com o uso de uma versão for-each de **for**:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x : nums) sum += x;
```

A cada passagem do laço, **x** recebe automaticamente um valor igual ao próximo elemento de **nums**. Portanto, na primeira iteração, **x** contém 1, na segunda, contém 2, e assim por diante. Além da otimização da sintaxe, também evitamos erros relacionados aos limites.

Veja um programa inteiro que demonstra a versão de estilo for-each de **for** que acabamos de descrever:

```
// Usa um laço for de estilo for-each.
class ForEach {
 public static void main(String[] args) {
 int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 int sum = 0;

 // Usa o laço for de estilo for-each para exibir e somar os valores.
 for(int x : nums) { ←
 System.out.println("Value is: " + x);
 sum += x;
 } Um laço for de estilo for-each.

 System.out.println("Summation: " + sum);
 }
}
```

A saída do programa é mostrada aqui:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

Como essa saída mostra, o laço **for** de estilo for-each percorre automaticamente um array em ordem, do índice menor ao maior.

Embora esse tipo de laço **for** itere até que todos os elementos de um array tenham sido examinados, é possível encerrar o laço antecipadamente usando uma instrução **break**. Por exemplo, o laço seguinte soma apenas os cinco primeiros elementos de **nums**:

```
// Soma apenas os 5 primeiros elementos.
for(int x : nums) {
 System.out.println("Value is: " + x);
 sum += x;
 if(x == 5) break; // interrompe o laço quando 5 é obtido
}
```

Há um ponto importante que precisa ser conhecido em relação ao laço **for** de estilo for-each. No que diz respeito ao array subjacente, sua variável de iteração é “somente de leitura”. Uma atribuição à variável de iteração não tem efeito sobre o array subjacente. Em outras palavras, você não pode alterar o conteúdo do array atribuindo um novo valor à variável de iteração. Por exemplo, considere este programa:

```
// O laço for-each é somente de leitura.
class NoChange {
 public static void main(String[] args) {
 int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

 for(int x : nums) {
 System.out.print(x + " ");
 x = x * 10; // sem efeito sobre nums ← Isso não altera nums.
 }

 System.out.println();

 for(int x : nums)
 System.out.print(x + " ");

 System.out.println();
 }
}
```

O primeiro laço **for** aumenta o valor da variável de iteração por um fator de 10. No entanto, essa atribuição não tem efeito sobre o array subjacente **nums**, como a saída do segundo laço **for** ilustra.

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Como você pode ver, o array **nums** permanece inalterado.

### Iterando por arrays multidimensionais

O laço **for** melhorado também funciona em arrays multidimensionais. Lembre-se, no entanto, de que, em Java, os arrays multidimensionais são *arrays de arrays*. (Por exemplo, um array bidimensional é um array composto por arrays unidimensionais.) Esse é um detalhe importante na iteração por um array multidimensional, porque cada iteração obtém o *array seguinte* e não um elemento individual. Além disso, a variável de iteração do laço **for** deve ser compatível com o tipo de array que está sendo obtido. Por exemplo, no caso de um array bidimensional, a variável de iteração deve ser uma referência a um array unidimensional. Em geral, quando o laço **for** de estilo for-each é usado na iteração por um array de  $N$  dimensões, os objetos obtidos são arrays de  $N - 1$  dimensões. Para entender as implicações desse fato, considere o programa a seguir. Ele usa laços **for** aninhados para obter os elementos de um array bidimensional por ordem de linha, da primeira à última. Observe como **x** é declarada.

```
// Usa o laço for de estilo for-each em um array bidimensional.
class ForEach2 {
 public static void main(String[] args) {
 int sum = 0;
 int[][] nums = new int[3][5];

 // fornece alguns valores a nums
 for(int i = 0; i < 3; i++)
 for(int j=0; j < 5; j++)
 nums[i][j] = (i+1)*(j+1);

 // Usa o laço for de estilo for-each para exibir e somar os valores.
 for(int[] x : nums) { ←
 for(int y : x) {
 System.out.println("Value is: " + y);
 sum += y;
 }
 }
 System.out.println("Summation: " + sum);
 }
}
```

Observe como **x** é declarada.

A saída desse programa é mostrada abaixo:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

Preste atenção à seguinte linha do programa:

```
| for(int [] x : nums) {
```

Observe como **x** é declarada. Ela é uma referência a um array unidimensional de inteiros. Isso é necessário porque cada iteração de **for** obtém o próximo *array* de **nums**, começando com o array especificado por **nums[0]**. Em seguida, o laço **for** interno percorre cada um desses arrays, exibindo os valores de cada elemento.

## Aplicando o laço for melhorado

Como o laço **for** de estilo for-each só pode percorrer o array sequencialmente, do início ao fim, você deve estar achando que seu uso é limitado. No entanto, isso não é verdade. Vários algoritmos precisam exatamente desse mecanismo. Um dos mais comuns é a busca. Por exemplo, o programa a seguir usa um laço **for** para procurar um valor em um array não classificado. Ele para quando o valor é encontrado.

```
// Pesquisa um array usando o laço for de estilo for-each.
class Search {
 public static void main(String[] args) {
 int[] nums = { 6, 8, 3, 7, 5, 6, 1, 4 };
 int val = 5;
 boolean found = false;

 // Usa o laço for de estilo for-each para procurar val em nums.
 for(int x : nums) {
 if(x == val) {
 found = true;
 break;
 }
 }
 }
}
```

```

 if(found)
 System.out.println("Value found!");
 }
}

```

O laço **for** de estilo for-each é uma ótima opção nesse caso, porque pesquisar um array não classificado envolve examinar cada elemento em sequência. Outros tipos de aplicações que se beneficiam dos laços de estilo for-each são calcular uma média, buscar o valor mínimo ou máximo de um conjunto, procurar duplicatas e assim por diante.

## Verificação do progresso

1. O que o laço **for** de estilo for-each faz?
2. Dado um array de **double** chamado **nums**, mostre um laço **for** de estilo for-each que o percorra.
3. O laço **for** de estilo for-each pode percorrer o conteúdo de um array multidimensional?

## STRINGS

Da perspectiva da programação cotidiana, um dos tipos de dados Java mais importantes é **String**. **String** define e dá suporte a strings de caracteres. Em outras linguagens de programação, um string é um array de caracteres. Não é esse o caso em Java. Em Java, strings são objetos.

Você vem usando a classe **String** desde o Capítulo 1, mas não sabia disso. Ao criar um literal de string, na verdade estava criando um objeto **String**. Por exemplo, na instrução

```
| System.out.println("In Java, strings are objects.");
```

o string “In Java, strings are objects.” é convertido automaticamente em um objeto **String** por Java. Portanto, o uso da classe **String** esteve “nas entrelinhas” dos programas anteriores. Nas seções a seguir, você aprenderá a tratá-la explicitamente. É bom ressaltar, no entanto, que a classe **String** é muito grande e aqui só a examinaremos superficialmente. Ela será vista em detalhes na Parte III.

## Construindo strings

Você pode construir um **String** como construiria qualquer outro tipo de objeto: usando **new** e chamando o construtor de **String**. Por exemplo:

```
| String str = new String("Hello");
```

---

### Respostas:

1. Um laço **for** de estilo for-each percorre o conteúdo de um conjunto, como um array, do início ao fim.
2. `for(double d: nums) ...`
3. Sim; no entanto, cada iteração obtém o próximo subarray.

Essa linha cria um objeto **String** chamado **str** que contém o string de caracteres “Hello”. Você também pode construir um **String** a partir de outro. Por exemplo:

```
| String str = new String("Hello");
| String str2 = new String(str);
```

Após essa sequência ser executada, **str2** também conterá o string de caracteres “Hello”.

Outra maneira fácil de criar um **String** é mostrada aqui:

```
| String str = "Java strings are powerful.,";
```

Nesse caso, **str** é inicializada com a sequência de caracteres “Java strings are powerful.”

Uma vez que você tiver criado um objeto **String**, poderá usá-lo em qualquer local em que um string entre aspas for permitido. Por exemplo, você pode usar um objeto **String** como argumento de **println()**, como mostrado neste exemplo:

```
// Introduz String.
class StringDemo {
 public static void main(String[] args) {
 // declara strings de várias maneiras
 String str1 = new String("Java strings are objects.");
 String str2 = "They are constructed various ways.";
 String str3 = new String(str2);

 System.out.println(str1);
 System.out.println(str2);
 System.out.println(str3);
 }
}
```

A saída do programa é mostrada abaixo:

```
| Java strings are objects.
| They are constructed various ways.
| They are constructed various ways.
```

## Operando com strings

A classe **String** contém vários métodos que operam com strings. Aqui estão as formas gerais de alguns:

|                              |                                                                                                                                                                                    |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean equals( <i>str</i> ) | Retorna verdadeiro se o string chamador tiver a mesma sequência de caracteres de <i>str</i> .                                                                                      |
| int length( )                | Retorna o número de caracteres do string.                                                                                                                                          |
| char charAt( <i>index</i> )  | Retorna o caractere do índice especificado por <i>index</i>                                                                                                                        |
| int compareTo( <i>str</i> )  | Retorna menor do que zero se o string chamador for menor do que <i>str</i> , maior do que zero se o string chamador for maior do que <i>str</i> e zero se os strings forem iguais. |

|                      |                                                                                                                              |
|----------------------|------------------------------------------------------------------------------------------------------------------------------|
| int indexOf(str)     | Procura no string chamador o substring especificado por str. Retorna o índice da primeira ocorrência ou -1 em caso de falha. |
| int lastIndexOf(str) | Procura no string chamador o substring especificado por str. Retorna o índice da última ocorrência ou -1 em caso de falha.   |

Veja um programa que demonstra esses métodos:

```
// Algumas operações com Strings.
class StrOps {
 public static void main(String[] args) {
 String str1 =
 "When it comes to Web programming, Java is #1.";
 String str2 = new String(str1);
 String str3 = "Java strings are powerful.";
 int result, idx;
 char ch;

 System.out.println("Length of str1: " + str1.length());

 // exibe um caractere de cada vez de str1.
 for(int i=0; i < str1.length(); i++)
 System.out.print(str1.charAt(i));
 System.out.println();

 if(str1.equals(str2))
 System.out.println("str1 equals str2");
 else
 System.out.println("str1 does not equal str2");

 if(str1.equals(str3))
 System.out.println("str1 equals str3");
 else
 System.out.println("str1 does not equal str3");

 result = str1.compareTo(str3);
 if(result == 0)
 System.out.println("str1 and str3 are equal");
 else if(result < 0)
 System.out.println("str1 is less than str3");
 else
 System.out.println("str1 is greater than str3");

 // atribui um novo string a str2
 str2 = "One Two Three One";

 idx = str2.indexOf("One");
 System.out.println("Index of first occurrence of One: " + idx);
```

```

 idx = str2.lastIndexOf("One");
 System.out.println("Index of last occurrence of One: " + idx);
}
}

```

Esse programa gera a saída a seguir:

```

Length of str1: 45
When it comes to Web programming, Java is #1.
str1 equals str2
str1 does not equal str3
str1 is greater than str3
Index of first occurrence of One: 0
Index of last occurrence of One: 14

```

Você pode *concatenar* (unir) dois strings usando o operador `+`. Por exemplo, esta sequência

```

String str1 = "One";
String str2 = "Two";
String str3 = "Three";
String str4 = str1 + str2 + str3;

```

inicializa **str4** com o string “OneTwoThree”.

### Pergunte ao especialista

**P** Por que **String** define o método **equals()**? Não posso simplesmente usar `==`?

**R** O método **equals()** compara as sequências de caracteres de dois objetos **String** em busca de igualdade. A aplicação de `==` a duas referências **String** determina apenas se elas referenciam o mesmo objeto.

## Arrays de strings

Como qualquer outro tipo de dado, os strings podem ser reunidos em arrays. Por exemplo:

```

// Demonstra arrays de Strings.
class StringArrays {
 public static void main(String[] args) {
 String[] strs = { "This", "is", "a", "test." };

 System.out.println("Original array: ");
 for(String s : strs)
 System.out.print(s + " ");
 System.out.println("\n");

 // altera um string do array
 strs[1] = "was";
 }
}

```

```

strs[3] = "test, too!";

System.out.println("Modified array: ");
for(String s : strs)
 System.out.print(s + " ");
}
}

```

Esta é a saída:

```

Original array:
This is a test.

Modified array:
This was a test, too!

```

Preste atenção nesta linha do programa:

```
|String[] strs = { "This", "is", "a", "test." };
```

Ela cria um array de strings chamado **strs** composto pelos strings especificados em sua lista de inicializadores. Como o programa ilustra, exceto por conter strings, **strs** funciona como qualquer outro array em Java.

## Strings não podem ser alterados

O conteúdo de um objeto **String** não pode ser mudado. Isto é, uma vez criada, a sequência de caracteres que compõe o string não pode ser alterada. Essa restrição permite que Java implemente strings de maneira mais eficiente. Ainda que possa parecer um problema sério, não é. Se você precisar de um string que seja uma variação de outro já existente, só terá de criar um novo string contendo as alterações desejadas. Já que objetos **String** não usados são coletados como lixo automaticamente, você não precisa se preocupar com o que ocorrerá com os strings descartados. No entanto, é preciso deixar claro que variáveis de referência **String** podem receber uma referência a um objeto **String** diferente. Só o conteúdo de um objeto **String** específico é que não pode ser alterado após ele ser criado.

### Pergunte ao especialista

**P** Você diz que, uma vez criados, os objetos **String** não podem ser alterados. Entendo que, de um ponto de vista prático, essa não seja uma restrição grave, mas e se eu quiser criar um string que possa ser alterado?

**R** Você está com sorte, porque a biblioteca Java fornece classes que dão suporte a strings mutáveis. Uma delas se chama **StringBuffer**, que adiciona métodos que modificam o string armazenado por ela. Por exemplo, além do método **charAt( )**, que obtém o caractere de um local específico, **StringBuffer** define **setCharAt( )**, que configura um caractere dentro do string. No entanto, na maioria dos casos é preferível usar **String** e não **StringBuffer**.

Para explicar exatamente por que não é um problema os strings não poderem ser alterados, usaremos outro dos métodos de **String**: **substring()**. O método **substring()** retorna um novo string contendo a parte especificada do string chamador. Como um novo objeto **String** contendo o substring é criado, o string original não é alterado e a regra de imutabilidade permanece intacta. A forma de **substring()** que usaremos é mostrada abaixo:

```
String substring(int índiceInicial, int índiceFinal)
```

Aqui, *índiceInicial* especifica o índice de partida, e *índiceFinal* é o ponto de chegada. O string retornado contém todos os caracteres do índice inicial até o índice final, porém sem que este seja incluído. Veja um programa que demonstra **substring()** e o princípio dos strings inalteráveis:

```
// Usa substring().
class SubStr {
 public static void main(String[] args) {
 String orgstr = "Java makes the Web move.";

 // constrói um substring
 String substr = orgstr.substring(5, 18); ← Esta linha cria um novo string
 contendo o substring desejado.
 System.out.println("orgstr: " + orgstr);
 System.out.println("substr: " + substr);
 }
}
```

Esta é a saída do programa:

```
orgstr: Java makes the Web move.
substr: makes the Web
```

Como você pode ver, o string original **orgstr** permanece inalterado e **substr** contém o substring.

## Usando um string para controlar uma instrução switch

Como mencionado no Capítulo 3, antes do JDK 7, **switch** tinha que ser controlada por um tipo inteiro, como **int** ou **char**, o que impedia seu uso em situações em que uma entre várias ações era escolhida com base no conteúdo de um string. Em vez disso, uma escada **if-else-if** era a solução mais comum. Embora uma escada **if-else-if** esteja semanticamente correta, uma instrução **switch** seria a expressão mais natural para tal seleção. Felizmente, essa situação foi remediada. Com o lançamento do JDK 7, agora você pode usar um **String** para controlar **switch**, o que em muitos casos resulta em um código mais legível e otimizado.

Um bom uso para um **switch** controlado por um **String** é quando alguma ação deve ser executada a partir de um comando fornecido na forma de string. Um exemplo seria uma instrução **switch** que gerenciasse uma conexão com a Internet a partir de um comando na forma de string. O comando poderia ter sido inserido pelo usuário

ou obtido em um script externo. O programa a seguir demonstra como tratar facilmente a situação com o uso de um **switch** controlado por um **String**:

```
// Usa um string para controlar uma instrução switch.

class StringSwitch {
 public static void main(String[] args) {

 String command = "cancel";

 switch(command) {
 case "connect":
 System.out.println("Connecting");
 // ...
 break;
 case "cancel":
 System.out.println("Canceling");
 // ...
 break;
 case "disconnect":
 System.out.println("Disconnecting");
 // ...
 break;
 default:
 System.out.println("Command Error!");
 break;
 }
 }
}
```

Como era de se esperar, a saída do programa é

```
| Canceling
```

O string contido em **command** (que é “cancel” nesse programa) é verificado em relação às constantes **case**. Quando uma coincidência é encontrada (como na segunda instrução **case**), a sequência de código associada a esse string é executada.

A possibilidade de usar strings em uma instrução **switch** pode ser muito conveniente e melhorar a legibilidade do código. Por exemplo, usar um **switch** baseado em strings é uma melhoria em relação à sequência equivalente de instruções **if/else**. No entanto, é mais dispendioso usar switch com strings do que com inteiros. Logo, é melhor só usar switch com strings em casos em que os dados de controle já estejam na forma de string, como quando o string é inserido pelo usuário ou obtido em uma fonte externa.

## USANDO ARGUMENTOS DE LINHA DE COMANDO

Agora que você conhece a classe **String**, entenderá o parâmetro **args** de **main()** que viu em todos os programas mostrados até aqui. Muitos programas aceitam os chamados *argumentos de linhas de comando*. Um argumento de linha de comando é a

informação que vem diretamente depois do nome do programa na linha de comando quando ele é executado. É muito fácil acessar os argumentos de linha de comando dentro de um programa Java – eles ficam armazenados como strings no array **String** passado para **main( )**. Por exemplo, o programa a seguir exibe todos os argumentos de linha de comando com os quais é chamado:

```
// Exibe todas as informações de linha de comando.
class CLDemo {
 public static void main(String[] args) {
 System.out.println("There are " + args.length +
 " command-line arguments.");
 System.out.println("They are: ");
 for(int i=0; i<args.length; i++)
 System.out.println("arg[" + i + "]: " + args[i]);
 }
}
```

Se **CLDemo** for executado assim,

```
| java CLDemo one two three
```

você verá a saída abaixo:

```
| There are 3 command-line arguments.
| They are:
| arg[0]: one
| arg[1]: two
| arg[2]: three
```

Observe que o primeiro argumento é armazenado no índice 0, o segundo no índice 1, e assim por diante.

Para ter uma ideia da maneira como os argumentos de linha de comando podem ser usados, considere o programa a seguir. Ele recebe um argumento de linha de comando que especifica o nome de uma pessoa. Em seguida, procura esse nome em um array bidimensional de strings. Se encontrar uma ocorrência, exibirá o número do telefone da pessoa.

```
// Uma lista telefônica simples automatizada.
class Phone {
 public static void main(String[] args) {
 String[][] numbers = {
 { "Tom", "555-3322" },
 { "Mary", "555-8976" },
 { "Jon", "555-1037" },
 { "Rachel", "555-1400" }
 };
 int i;

 if(args.length != 1) ← Para o programa ser
 usado, um argumento de
 linha de comando deve
 estar presente.
 System.out.println("Usage: java Phone <name>");
 else {
 for(i=0; i<numbers.length; i++) {
```

```

 if (numbers[i][0].equals(args[0])) {
 System.out.println(numbers[i][0] + ":" +
 numbers[i][1]);
 break;
 }
 }
 if (i == numbers.length)
 System.out.println("Name not found.");
}
}
}

```

Veja um exemplo da execução:

```
C>java Phone Mary
Mary: 555-8976
```

## Verificação do progresso

1. Em Java, todos os strings são objetos. Verdadeiro ou falso?
2. Como você pode obter o tamanho de um string?
3. O que são argumentos de linha de comando?

## OS OPERADORES BITWISE

No Capítulo 2, você conheceu os operadores aritméticos, relacionais e lógicos de Java. Embora esses sejam alguns dos mais usados, a linguagem fornece operadores adicionais que expandem o conjunto de problemas ao qual Java pode ser aplicada. Entre eles estão os operadores bitwise. Esses operadores podem ser aplicados a valores de tipo **long**, **int**, **short**, **char** ou **byte**. As operações bitwise não podem ser usadas com tipos **boolean**, **float**, **double** ou tipos de classe. Eles são chamados de bitwise por serem usados para testar, configurar ou deslocar os bits individuais que compõem um valor. As operações bitwise são importantes em várias tarefas de programação, como quando informações de status de um dispositivo devem ser consultadas ou construídas. A Tabela 5-1 lista os operadores bitwise.

### Os operadores bitwise AND, OR, XOR e NOT

Os operadores bitwise AND, OR, XOR e NOT são **&**, **|**, **^** e **~**. Eles executam as mesmas operações de seus equivalentes lógicos booleanos descritos no Capítulo 2. A

---

Respostas:

1. Verdadeiro.
2. O tamanho de um string pode ser obtido com uma chamada ao método **length()**.
3. Os argumentos de linha de comando são especificados na linha de comando quando um programa é executado. Eles são passados como strings para o parâmetro **args** de **main()**.

**Tabela 5-1** Operadores bitwise

| Operador | Resultado                             |
|----------|---------------------------------------|
| &        | AND bitwise                           |
|          | OR bitwise                            |
| ^        | Exclusivo OR bitwise                  |
| >>       | Deslocamento para a direita           |
| >>>      | Deslocamento para a direita sem sinal |
| <<       | Deslocamento para a esquerda          |
| ~        | Complemento de um (NOT unário)        |

diferença é que os operadores bitwise funcionam bit a bit. A tabela a seguir mostra o resultado de cada operação com o uso de 1s e 0s:

| p | q | p & q | p   q | p ^ q | ~p |
|---|---|-------|-------|-------|----|
| 0 | 0 | 0     | 0     | 0     | 1  |
| 1 | 0 | 0     | 1     | 1     | 0  |
| 0 | 1 | 0     | 1     | 1     | 1  |
| 1 | 1 | 1     | 1     | 0     | 0  |

Pode ajudar considerarmos o AND bitwise como uma maneira de desativar bits. Isto é, qualquer bit que for 0 em um dos operandos fará o bit correspondente do resultado ser configurado com 0. Por exemplo:

$$\begin{array}{r} 1101\ 0011 \\ \& 1010\ 1010 \\ \hline 1000\ 0010 \end{array}$$

O programa a seguir demonstra o operador **&** ao transformar letras minúsculas em maiúsculas pela redefinição do 6º bit com 0. Como definido no conjunto de caracteres Unicode/ASCII, as letras minúsculas são iguais às maiúsculas, exceto pelo fato de minúsculas terem o valor maior em exatamente 32 unidades. Logo, para transformar uma letra minúscula em maiúscula, apenas desative o 6º bit, como este programa ilustra:

```
// Letras maiúsculas.
class UpCase {
 public static void main(String[] args) {
 char ch;

 for(int i=0; i < 10; i++) {
 ch = (char) ('a' + i);
 System.out.print(ch);

 // Essa instrução desativa o 6º bit.
 ch = (char) ((int) ch & 65503); // agora ch é maiúscula
 }
 }
}
```

```
 System.out.print(ch + " ");
 }
}
```

A saída do programa é mostrada aqui:

```
| aA bB cC dD eE fF gG hH iI jJ
```

O valor 65.503 usado na instrução AND é a representação hexadecimal do valor binário 1111 1111 1101 1111. Portanto, a operação AND mantém todos os bits de **ch** inalterados exceto o 6º, que é configurado com 0.

O operador AND também é útil quando queremos determinar se um bit está ativado ou desativado. Por exemplo, esta instrução determina se o bit 4 de **status** está ativado:

```
| if((status & 8) != 0) System.out.println("bit 4 is on");
```

O número 8 é usado porque é convertido em um valor binário que só tem o 4º bit ativado. Logo, a instrução **if** só pode ser bem-sucedida quando o bit 4 de **status** também estiver ativado. Um uso interessante desse conceito seria mostrar os bits de um valor **byte** no formato binário.

```
// Exibe os bits de um byte.
class ShowBitsInByte {
 public static void main(String[] args) {
 int t;
 byte val;

 val = 123;
 for(t=128; t > 0; t = t/2) {
 if((val & t) != 0) System.out.print("1 ");
 else System.out.print("0 ");
 }
 }
}
```

A saída é mostrada aqui:

```
| 0 1 1 1 0 1 1
```

O laço **for** testa sucessivamente cada bit de **val**, usando o operador bitwise AND, para determinar se ele está ativado ou desativado. Se o bit estiver ativado, o dígito **1** será exibido; caso contrário, **0** será exibido. Na seção Tente isto 5-3, você verá como esse conceito básico pode ser expandido para criarmos uma classe que exiba os bits de qualquer tipo de inteiro.

Como oposto de AND, o operador bitwise OR pode ser usado para ativar bits. Qualquer bit que estiver configurado com 1 em um dos operandos fará o bit correspondente do resultado ser configurado com 1. Por exemplo:

```
1 1 0 1 0 0 1 1
| 1 0 1 0 1 0 1 0
1 1 1 1 0 1 1 1
```

Podemos fazer uso de OR para alterar o programa de conversão em maiúsculas para um programa de conversão em minúsculas, como mostrado abaixo:

```
// Letras minúsculas.
class LowCase {
 public static void main(String[] args) {
 char ch;

 for(int i=0; i < 10; i++) {
 ch = (char) ('A' + i);
 System.out.print(ch);

 // Essa instrução ativa o 6º bit.
 ch = (char) ((int) ch | 32); // agora ch é minúscula

 System.out.print(ch + " ");
 }
 }
}
```

A saída desse programa é mostrada a seguir:

```
|Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj
```

O programa funciona usando OR para comparar cada caractere ao valor 32, que é 0000 0000 0010 0000 em binário. Portanto, 32 é o valor que, em binário, só tem o 6º bit ativado. Quando esse valor é comparado com qualquer outro valor por intermédio de OR, ele produz um resultado em que o 6º bit é ativado e todos os outros bits permanecem inalterados. Como explicado, para caracteres, isso resulta em cada letra maiúscula ser transformada em sua equivalente minúscula.

Um exclusive OR, geralmente abreviado para XOR, resultará em um bit ativado se, e somente se, os bits que estiverem sendo comparados forem diferentes, como ilustrado aqui:

$$\begin{array}{r} 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ \wedge \quad 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1 \\ \hline 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0 \end{array}$$

O operador XOR tem uma propriedade interessante. Quando algum valor X é comparado por XOR a um valor Y e o resultado é comparado novamente por XOR a Y, X é produzido. Isto é, dada a sequência

```
|R1 = X ^ Y; R2 = R1 ^ Y;
```

R2 tem o mesmo valor de X. Portanto, o resultado de uma sequência de dois XORs produz o valor original.

Você pode usar esse princípio para criar um programa simples de codificação em que um inteiro seja a chave usada tanto para codificar quanto para decodificar uma mensagem pela comparação de seus caracteres por XOR. Para codificar, a operação XOR é aplicada pela primeira vez, gerando o texto codificado. Para decodificar,

XOR é aplicado uma segunda vez, gerando o texto sem codificação. Obviamente, uma codificação assim não tem valor prático, sendo muito fácil de decifrar. No entanto, fornece uma maneira interessante de demonstrar XOR. Aqui está um programa que usa essa abordagem para codificar e decodificar uma mensagem curta:

```
// Usa XOR para codificar e decodificar uma mensagem.
class SimpleCipher {
 public static void main(String[] args) {
 String msg = "This is a test";
 String encMsg = "";
 String decMsg = "";
 int key = 88;

 System.out.print("Original message: ");
 System.out.println(msg);

 // codifica a mensagem
 for(int i=0; i < msg.length(); i++) {
 encMsg = encMsg + (char) (msg.charAt(i) ^ key);
 }

 System.out.print("Encoded message: ");
 System.out.println(encMsg);

 // decodifica a mensagem
 for(int i=0; i < msg.length(); i++)
 decMsg = decMsg + (char) (encMsg.charAt(i) ^ key);
 System.out.print("Decoded message: ");
 System.out.println(decMsg);
 }
}
```

Esta é a saída:

```
Original message: This is a test
Encoded message: 01+x1+x9x,=+,
Decoded message: This is a test
```

Como você pode ver, o resultado de dois XORs usando a mesma chave produz a mensagem decodificada.

O operador unário complemento de um (NOT) inverte o estado de todos os bits do operando. Por exemplo, se um inteiro chamado A tiver o padrão de bits 1001 0110, então  $\sim A$  produzirá um resultado com o padrão de bits 0110 1001.

O programa a seguir demonstra o operador NOT pela exibição de um número e seu complemento em binário:

```
// Demonstra o NOT bitwise.
class NotDemo {
 public static void main(String[] args) {
 byte b = -34;

 for(int t=128; t > 0; t = t/2) {
```

```

 if((b & t) != 0) System.out.print("1 ");
 else System.out.print("0 ");
 }
 System.out.println();

 // inverte todos os bits
 b = (byte) ~b;

 for(int t=128; t > 0; t = t/2) {
 if((b & t) != 0) System.out.print("1 ");
 else System.out.print("0 ");
 }
}
}

```

Aqui está a saída:

```

1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1

```

## Os operadores de deslocamento

Em Java, podemos deslocar os bits que compõem um valor para a esquerda ou para a direita de acordo com um número especificado. A linguagem define os três operadores de deslocamento de bits mostrados abaixo:

|     |                                       |
|-----|---------------------------------------|
| <<  | Deslocamento para a esquerda          |
| >>  | Deslocamento para a direita           |
| >>> | Deslocamento para a direita sem sinal |

Veja a seguir as formas gerais desses operadores:

*valor << num-bits*  
*valor >> num-bits*  
*valor >>> num-bits*

Aqui, *valor* é o valor que está sendo deslocado de acordo com o número de posições de bits especificado por *num-bits*.

Cada deslocamento para a esquerda faz todos os bits do valor especificado serem deslocados uma posição para a esquerda e um bit 0 ser inserido à direita. Cada deslocamento para a direita desloca todos os bits uma posição para a direita e preserva o bit do sinal. Como você deve saber, geralmente os números negativos são representados pela configuração do bit de ordem superior de um valor inteiro com 1, e essa é a abordagem usada por Java. Logo, se o valor que está sendo deslocado for negativo, cada deslocamento para a direita inserirá um número 1 à esquerda. Se o valor for positivo, cada deslocamento para a direita inserirá um 0 à esquerda.

Além do bit de sinal, devemos estar atentos a mais uma coisa no deslocamento para a direita. Java usa *complemento de dois* para representar valores negativos. Nessa abordagem, valores negativos são armazenados primeiro pela inversão dos bits do valor positivo equivalente e então com a adição de 1. Portanto, o valor do byte

para  $-1$  em binário é 1111 1111. O deslocamento desse valor para a direita sempre produzirá  $-1$ !

Se não quiser preservar o bit de sinal no deslocamento para a direita, você pode usar um deslocamento para a direita sem sinal ( $>>$ ), que sempre insere um 0 à esquerda. Por essa razão, o operador  $>>$  também é chamado de deslocamento para a direita *com preenchimento de zero*. Você usará o deslocamento para a direita sem sinal no deslocamento de padrões de bits, como nos códigos de status, que não representem inteiros.

Em todos os deslocamentos, os bits deslocados para fora são perdidos. Logo, um deslocamento não é rotatório. Quando um bit é deslocado para fora, ele é perdido.

A seguir, mostramos um programa que ilustra graficamente o efeito de um deslocamento para a esquerda e para a direita. Aqui, um inteiro recebe um valor inicial igual a 1, ou seja, seu bit de ordem inferior está ativado. Então, uma série de oito deslocamentos é executada no inteiro. Após cada deslocamento, os 8 bits inferiores do valor são mostrados. O processo é repetido, exceto por um número 1 ser inserido na 8<sup>a</sup> posição, e deslocamentos para a direita são executados.

```
// Demonstra os operadores de deslocamento << e >>.
class ShiftDemo {
 public static void main(String[] args) {
 int val = 1;

 for(int i = 0; i < 8; i++) {
 for(int t=128; t > 0; t = t/2) {
 if((val & t) != 0) System.out.print("1 ");
 else System.out.print("0 ");
 }
 System.out.println();
 val = val << 1; // desloca para a esquerda
 }
 System.out.println();

 val = 128;
 for(int i = 0; i < 8; i++) {
 for(int t=128; t > 0; t = t/2) {
 if((val & t) != 0) System.out.print("1 ");
 else System.out.print("0 ");
 }
 System.out.println();
 val = val >> 1; // desloca para a direita
 }
 }
}
```

A saída do programa é mostrada abaixo:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

```

0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1

```

Você precisa tomar cuidado quando deslocar valores **byte** e **short**, porque Java promoverá automaticamente esses tipos a **int** ao avaliar uma expressão. Por exemplo, se você deslocar para a direita um valor **byte**, primeiro ele será promovido a **int** para então ser deslocado. O resultado do deslocamento também será de tipo **int**. Geralmente, essa conversão não traz consequências. No entanto, se você deslocar um valor **byte** ou **short** negativo, ele será estendido pelo sinal quando for promovido a **int**. Logo, os bits de ordem superior do valor inteiro resultante serão preenchidos com números 1. Isso não causa problemas na execução de um deslocamento comum para a direita. Mas, quando você executar um deslocamento para a direita com preenchimento de zeros, haverá 24 algarismos 1 a serem deslocados antes de o valor **byte** começar a ver zeros.

### Pergunte ao especialista

**P** Já que os binários se baseiam em potências de dois, os operadores de deslocamento podem ser usados como um atalho para a multiplicação ou divisão de um inteiro por dois?

**R** Sim. Os operadores de deslocamento bitwise podem ser usados para executar uma multiplicação ou divisão muito rápida por dois. Um deslocamento para a esquerda dobra o valor. Um deslocamento para a direita o reduz à metade.

### Atribuições abreviadas bitwise

Todos os operadores bitwise binários têm uma forma abreviada que combina uma atribuição com a operação bitwise. Por exemplo, as duas instruções a seguir atribuem a **x** o resultado de uma operação XOR de **x** com o valor 127.

```

x = x ^ 127;
x ^= 127;

```

### TENTE ISTO 5-3 Uma classe de uso geral para a exibição de bits

ShowBitsDemo.java

Este projeto cria uma classe utilitária chamada **BitOut** que permite a exibição do padrão de bits de qualquer valor inteiro em binários. Uma classe assim pode ser

muito útil em programação. Por exemplo, na depuração de uma conexão da Internet, pode ser benéfico monitorar o fluxo de dados em binário. Ela também fornece uma estrutura à qual você poderia adicionar outras opções de exibição de bits.

### PASSO A PASSO

1. Crie um arquivo chamado **ShowBitsDemo.java**.
2. Comece a classe chamada **BitOut** como mostrado aqui:

```
class BitOut {
 int numBits; // número de bits a serem exibidos

 BitOut(int n) {
 if(n < 1) n = 1;
 if(n > 64) n = 64;
 numBits = n;
 }
}
```

**BitOut** cria um objeto que pode exibir um número especificado de bits entre 1 e 64. Por exemplo, para criar um objeto que exiba os 8 bits de ordem inferior de um valor, use

```
|BitOut byteval = new BitOut(8)
```

O número de bits a serem exibidos é armazenado em **numbits**.

3. Para exibir realmente o padrão de bits, **BitOut** fornece o método **showBits()**, que é mostrado abaixo:

```
// Exibe a sequência de bits.
void showBits(long val) {
 long mask = 1;

 // desloca um 1 para a esquerda para a posição apropriada
 mask <= numBits-1;

 int spacer = 8 - (numBits % 8);
 for(; mask != 0; mask >>>= 1) {
 if((val & mask) != 0) System.out.print("1");
 else System.out.print("0");

 spacer++;

 if((spacer % 8) == 0) {
 System.out.print(" ");
 spacer = 0;
 }
 }
 System.out.println();
}
```

Observe que **showBits()** especifica um parâmetro **long**. No entanto, isso não significa que você terá sempre de passar para **showBits()** um valor **long**. Devido às promoções de tipo automáticas de Java, qualquer tipo inteiro

ro pode ser passado para **showBits()**. O número de bits exibidos é determinado pelo valor armazenado em **numbits**. Os bits são exibidos em blocos de 8, começando pela direita. Isso facilita a leitura dos valores binários de padrões de bits longos.

- O programa a seguir junta as partes e adiciona a classe **ShowBitsDemo**, que demonstra o uso de **BitOut** e **showBits()**.

```
/*
Tente isto 5-3

Uma classe que armazena o número de bits que serão
exibidos. Em seguida, ele implementa o método showBits(),
que exibe esse número de bits da representação binária
do valor que recebe.

*/
class BitOut {
 int numBits; // número de bits a serem exibidos

 BitOut(int n) {
 if(n < 1) n = 1;
 if(n > 64) n = 64;
 numBits = n;
 }

 // Exibe a sequência de bits.
 void showBits(long val) {
 long mask = 1;

 // desloca um 1 para a esquerda para a posição apropriada
 mask <= numBits-1;

 int spacer = 8 - (numBits % 8);
 for(; mask != 0; mask >>>= 1) {
 if((val & mask) != 0) System.out.print("1");
 else System.out.print("0");

 spacer++;

 if((spacer % 8) == 0) {
 System.out.print(" ");
 spacer = 0;
 }
 }
 System.out.println();
 }
}

// Demonstra showBits().
class ShowBitsDemo {
 public static void main(String[] args) {
 BitOut b = new BitOut(8);
 BitOut i = new BitOut(32);
```

```

BitOut li = new BitOut(64);

System.out.println("123 in binary: ");
b.showBits(123);

System.out.println("\n87987 in binary: ");
i.showBits(87987);

System.out.println("\n237658768 in binary: ");
li.showBits(237658768);

// você também pode exibir os bits de ordem inferior de
// qualquer inteiro
System.out.println("\nLow order 8 bits of 87987 in binary: ");
b.showBits(87987);
}
}

```

5. A saída de **ShowBitsDemo** é mostrada abaixo:

```

123 in binary:
01111011

87987 in binary:
00000000 00000001 01010111 10110011

237658768 in binary:
00000000 00000000 00000000 00001110 00101010 01100010
10010000

Low order 8 bits of 87987 in binary:
10110011

```

6. Se quiser, tente adicionar à classe **BitOut** outros métodos que exibam os bits de maneiras diferentes. Por exemplo, você poderia exibir os bits na ordem inversa ou exibir apenas um subconjunto deles.

## Verificação do progresso

1. A que tipos os operadores bitwise podem ser aplicados?
2. O que é **>>>**?

Respostas:

1. **byte, short, int, long e char.**
2. O operador **>>>** executa um deslocamento para a direita sem sinal. Isso faz um zero ser deslocado para a posição do bit da extrema esquerda. Ele é diferente de **>>**, que preserva o bit do sinal.

## O OPERADOR ?

Um dos operadores mais fascinantes de Java é o operador **?**. Geralmente, o operador **?** é usado para substituir instruções **if-else** que têm esta forma geral:

```
if(condição)
 var = expressão1;
else
 var = expressão2;
```

Aqui, o valor atribuído a *var* depende do resultado da condição que controla **if**.

O operador **?** é chamado de *operador ternário* porque requer três operandos. Ele tem a forma geral

$$\text{condição} ? \text{expressão1} : \text{expressão2}$$

em que *condição* é uma expressão **boolean** e *expressão1* e *expressão2* são expressões de qualquer tipo menos **void**. No entanto, o tipo de *expressão1* e *expressão2* deve ser o mesmo (ou compatível). Observe o uso e a posição dos dois pontos.

O valor de uma expressão **?** é determinado desta forma: a *condição* é avaliada. Se for verdadeira, a *expressão1* será avaliada passando a ser o valor da expressão **?** inteira. Se a *condição* for falsa, a *expressão2* será avaliada e seu valor passará a ser o valor da expressão. Considere este exemplo, que atribui a **absval** o valor absoluto de **val**:

```
| absval = val < 0 ? -val : val; // obtém o valor absoluto de val
```

Aqui, **absval** receberá o valor de **val** se **val** for zero ou maior. Se **val** for negativa, **absval** receberá o negativo desse valor (que gera um valor positivo). O mesmo código escrito com o uso da estrutura **if-else** teria a seguinte aparência:

```
| if(val < 0) absval = -val;
| else absval = val;
```

Aqui está outro exemplo do operador **?**. Este programa divide dois números, mas não permitirá uma divisão por zero.

```
// Impede uma divisão por zero usando o operador ?.
class NoZeroDiv {
 public static void main(String[] args) {
 int result;

 for(int i = -5; i < 6; i++) {
 result = i != 0 ? 100 / i : 0; ← Esta parte impede uma
 if(i != 0) divisão por zero.
 System.out.println("100 / " + i + " is " + result);
 }
 }
}
```

A saída do programa é mostrada a seguir:

```
| 100 / -5 is -20
| 100 / -4 is -25
```

```
100 / -3 is -33
100 / -2 is -50
100 / -1 is -100
100 / 1 is 100
100 / 2 is 50
100 / 3 is 33
100 / 4 is 25
100 / 5 is 20
```

Preste atenção nesta linha do programa:

```
| result = i != 0 ? 100 / i : 0;
```

Nela, **result** recebe o resultado da divisão de 100 por **i**. No entanto, essa divisão só ocorre se **i** não for zero. Quando **i** é zero, um preenchimento de valor zero é atribuído a **result**.

Você não precisa atribuir o valor produzido pelo operador **?** a uma variável. Você poderia usar o valor como argumento na chamada a um método. Ou, se as expressões forem todas de tipo **boolean**, o operador **?** pode ser usado como a expressão condicional em um laço ou instrução **if**. Por exemplo, este é o programa anterior reescrito de maneira um pouco mais compacta. Ele produz o mesmo resultado de antes:

```
// Impede uma divisão por zero usando o operador ?.
class NoZeroDiv2 {
 public static void main(String[] args) {

 for(int i = -5; i < 6; i++)
 if(i != 0 ? true : false)
 System.out.println("100 / " + i +
 " is " + 100 / i);
 }
}
```

Observe a instrução **if**. Se **i** for zero, a expressão condicional de **if** será falsa, a divisão por zero será evitada e nenhum resultado será exibido. Caso contrário, a divisão ocorrerá.

---

## EXERCÍCIOS

1. Mostre duas maneiras de declarar um array unidimensional de 12 **doubles**.
2. Mostre como inicializar um array unidimensional de inteiros com os valores de 1 a 5.
3. Escreva um programa que use um array para encontrar a média de 10 valores **double**. Use os 10 valores que quiser.
4. Altere a classificação da seção Tente isto 5-1 para que classifique um array de strings. Demonstre que isso funciona.
5. Qual é a diferença entre os métodos **indexOf( )** e **lastIndexOf( )** de **String**?

6. Já que todos os strings são objetos de tipo **String**, mostre como chamar os métodos **length()** e **charAt()** neste literal de string: “I like Java”.
7. Expandindo a classe **SimpleCipher**, modifique-a para que use um string de oito caracteres como chave.
8. Os operadores bitwise podem ser aplicados ao tipo **double**?
9. Mostre como a sequência a seguir pode ser reescrita com o uso do operador **?**.

```
| if(x < 0) y = 10;
| else y = 20;
```

10. No fragmento a seguir, **&** é um operador bitwise ou lógico? Por quê?

```
| boolean a, b;
| // ...
| if(a & b) ...
```

11. É um erro ultrapassar o fim de um array? E indexar um array com um valor negativo?
12. Qual é o símbolo usado para o operador de deslocamento para a direita sem sinal?
13. Reescreva a classe **MinMax** mostrada anteriormente neste capítulo para que use um laço **for** de estilo for-each.
14. Os laços **for** que executam a classificação na classe **Bubble** mostrada na seção Tente isto 5-1 podem ser convertidos em laços de estilo for-each? Em caso negativo, por que não?
15. Um **String** pode controlar uma instrução **switch**?
16. Escreva um programa que crie um array de inteiros de tamanho 30, prencha o array com a sequência de valores (mostrada abaixo) usando um laço **for** e percorra o array exibindo os valores. Use um laço **for** de estilo for-each para exibir os valores.
  - A. 1, -2, 3, -4, 5, -6, ..., 29, -30
  - B. 1, 1, 2, 2, 3, 3, ..., 15, 15
  - C. 1, 2, 4, 8, 16, ...
  - D. 1, 1, 2, 3, 5, 8, 13, ... (Exceto os dois primeiros valores, cada valor é a soma dos dois valores anteriores)
17. Escreva um programa que crie um array de **doubles** de tamanho 50. Em seguida, ele preenche o array com a sequência de valores  $2^{1/2}, 2^{1/4}, 2^{1/8}, 2^{1/16}, \dots$  e então o percorre exibindo os valores. Use a função **Math.sqrt()** para calcular as raízes quadradas dos valores.
18. Você pode ter um array de tamanho 0? Se puder, como criaria um?
19. Altere o código da classificação por bolha da seção Tente isto 5-1 para que ele classifique o array de inteiros do maior para o menor em vez de do menor para o maior.

20. Altere o código da classificação por bolha da seção Tente isto 5-1 para que ele classifique um array de strings por seus tamanhos (em vez de classificá-los alfabeticamente). Demonstre que funciona.
21. Observe que, se o laço interno da classificação por bolha não trocar pares de valores, o array estará classificado e a rotina de classificação poderá parar. Modifique o código da seção Tente isto 5-1 para que a classificação por bolha pare assim que o laço interno não trocar mais valores.
22. Escreva um programa que crie um array bidimensional “triangular” **A** de 10 linhas. A primeira linha tem tamanho 1, a segunda tem tamanho 2, a terceira tem tamanho 3 e assim por diante. Em seguida, inicialize o array usando laços **for** aninhados para que o valor de **A[i][j]** seja **i+j**. Para concluir, exiba o array em uma bela forma triangular.
23. Escreva um programa que crie um array de inteiros e use um laço **for** para inverter a ordem dos elementos do array.
24. Insira o método **indexOf()** na classe abaixo para que use um laço **for** e encontre o índice de **x** em **data**. Ele retorna o índice ou -1 se **x** não estiver em **data**. Por exemplo, **indexOf(3)** retorna 2. Demonstre sua solução.

```
class MyClass {
 int[] data = { 1, 8, 3, 5, 4, 6, 10, 9, 2, 7 };

 int indexOf(int x) {
 // ... adicione seu código aqui
 }
}
```

25. Escreva um programa que crie um array de inteiros **data** e use um laço **for** para criar um novo **String** que exiba o conteúdo do array **data** entre chaves e separado por vírgulas. Por exemplo, se o array **data** tiver tamanho 4 e armazenar os valores 3, 4, 1, 5, o **String** “{3, 4, 1, 5}” deve ser criado e exibido.
26. Escreva um programa que crie dois arrays de inteiros **data1** e **data2**, possivelmente de tamanhos diferentes. Em seguida, use laços **for** para criar um novo array **data3** com tamanho igual à soma dos tamanhos de **data1** e **data2** e com conteúdo composto pelo conteúdo de **data1** seguido pelo conteúdo de **data 2**. Por exemplo, se os dois arrays forem {1,2,3} e {4,5,6,7}, o código deve criar o novo array {1,2,3,4,5,6,7}.
27. Escreva um programa que crie um array de inteiros e use um laço **for** para verificar se o array está classificado do maior para o menor. Se estiver, ele exibirá “Sorted”. Caso contrário, exibirá “Not sorted”.
28. Escreva um programa que crie um **String** e use um laço **for** para verificar se ele é um palíndromo, ou seja, se você inverter a ordem dos caracteres do **String**, obterá o mesmo **String**. Por exemplo, “abcdcba” é um palíndromo.
29. Escreva um programa que crie um string e use um laço **for** para dividi-lo em um array de substrings, usando a vírgula como separador. Por exemplo, se o string fosse “abc,def,hi”, o array criado seria {"abc", "def", "hi"}. Em seguida, ele exibe o string e o novo array. Não use o método **split()** da classe **String**.

30. Reescreva as instruções a seguir sem usar o operador ?. Presuma que **y** é uma variável inteira já declarada e inicializada.

```
a. int x = y > 0 ? 3 : 4;
b. int x = ((y > 0) ? (y > 5) : (y < -5)) ? 3 : 4;
c. int x = y > 0 ? (y > 5 ? 3 : 4) : (y < -5 ? 6 : 7);
```

31. Implemente um método **cyclicShift()** que use dois inteiros, **x** e **dist**, como parâmetros. Ele desloca ciclicamente os bits da representação de **x** de acordo com a distância fornecida por **dist** e retorna o resultado. Se **dist**  $\geq 0$ , deslocará para a direita; caso contrário, deslocará para a esquerda. Um *deslocamento cíclico* desloca os bits para a esquerda ou direita e os bits que são deslocados para fora de uma extremidade são inseridos na outra para preencher os bits vagos. (O deslocamento cíclico também é chamado de *rotação*.) Por exemplo, um deslocamento cíclico de 10010111 para a direita a uma distância igual a 3 resultaria em 11110010. Use os operadores OR e de deslocamento para executar o deslocamento cíclico. *Dica:* desloque uma cópia de **x** para a direita e outra para a esquerda, usando o fato de que um inteiro tem 32 bits, e então empregue o operador OR para combiná-las. Demonstre seu método usando a classe **BitOut** da seção Tente isto 5-3 para exibir os resultados.

32. Suponhamos que um **SimpleStack** fosse criado com o nome **stack** e a sequência de instruções a seguir fosse executada. Quando isso for feito, que caracteres serão deixados na pilha e em que ordem?

```
stack.push('a');
stack.push('b');
stack.pop();
stack.push('c');
stack.push('d');
stack.pop();
stack.push('e');
stack.pop();
stack.pop();
```

# 6

---

# Verificação minuciosa dos métodos e classes

## PRINCIPAIS HABILIDADES E CONCEITOS

- Controlar o acesso a membros
- Passar objetos para um método
- Retornar objetos de um método
- Sobrecarregar métodos
- Sobrecarregar construtores
- Usar recursão
- Aplicar **static**
- Usar classes internas
- Usar varargs

Este capítulo retoma nosso estudo das classes e métodos. Ele começa explicando como controlar o acesso aos membros de uma classe. Em seguida, discute a passagem e o retorno de objetos, a sobrecarga de métodos, a recursão e o uso da palavra-chave **static**. Também são descritas as classes aninhadas e os argumentos em quantidade variável.

## CONTROLANDO O ACESSO A MEMBROS DE CLASSES

Em seu suporte ao encapsulamento, a classe fornece dois grandes benefícios. Em primeiro lugar, ela vincula os dados ao código que os trata. Você vem se beneficiando desse aspecto da classe desde o Capítulo 4. Em segundo lugar, fornece o meio pelo qual o acesso a membros pode ser controlado. Esse recurso será examinado aqui.

Embora a abordagem de Java seja um pouco mais sofisticada, há dois tipos básicos de membros de classes: públicos e privados. Um membro público pode ser acessado livremente por um código definido fora de sua classe. Esse é o tipo de membro que usamos até agora. Um membro privado só pode ser acessado por outros métodos definidos por sua classe. Com o uso de membros privados, o acesso é controlado.

A restrição do acesso a membros de uma classe é parte fundamental da programação orientada a objetos, porque ajuda a impedir a má utilização de um objeto. Ao permitir o acesso a dados privados apenas por intermédio de um conjunto de métodos bem definido, você pode impedir que valores inapropriados sejam atribuídos a esses dados – executando uma verificação de intervalo, por exemplo. Um código de fora da classe não pode definir o valor de um membro privado diretamente. Você também pode controlar exatamente como e quando os dados de um objeto serão usados. Logo, quando corretamente implementada, uma classe cria uma “caixa preta” que pode ser usada, mas cujo funcionamento interno não está aberto a alterações.

Até o momento, você não teve de se preocupar com o controle de acesso, porque Java fornece uma configuração de acesso padrão em que, nos programas vistos até agora, os membros de uma classe ficam livremente disponíveis para outros códigos do programa. (Portanto, para esses programas, a configuração de acesso padrão é basicamente pública.) Embora conveniente para classes simples (e exemplos de programa de livros como este), essa configuração padrão é inadequada em muitas situações do mundo real. Aqui você verá como usar outros recursos de controle de acesso de Java.

## Modificadores de acesso da linguagem Java

O controle de acesso a membros é obtido com o uso de três *modificadores de acesso*: **public**, **private** e **protected**. Como explicado, se nenhum modificador de acesso for usado, será presumido o uso da configuração de acesso padrão. Neste capítulo, ocuparemos-nos de **public** e **private**. O modificador **protected** só é útil quando há herança envolvida; ele será descrito no Capítulo 9.

Quando o membro de uma classe é modificado pelo especificador **public**, esse membro pode ser acessado por qualquer código do programa. Isso inclui métodos definidos dentro de outras classes.

Quando o membro de uma classe é especificado como **private**, ele só pode ser acessado por outros membros de sua classe. Logo, métodos de classes diferentes não podem acessar um membro **private** de outra classe.

A configuração de acesso padrão (em que nenhum modificador de acesso é usado) é igual a **public**, a menos que dois ou mais pacotes estejam envolvidos. Um *pacote* é, basicamente, um agrupamento de classes. Os pacotes são um recurso tanto organizacional quanto de controle de acesso, mas sua discussão deve esperar até o Capítulo 9. Para os tipos de programas mostrados neste capítulo e nos anteriores, o acesso **public** é igual ao acesso padrão.

Um modificador de acesso precede o resto da especificação de tipo de um membro. Isto é, ele deve começar a instrução de declaração do membro. Aqui estão alguns exemplos:

```
public String errMsg;
private accountBalance bal;

private boolean isError(byte status) { // ...
```

Para entender os efeitos de **public** e **private**, considere o programa a seguir:

```
// Acesso público versus privado.
class MyClass {
 private int alpha; // acesso privado
```

```
public int beta; // acesso público
int gamma; // acesso padrão
/* Métodos para acessar alpha. Não há problema em um
 membro de uma classe acessar um membro privado
 da mesma classe.
*/
void setAlpha(int a) {
 alpha = a;
}

int getAlpha() {
 return alpha;
}
}

class AccessDemo {
 public static void main(String[] args) {
 MyClass ob = new MyClass();

 /* O acesso a alpha só é permitido
 por intermédio de seus métodos acessadores. */
 ob.setAlpha(-99);
 System.out.println("ob.alpha is " + ob.getAlpha());

 // Você não pode acessar alpha dessa forma:
// ob.alpha = 10; // Errado! alpha é privado! ← Errado – alpha é privado!

 // Essas linhas estão corretas porque beta e gamma podem ser
 // acessados.
 ob.beta = 88; ← Certo porque esses membros podem ser acessados.
 ob.gamma = 99;
 }
}
```

Como você pode ver, dentro da classe **MyClass**, **alpha** é especificado como **private**, **beta** é especificado explicitamente como **public** e **gamma** usa o acesso padrão, que nesse exemplo é igual à especificação de **public**. Já que **alpha** é privado, não pode ser acessado por um código de fora de sua classe. Logo, dentro da classe **AccessDemo**, **alpha** não pode ser usado diretamente. Deve ser acessado por intermédio de seus métodos acessadores públicos: **setAlpha()** e **getAlpha()**. Se você removesse o símbolo de comentário do começo da linha abaixo,

```
// ob.alpha = 10; // Errado! alpha é privado!
```

não poderia compilar esse programa devido à violação de acesso. Embora o acesso ao membro **alpha** por um código de fora de **MyClass** não seja permitido, métodos definidos dentro de **MyClass** podem acessá-lo livremente, como mostram os métodos **setAlpha()** e **getAlpha()**.

O ponto-chave é este: um membro privado pode ser usado livremente por outros membros de sua classe, mas não pode ser acessado por um código de fora dela.

Para ver como o controle de acesso pode ser aplicado a um exemplo mais prático, considere o programa a seguir que implementa um array **int** “resistente a falhas”,

em que é impedida a ocorrência de erros relacionados a limites, o que evita que uma exceção de tempo de execução seja gerada. Isso é feito com o encapsulamento do array como membro privado de uma classe, sendo seu acesso permitido apenas por intermédio de métodos membros. Nessa abordagem, qualquer tentativa de acessar o array fora de seus limites pode ser evitada, com a tentativa falhando silenciosamente (com uma “leve aterrissagem” e não uma “queda”). O array resistente a falhas é implementado pela classe **FailSoftArray**, mostrada aqui:

```
/* Esta classe implementa um array "resistente a falhas" que impede a
 ocorrência de erros de tempo de execução.
*/
class FailSoftArray {
 private int[] a; // referência a array
 private int errval; // valor a retornar se get() falhar
 public int length; // length é público

 /* Constrói o array dados seu tamanho e o valor a ser
 retornado se get() falhar. */
 public FailSoftArray(int size, int errv) {
 a = new int[size];
 errval = errv;
 length = size;
 }

 // Retorna o valor do índice especificado.
 public int get(int index) {
 if(ok(index)) return a[index]; ← Detecta índice fora dos limites.
 return errval;
 }

 // Insere um valor em um índice. Retorna false em caso de falha.
 public boolean put(int index, int val) {
 if(ok(index)) { ←
 a[index] = val;
 return true;
 }
 return false;
 }

 // Retorna true se index estiver dentro dos limites.
 private boolean ok(int index) {
 if(index >= 0 & index < length) return true;
 return false;
 }
}

// Demonstra o array resistente a falhas.
class FSDemo {
 public static void main(String[] args) {
 FailSoftArray fs = new FailSoftArray(5, -1);
 int x;
```

```
// exibe falhas silenciosas
System.out.println("Fail quietly.");
for(int i=0; i < (fs.length * 2); i++)
 fs.put(i, i*10); ← O acesso ao array deve ser feito por
 intermédio de seus métodos de acesso.
for(int i=0; i < (fs.length * 2); i++) {
 x = fs.get(i); ←
 if(x != -1) System.out.print(x + " ");
}
System.out.println("");

// agora, trata as falhas
System.out.println("\nFail with error reports.");
for(int i=0; i < (fs.length * 2); i++)
 if(!fs.put(i, i*10))
 System.out.println("Index " + i + " out-of-bounds");

for(int i=0; i < (fs.length * 2); i++) {
 x = fs.get(i);
 if(x != -1) System.out.print(x + " ");
 else
 System.out.println("Index " + i + " out-of-bounds");
}
}
```

A saída do programa é mostrada abaixo:

```
Fail quietly.
0 10 20 30 40

Fail with error reports.
Index 5 out-of-bounds
Index 6 out-of-bounds
Index 7 out-of-bounds
Index 8 out-of-bounds
Index 9 out-of-bounds
0 10 20 30 40 Index 5 out-of-bounds
Index 6 out-of-bounds
Index 7 out-of-bounds
Index 8 out-of-bounds
Index 9 out-of-bounds
```

Examinemos melhor esse exemplo. Dentro de **FailSoftArray** são definidos três membros **private**. O primeiro é **a**, que armazena uma referência ao array que conterá realmente as informações. O segundo é **errval**, que é o valor a ser retornado quando uma chamada a **get( )** falhar. O terceiro é o método **private ok( )**, que determina se um índice está dentro dos limites. Portanto, esses três membros só podem ser usados por outros membros da classe **FailSoftArray**. Especificamente, **a** e **errval** só podem ser usados por outros métodos da classe e **ok( )** só pode ser chamado por outros

membros de **`FailSoftArray`**. O resto dos membros da classe são **public** e podem ser chamados por qualquer código de um programa que use **`FailSoftArray`**.

Quando um objeto **`FailSoftArray`** for construído, você deve especificar o tamanho do array e o valor que deseja retornar se uma chamada a **`get()`** falhar. O valor de erro deve ser um valor que de outra forma não seria armazenado no array. Após a construção, o array real referenciado por **a** e o valor de erro armazenado em **errval** não poderão ser acessados por usuários do objeto **`FailSoftArray`**. Logo, eles não podem ser mal utilizados. Por exemplo, o usuário não pode tentar indexar **a** diretamente e exceder seus limites. O acesso só está disponível por intermédio dos métodos **`get()`** e **`put()`**.

O método **`ok()`** é **private** principalmente a título de ilustração. Seria inofensivo torná-lo **public**, porque ele não modifica o objeto. No entanto, já que é usado internamente pela classe **`FailSoftArray`**, pode ser **private**.

Observe que a variável de instância **length** é **public**. Isso está de acordo com a maneira como Java implementa arrays. Para obter o tamanho de um **`FailSoftArray`**, só temos que usar seu membro **length**. É importante ressaltar, no entanto, que como **length** é público, pode ser mal utilizado. Por exemplo, poderia ser configurado com um valor inapropriado. No Capítulo 7, veremos outro recurso Java que pode ser usado para impedir essa má utilização.

Ao usar um array **`FailSoftArray`**, você deve chamar **`put()`** para armazenar um valor no índice especificado e chamar **`get()`** para recuperar um valor de um índice especificado. Se o índice estiver fora dos limites, **`put()`** retornará **false** e **`get()`** retornará **errval**.

Por conveniência, grande parte dos exemplos deste livro continuará a usar o acesso padrão para a maioria dos membros. Lembre-se, no entanto, de que, no mundo real, restringir o acesso aos membros – principalmente variáveis de instância – é parte importante de uma programação orientada a objetos bem-sucedida. Como você verá no Capítulo 7, o controle de acesso é ainda mais vital quando a herança está envolvida.

## Verificação do progresso

1. Cite quais são os modificadores de acesso Java.
2. Explique o que **private** faz.

### TENTE ISTO 6-1 Melhorando SimpleStack

`SimpleStack.java`

Você pode usar o modificador **private** para fazer uma melhoria importante na classe **SimpleStack** desenvolvida na seção Tente isto 5-2 do capítulo anterior. Naquela versão, todos os membros de **SimpleStack** usam o acesso padrão. Ou seja, um programa que usasse **SimpleStack** poderia acessar diretamente o array subjacente, possivelmente usando seus elementos fora de ordem. Já que o que

Respostas:

1. **private**, **public** e **protected**. Um acesso padrão também está disponível.
2. Quando um membro é especificado como **private**, ele só pode ser acessado por outros membros de sua classe.

interessa em uma pilha é o fornecimento de uma lista “último a entrar, primeiro a sair”, não é desejável permitir o acesso fora de ordem. Além disso, os valores do array subjacente poderiam ser alterados diretamente, com o uso de **push()** e **pop()** sendo ignorado, o que adulteraria a pilha. Também seria possível um programador malicioso alterar o valor armazenado na variável **tos**. Isso também adulteraria a pilha. Felizmente, esses tipos de problemas são fáceis de evitar com a aplicação do especificador **private**.

#### PASSO A PASSO

1. Comece com a classe **SimpleStack** original da seção Tente isto 5-2.
2. Em **SimpleStack**, adicione o modificador **private** ao array **data** e à variável **tos**, como mostrado aqui:

```
| private char[] data; // esse array contém a pilha
| private int tos; // índice do topo da pilha
```

3. A alteração de **data** e **tos** do acesso padrão para o acesso privado não terá efeito sobre um programa que use **SimpleStack** apropriadamente. Por exemplo, também funcionaria bem com a classe **SimpleStackDemo** da seção Tente isto 5-2. No entanto, o uso de **private** impede o uso inapropriado de **SimpleStack**. Por exemplo, dado o código

```
| SimpleStack myStack = new SimpleStack(10);
```

os tipos de instruções a seguir são inválidos:

```
| myStack.data[0] = 'X'; // errado!
| myStack.tos = -100; // não funcionará!
```

4. Agora que **tos** e **data** são privadas, a classe **SimpleStack** está impondo rigorosamente o atributo último a entrar, primeiro a sair de uma pilha. Isso também impede que a pilha seja adulterada maliciosa ou accidentalmente pelo uso inapropriado de **data** ou **tos**. A simples adição de **private** a esses dois campos é uma das maneiras mais fáceis, porém eficazes, de melhorar a consistência de **SimpleStack**.

## PASSE OBJETOS PARA OS MÉTODOS

Até agora, os exemplos deste livro têm usado tipos primitivos, como **int**, como parâmetros dos métodos. No entanto, é correta e comum a passagem de objetos para métodos. Por exemplo, o programa a seguir define uma classe chamada **Block** que armazena as dimensões de um bloco tridimensional:

```
// Objetos podem ser passados para os métodos.
class Block {
 int a, b, c;
 int volume;

 Block(int i, int j, int k) {
 a = i;
```

```

b = j;
c = k;
volume = a * b * c;
}

// Retorna true se ob definir o mesmo bloco.
boolean sameBlock(Block ob) { ← Usa um tipo de objeto no parâmetro.
 if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
 else return false;
}

// Retorna true se ob tiver o mesmo volume.
boolean sameVolume(Block ob) { ←
 if(ob.volume == volume) return true;
 else return false;
}
}

class PassOb {
 public static void main(String[] args) {
 Block ob1 = new Block(10, 2, 5);
 Block ob2 = new Block(10, 2, 5);
 Block ob3 = new Block(4, 5, 5);

 System.out.println("ob1 same dimensions as ob2: " +
 ob1.sameBlock(ob2)); ← Passa um objeto.
 System.out.println("ob1 same dimensions as ob3: " +
 ob1.sameBlock(ob3)); ←
 System.out.println("ob1 same volume as ob3: " +
 ob1.sameVolume(ob3)); ←
 }
}

```

Esse programa gera a saída abaixo:

```

ob1 same dimensions as ob2: true
ob1 same dimensions as ob3: false
ob1 same volume as ob3: true

```

Os métodos **sameBlock()** e **sameVolume()** comparam o objeto **Block** passado como parâmetro para o objeto chamador. Em **sameBlock()**, as dimensões dos objetos são comparadas e **true** é retornado apenas quando os dois blocos são idênticos. Em **sameVolume()**, os dois blocos só são comparados para sabermos se eles têm o mesmo volume. Nos dois casos, observe que o parâmetro **ob** especifica **Block** como seu tipo. Embora **Block** seja um tipo de classe criado pelo programa, é usado da mesma forma que os tipos internos de Java.

## COMO OS ARGUMENTOS SÃO PASSADOS

Como o exemplo anterior demonstrou, é tarefa simples passar um objeto para um método. No entanto, há algumas nuances na passagem de um objeto que não são

mostradas no exemplo. Em certos casos, os efeitos da passagem de um objeto serão diferentes dos vivenciados na passagem de argumentos que não sejam objetos. Para ver o porquê, será útil começarmos descrevendo resumidamente duas maneiras pelas quais uma linguagem de computador pode passar um argumento para uma sub-rotina.

A primeira maneira é a *chamada por valor*. Essa abordagem copia o *valor* de um argumento no parâmetro formal da sub-rotina. Portanto, alterações feitas no parâmetro da sub-rotina não têm efeito sobre o argumento da chamada. A segunda maneira de um argumento poder ser passado é a *chamada por referência*. Nessa abordagem, uma referência a um argumento (e não o valor do argumento) é passada para o parâmetro. Dentro da sub-rotina, essa referência é usada no acesso ao argumento real especificado na chamada. Ou seja, alterações feitas no parâmetro afetarão o argumento usado para chamar a sub-rotina. Como você verá, embora Java use a chamada por valor para passar argumentos, o efeito exato produzido difere entre a passagem de um tipo primitivo ou um tipo de referência.

Quando passamos um tipo primitivo, como **int** ou **double**, para um método, seu valor é passado para o parâmetro. Portanto, uma cópia do argumento é feita e o que ocorre ao parâmetro recebedor do argumento não tem efeito fora do método. Por exemplo, considere o programa a seguir:

```
// Tipos primitivos são passados por valor.
class Test {
 /* Este método não causa alteração nos argumentos
 usados na chamada. */
 void noChange(int i, int j) {
 i = i + j;
 j = -j;
 }
}

class CallByValue {
 public static void main(String[] args) {
 Test ob = new Test();

 int a = 15, b = 20;

 System.out.println("a and b before call: " +
 a + " " + b);

 ob.noChange(a, b);

 System.out.println("a and b after call: " +
 a + " " + b);
 }
}
```

A saída do programa é mostrada aqui:

```
a and b before call: 15 20
a and b after call: 15 20
```

Como você pode ver, as operações que ocorrem dentro de **noChange()** não têm efeito sobre os valores de **a** e **b** usados na chamada.

Quando passamos um objeto para um método, a situação muda drasticamente. Primeiro, lembre-se de que, quando criamos uma variável de um tipo de classe, estamos criando uma referência. Quando passamos um objeto como argumento, não estamos passando o objeto propriamente dito, mas a referência que aponta para esse objeto. Ou seja, se passarmos uma referência de objeto para um método, o parâmetro que a receber referenciará o *mesmo* objeto referenciado pelo argumento. Portanto, na verdade os objetos são passados para os métodos pela chamada por referência (já que só a referência é passada). Alterações no objeto dentro do método *afetam* o objeto usado como argumento. Por exemplo, considere o programa abaixo:

```
// Objetos são passados por suas referências.
class Test {
 int a, b;

 Test(int i, int j) {
 a = i;
 b = j;
 }
 /* Passa um objeto. Agora, os valores ob.a e ob.b do objeto
 usados na chamada serão alterados. */
 void change(Test ob) {
 ob.a = ob.a + ob.b;
 ob.b = -ob.b;
 }
}

class PassObjRef {
 public static void main(String[] args) {
 Test ob = new Test(15, 20);

 System.out.println("ob.a and ob.b before call: " +
 ob.a + " " + ob.b);

 ob.change(ob);

 System.out.println("ob.a and ob.b after call: " +
 ob.a + " " + ob.b);
 }
}
```

Esse programa gera a saída abaixo:

```
| ob.a and ob.b before call: 15 20
| ob.a and ob.b after call: 35 -20
```

Como você pode ver, nesse caso, as ações ocorridas dentro de **change()** afetaram o objeto passado para o método.

Resumindo: quando uma referência de objeto é passada para um método, a própria referência é passada com o uso da chamada por valor. Logo, o parâmetro rece-

be uma *cópia* da referência usada como argumento. Como resultado, uma alteração no parâmetro (como se o fizéssemos referenciar um objeto diferente) não afetará a referência usada como argumento. No entanto, já que tanto o parâmetro quanto o argumento referenciam o *mesmo objeto*, uma alteração *através do parâmetro* afetará o objeto referenciado pelo argumento.

### Pergunte ao especialista

**P** Há alguma maneira de passar um tipo primitivo por referência?

**R** Não diretamente. No entanto, Java define um conjunto de classes que *encapsulam* tipos primitivos em objetos. Elas são **Double**, **Float**, **Byte**, **Short**, **Integer**, **Long** e **Character**. Além de permitir que um tipo primitivo seja passado por referência, essas classes encapsuladoras definem vários métodos que permitem o tratamento de seus valores. Por exemplo, os encapsuladores de tipos numéricos incluem métodos que convertem um valor numérico de sua forma binária para um string legível por humanos e vice-versa.

### Verificação do progresso

1. Qual é a diferença entre chamada por valor e chamada por referência?
2. Como Java passa tipos primitivos? E objetos?

## RETORNANDO OBJETOS

Um método pode retornar qualquer tipo de dado, inclusive tipos de classe. Por exemplo, a classe **ErrorMsg** mostrada aqui poderia ser usada para relatar erros. Seu método, **getErrorMsg()**, retorna uma referência a um objeto **String** contendo a descrição de um erro com base no código de erro recebido.

```
// Retorna um objeto String.
class ErrorMsg {
 String[] msgs = {
 "Output Error",
 "Input Error",
 "Disk Full",
 "Index Out-Of-Bounds"
 };

 // Retorna a mensagem de erro.
 String getErrorMsg(int i) { ←———— Retorno um objeto de tipo String.
 if(i >= 0 & i < msgs.length)
 return msgs[i];
 }
}
```

### Respostas:

1. Na chamada por valor, uma cópia do argumento é passada para uma sub-rotina. Na chamada por referência, uma referência ao argumento é passada.
2. Java passa tipos primitivos por valor. Um objeto é passado por intermédio de sua referência. (A própria referência é passada por valor.)

```

 else
 return "Invalid Error Code";
 }
}

class ErrMsgDemo {
 public static void main(String[] args) {
 ErrMsg err = new ErrMsg();

 System.out.println(err.getErrorMsg(2));
 System.out.println(err.getErrorMsg(19));
 }
}

```

Sua saída é mostrada abaixo:

```

Disk Full
Invalid Error Code

```

É claro que você também pode retornar objetos de classes que criar. Por exemplo, esta é uma versão retrabalhada do programa anterior que cria duas classes de erro. Uma se chama **Err** e encapsula uma mensagem de erro junto com um código de gravidade. A segunda se chama **ErrorInfo**. Ela define um método chamado **getErrorInfo()**, que retorna uma referência a um objeto **Err**.

```

// Retorna um objeto definido pelo programador.

class Err {
 String msg; // mensagem de erro
 int severity; // código indicando a gravidade do erro

 Err(String m, int s) {
 msg = m;
 severity = s;
 }
}

class ErrorInfo {
 String[] msgs = {
 "Output Error",
 "Input Error",
 "Disk Full",
 "Index Out-Of-Bounds"
 };
 int[] howbad = { 3, 3, 2, 4 };

 Err getErrorInfo(int i) { ←———— Retorna um objeto de tipo Err.
 if(i >= 0 & i < msgs.length)
 return new Err(msgs[i], howbad[i]);
 else
 return new Err("Invalid Error Code", 0);
 }
}

```

```
}

class ErrInfoDemo {
 public static void main(String[] args) {
 ErrorInfo err = new ErrorInfo();
 Err e;

 e = err.getErrorInfo(2);
 System.out.println(e.msg + " severity: " + e.severity);

 e = err.getErrorInfo(19);
 System.out.println(e.msg + " severity: " + e.severity);
 }
}
```

Aqui está a saída:

```
Disk Full severity: 2
Invalid Error Code severity: 0
```

Sempre que `getErrorInfo()` é chamado, um novo objeto `Err` é criado e uma referência a ele é retornada para a rotina chamadora. Esse objeto é então usado dentro de `main()` para exibir a mensagem de erro e o código de gravidade.

Quando um objeto é retornado por um método, ele continua existindo até não ser mais referenciado. Nesse momento, é alvo da coleta de lixo. Logo, um objeto não será destruído só porque o método que o criou foi encerrado.

## SOBRECARGA DE MÉTODOS

Nesta seção, você conhecerá um recurso Java fascinante: a sobrecarga de métodos. Em Java, dois ou mais métodos da mesma classe podem compartilhar o mesmo nome, desde que suas declarações de parâmetros sejam diferentes. Quando é esse o caso, diz-se que os métodos são *sobre carregados* e o processo é chamado de *sobrecarga de método*. A sobrecarga de métodos é uma das maneiras pelas quais Java implementa o polimorfismo.

Em geral, para sobre carregar um método, só temos que declarar versões diferentes dele. O compilador se incumbe do resto. Porém, é preciso prestar atenção em uma restrição importante: o tipo e/ou a quantidade dos parâmetros de cada método sobre carregado devem diferir. Não é o bastante dois métodos diferirem apenas em seus tipos de retorno. (Os tipos de retorno não fornecem informações suficientes em todos os casos para Java decidir que método usar.) Mas os métodos sobre carregados também *podem* diferir em seus tipos de retorno. Quando um método sobre carregado é chamado, sua versão cujos parâmetros coincidem com os argumentos é executada.

Aqui está um exemplo simples que ilustra a sobrecarga de métodos:

```
// Demonstra a sobre carga de métodos.
class Overload {
 void ovlDemo() { ← Primeira versão.
 System.out.println("No parameters");
 }
}
```

```

// Sobrecreve ovlDemo para um parâmetro inteiro.
void ovlDemo(int a) { ← Segunda versão.
 System.out.println("One parameter: " + a);
}

// Sobrecreve ovlDemo para dois parâmetros inteiros.
int ovlDemo(int a, int b) { ← Terceira versão.
 System.out.println("Two parameters: " + a + " " + b);
 return a + b;
}

// Sobrecreve ovlDemo para dois parâmetros double.
double ovlDemo(double a, double b) { ← Quarta versão.
 System.out.println("Two double parameters: " +
 a + " " + b);
 return a + b;
}

class OverloadDemo {
 public static void main(String[] args) {
 Overload ob = new Overload();
 int resI;
 double resD;

 // chama todas as versões de ovlDemo()
 ob.ovlDemo();
 System.out.println();

 ob.ovlDemo(2);
 System.out.println();

 resI = ob.ovlDemo(4, 6);
 System.out.println("Result of ob.ovlDemo(4, 6): " +
 resI);
 System.out.println();

 resD = ob.ovlDemo(1.1, 2.32);
 System.out.println("Result of ob.ovlDemo(1.1, 2.32): " +
 resD);
 }
}

```

Esse programa gera a saída a seguir:

```

No parameters

One parameter: 2

Two parameters: 4 6
Result of ob.ovlDemo(4, 6): 10

```

```
Two double parameters: 1.1 2.32
Result of ob.ovlDemo(1.1, 2.32): 3.42
```

Como ficou claro, **ovlDemo()** é sobre carregado quatro vezes. A primeira versão não usa parâmetros, a segunda recebe um parâmetro inteiro, a terceira recebe dois parâmetros inteiros e a quarta usa dois parâmetros **double**. Observe que as duas primeiras versões de **ovlDemo()** retornam **void** e as outras duas retornam um valor. Isso é perfeitamente válido, mas, como explicado, a sobrecarga não é afetada pelo tipo de retorno de um método. Logo, a tentativa de usar as duas versões a seguir de **ovlDemo()** causará erro:

```
// É correto usar um método ovlDemo(int).
void ovlDemo(int a) { ←
 System.out.println("One parameter: " + a);
}

/* Erro! Não é correto usar dois métodos ovlDemo(int)
 mesmo que os tipos de retorno sejam diferentes.
*/
int ovlDemo(int a) { ←
 System.out.println("One parameter: " + a);
 return a * a;
}
```

Os tipos de retorno não podem ser usados para diferenciar métodos sobre carregados.

Como os comentários sugerem, a diferença nos tipos de retorno é insuficiente no caso da sobrecarga.

Pelo que vimos no Capítulo 2, Java fornece algumas conversões de tipo automáticas. Essas conversões também são aplicáveis a parâmetros de métodos sobre carregados. Por exemplo, considere o seguinte:

```
/* Conversões de tipo automáticas podem afetar
 a definição do método sobre carregado.
*/
class Overload2 {
 void f(int x) {
 System.out.println("Inside f(int): " + x);
 }

 void f(double x) {
 System.out.println("Inside f(double): " + x);
 }
}

class TypeConv {
 public static void main(String[] args) {
 Overload2 ob = new Overload2();

 int i = 10;
 double d = 10.1;
 byte b = 99;
 short s = 10;
```

```

 float f = 11.5F;

 ob.f(i); // chama ob.f(int)
 ob.f(d); // chama ob.f(double)

 ob.f(b); // chama ob.f(int) - conversão de tipo
 ob.f(s); // chama ob.f(int) - conversão de tipo
 ob.f(f); // chama ob.f(double) - conversão de tipo
}
}

```

A saída do programa é mostrada aqui:

```

Inside f(int): 10
Inside f(double): 10.1
Inside f(int): 99
Inside f(int): 10
Inside f(double): 11.5

```

Nesse exemplo, só duas versões de **f()** são definidas: uma com parâmetro **int** e outra com parâmetro **double**. No entanto, é possível passar para **f()** um valor **byte**, **short** ou **float**. No caso de **byte** e **short**, Java os converte automaticamente em **int**. Logo, **f(int)** é chamado. No caso de **float**, o valor é convertido para **double** e **f(double)** é chamado.

Porém, é importante entender que as conversões automáticas só são aplicáveis quando não há correspondência direta entre um parâmetro e um argumento. Por exemplo, este é o programa anterior com a inclusão de uma versão de **f()** que especifica um parâmetro **byte**:

```

// Adiciona f(byte).
class Overload2 {
 void f(byte x) {
 System.out.println("Inside f(byte): " + x);
 }

 void f(int x) {
 System.out.println("Inside f(int): " + x);
 }

 void f(double x) {
 System.out.println("Inside f(double): " + x);
 }
}

class TypeConv {
 public static void main(String[] args) {
 Overload2 ob = new Overload2();

 int i = 10;
 double d = 10.1;
 byte b = 99;
 short s = 10;
 }
}

```

```

float f = 11.5F;

ob.f(i); // chama ob.f(int)
ob.f(d); // chama ob.f(double)

ob.f(b); // chama ob.f(byte) - agora, sem conversão de tipo

ob.f(s); // chama ob.f(int) - conversão de tipo
ob.f(f); // chama ob.f(double) - conversão de tipo
}
}

```

Agora, quando o programa é executado, a saída a seguir é produzida:

```

Inside f(int): 10
Inside f(double): 10.1
Inside f(byte): 99
Inside f(int): 10
Inside f(double): 11.5

```

Nessa versão, como há uma variante de **f( )** que recebe um argumento **byte**, quando **f( )** é chamado com esse argumento, **f(byte)** é chamado e não ocorre a conversão automática para **int**.

A sobrecarga de métodos dá suporte ao polimorfismo, porque é uma maneira de Java implementar o paradigma “uma interface, vários métodos”. Para entender como, considere o seguinte: em linguagens que não dão suporte à sobrecarga de métodos, cada método deve receber um nome exclusivo. O problema é que, muitas vezes, queremos implementar um conjunto de métodos relacionados, como quando cada método difere um do outro apenas em termos dos dados que estão sendo tratados. Considere o método que retorna o valor absoluto de seu argumento. Em linguagens que não dão suporte à sobrecarga, geralmente há três ou mais versões desse método, cada uma com um nome um pouco diferente. Por exemplo, na linguagem C (que não dá suporte à sobrecarga de métodos), o método **abs( )** retorna o valor absoluto de um inteiro, **labs( )** retorna o valor absoluto de um inteiro longo e **fabs( )** retorna o valor absoluto de um valor de ponto flutuante. Uma vez que C não dá suporte à sobrecarga, cada método deve ter seu próprio nome, ainda que os três façam essencialmente a mesma coisa.

É claro que, conceitualmente, o uso de três nomes torna a situação mais complicada do que já é. Embora o conceito subjacente a todos os métodos seja igual (retornar o valor absoluto), temos três nomes para lembrar. Essa situação não ocorre em Java, porque todos os métodos do valor absoluto podem usar o mesmo nome. Na verdade, a biblioteca padrão de classes Java inclui um método do valor absoluto, chamado **abs( )**. Esse método é sobrecarregado pela classe Java **Math** para tratar todos os tipos numéricos. Java determina que versão de **abs( )** será chamada com base no tipo de argumento.

A vantagem da sobrecarga é ela permitir que métodos relacionados sejam acessados com o uso de um nome comum. Portanto, o nome **abs** representa a *ação geral* que está sendo executada. A seleção da versão correta *específica* para uma determinada circunstância é deixada para o compilador. Você, o programador, só tem que lembrar a operação geral. Com a aplicação do polimorfismo, vários nomes foram

reduzidos para um. Embora esse exemplo seja muito simples, se você expandir o conceito, verá como a sobrecarga ajuda a gerenciar uma complexidade ainda maior.

Quando você sobreporcarregar um método, cada versão dele poderá executar qualquer atividade desejada. Não há uma regra declarando que os métodos sobreporcarregados devem estar relacionados. No entanto, de um ponto de vista estilístico, a sobrecarga de métodos implica um relacionamento. Assim, embora você possa usar o mesmo nome para sobreporcarregar métodos não relacionados, não deve fazê-lo. Por exemplo, você poderia usar o nome `sqr` para criar métodos que retornassem o *quadrado* de um inteiro e a *raiz quadrada* de um valor de ponto flutuante. Mas essa duas operações são basicamente diferentes. A aplicação da sobrecarga de métodos dessa maneira frustra seu objetivo original. Na prática, você só deve sobreporcarregar operações intimamente relacionadas.

### Pergunte ao especialista

**P** Ouvi o termo *assinatura* sendo usado por programadores de Java. Do que se trata?

**R** No contexto Java, uma assinatura é o nome de um método mais sua lista de parâmetros. Logo, para fins de sobreporcarregagem, dois métodos da mesma classe não podem ter a mesma assinatura. É bom ressaltar que uma assinatura não inclui o tipo de retorno, já que ele não é usado por Java para a definição da sobreporcarregagem.

## SOBREPORCARREGANDO CONSTRUTORES

Como os métodos, os construtores também podem ser sobreporcarregados. Isso permite a construção de objetos de várias maneiras. Por exemplo, considere o programa a seguir:

```
// Demonstra um construtor sobreporcarregado.
class MyClass {
 int x;

 MyClass() { ← Constrói objetos de várias maneiras.
 System.out.println("Inside MyClass() .");
 x = 0;
 }

 MyClass(int i) { ←
 System.out.println("Inside MyClass(int) .");
 x = i;
 }

 MyClass(double d) { ←
 System.out.println("Inside MyClass(double) .");
 x = (int) d;
 }

 MyClass(int i, int j) { ←

```

```
 System.out.println("Inside MyClass(int, int).");
 x = i * j;
 }
}

class OverloadConsDemo {
 public static void main(String[] args) {
 MyClass t1 = new MyClass();
 MyClass t2 = new MyClass(88);
 MyClass t3 = new MyClass(17.23);
 MyClass t4 = new MyClass(2, 4);

 System.out.println("t1.x: " + t1.x);
 System.out.println("t2.x: " + t2.x);
 System.out.println("t3.x: " + t3.x);
 System.out.println("t4.x: " + t4.x);
 }
}
```

A saída do programa é mostrada aqui:

```
Inside MyClass().
Inside MyClass(int).
Inside MyClass(double).
Inside MyClass(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8
```

**MyClass()** é sobreescrito de quatro maneiras, cada uma construindo um objeto diferentemente. O construtor apropriado é chamado de acordo com os parâmetros especificados quando a instrução **new** é executada. Ao sobreescrutar o construtor, damos ao usuário da classe flexibilidade na maneira como os objetos são construídos.

Uma das razões para a sobreescrita de construtores é um objeto poder inicializar outro. Por exemplo, considere esse programa que usa a classe **Summation** para calcular a soma de um valor inteiro:

```
// Inicializa um objeto com outro.
class Summation {
 int sum;

 // Constrói a partir de um int.
 Summation(int num) {
 sum = 0;
 for(int i=1; i <= num; i++)
 sum += i;
 }

 // Constrói a partir de outro objeto.
 Summation(Summation ob) { ←———— Constrói um objeto a partir de outro.
 sum = ob.sum;
```

```

 }
}

class SumDemo {
 public static void main(String[] args) {
 Summation s1 = new Summation(5);
 Summation s2 = new Summation(s1);

 System.out.println("s1.sum: " + s1.sum);
 System.out.println("s2.sum: " + s2.sum);
 }
}

```

A saída é mostrada abaixo:

```

| s1.sum: 15
| s2.sum: 15

```

Muitas vezes, como esse exemplo mostra, uma vantagem de fornecer um construtor que use um objeto para inicializar outro é a eficiência. Nesse caso, quando **s2** é construído, não é necessário recalcular a soma. É claro que, até mesmo em casos em que a eficiência não é um problema, geralmente é útil fornecer um construtor que faça uma cópia de um objeto.

## Verificação do progresso

1. Um construtor pode usar um objeto de sua própria classe como parâmetro?
2. O que faria você querer fornecer construtores sobrecarregados?

### TENTE ISTO 6-2 Sobrecarregando o construtor de SimpleStack

`SimpleStackDemo2.java`

Neste projeto, você melhorará a classe **SimpleStack** dando a ela dois construtores adicionais. O primeiro construirá uma nova pilha a partir de outra. O segundo construirá uma pilha, dando a ela valores iniciais. Como você verá, a inclusão desses construtores vai melhorar a usabilidade de **SimpleStack**.

#### PASSO A PASSO

1. Crie um arquivo chamado **SimpleStackDemo2.java** e copie nele a classe **SimpleStack** atualizada que você criou na seção Tente isto 6-1.

Respostas:

1. Sim.
2. Proporcionar conveniência e flexibilidade ao usuário da classe.

2. Adicione o construtor a seguir, que constrói uma pilha a partir de outra.

```
// Constrói uma pilha a partir de outra.
SimpleStack(SimpleStack otherStack) {
 // o tamanho da nova pilha é igual ao de otherStack
 data = new char[otherStack.data.length];

 // configura tos com a mesma posição
 tos = otherStack.tos;

 // copia o conteúdo
 for(int i = 0; i < tos; i++)
 data[i] = otherStack.data[i];
}
```

Observe atentamente esse construtor. Ele inicializa **tos** com o valor contido no parâmetro **otherStack.tos**. Também aloca um novo array para conter a pilha e copia os elementos de **otherStack** para esse array. Uma vez construída, a nova pilha será uma cópia idêntica da original, mas as duas serão objetos totalmente separados.

3. Adicione o construtor que inicializa a pilha a partir de um array de caracteres, como mostrado aqui:

```
// Constrói uma pilha com valores iniciais.
SimpleStack(char[] chrs) {
 // cria o array para armazenar os valores iniciais
 data = new char[chrs.length];
 tos = 0;

 // inicializa a pilha inserindo nela
 // o conteúdo de chrs
 for(char ch : chrs)
 push(ch);
}
```

Esse construtor cria uma pilha suficientemente grande para conter os caracteres de **chrs** e então armazena-os na pilha chamando **push()**.

4. Esta é a classe **SimpleStack** atualizada e completa junto com a classe **SimpleStackDemo2**, que a demonstra. Observe que essa versão inclui a adição de **private** a **data** e **tos** como descrito na seção Tente isto 6-1.

```
/*
 Tente isto 6-2

 Adiciona construtores sobreporregados a SimpleStack.
*/

class SimpleStack {
 // agora os membros a seguir são privados
 private char[] data; // esse array contém a pilha
```

```
private int tos; // índice do topo da pilha

// Constrói uma pilha vazia dado seu tamanho.
SimpleStack(int size) {
 data = new char[size]; // cria o array para conter a pilha
 tos = 0;
}

// Constrói uma pilha a partir de outra.
SimpleStack(SimpleStack otherStack) {
 // o tamanho da nova pilha é igual ao de otherStack
 data = new char[otherStack.data.length];

 // configura tos com a mesma posição
 tos = otherStack.tos;

 // copia o conteúdo
 for(int i = 0; i < tos; i++)
 data[i] = otherStack.data[i];
}

// Constrói uma pilha com valores iniciais.
SimpleStack(char[] chrs) {
 // cria o array para armazenar os valores iniciais
 data = new char[chrs.length];
 tos = 0;

 // inicializa a pilha inserindo nela
 // o conteúdo de chars
 for(char ch : chrs)
 push(ch);
}

// Insere um caractere na pilha.
void push(char ch) {
 if(isFull()) {
 System.out.println(" -- Stack is full.");
 return;
 }

 data[tos] = ch;
 tos++;
}

// Extrai um caractere da pilha.
char pop() {
 if(isEmpty()) {
 System.out.println(" -- Stack is empty.");
 return (char) 0; // um valor de espaço reservado
 }
 tos--;
 return data[tos];
```

```

 }

 // Retorna true se a pilha estiver vazia.
 boolean isEmpty() {
 return tos==0;
 }

 // Retorna true se a pilha estiver cheia.
 boolean isFull() {
 return tos==data.length;
 }
}

// Demonstra os construtores sobrecarregados da classe SimpleStack.
class SimpleStackDemo2 {
 public static void main(String[] args) {
 int i;
 char ch;

 char[] chrs = { 'A', 'B', 'C', 'D' };

 // Inicializa stack1 com chrs.
 SimpleStack stack1 = new SimpleStack(chrs);

 // Inicializa stack2 com o conteúdo de stack1.
 SimpleStack stack2 = new SimpleStack(stack1);

 System.out.print("Popping contents of stack1: ");
 while(!stack1.isEmpty()) {
 ch = stack1.pop();
 System.out.print(ch);
 }

 System.out.print("\nPopping contents of stack2: ");
 while(!stack2.isEmpty()) {
 ch = stack2.pop();
 System.out.print(ch);
 }
 }
}

```

A saída do programa é mostrada aqui:

```

| Popping contents of stack1: DCBA
| Popping contents of stack2: DCBA

```

## RECURSÃO

Em Java, um método pode chamar a si mesmo. Esse processo se chama *recursão*, e dizemos que um método é *recursivo* quando chama a si próprio. Em geral, recursão é o processo em que algo é definido a partir de si mesmo e é um pouco parecido com uma definição circular. O componente-chave do método recursivo é a instrução que executa uma chamada a esse método. A recursão é um mecanismo de controle poderoso.

Comecemos com um exemplo muito simples que demonstra os elementos-chave da recursão. O método **drawStars()** mostrado a seguir usa um parâmetro inteiro chamado **n** e desenha uma linha de **n** estrelas (na verdade, asteriscos). Ele usa a recursão para controlar o processo.

```
// Usa a recursão para desenhar uma linha de n asteriscos.
void drawStars(int n) {
 if(n == 1)
 System.out.print("*");
 else {
 System.out.print("*");
 drawStars(n-1); // uma chamada recursiva
 }
}
```

Observe que dentro de **drawStars()** é feita outra chamada a **drawStars()**. Isso é o que chamamos de *chamada recursiva*.

Antes de examinar detalhadamente como **drawStars()** funciona, será útil descrevermos o problema de desenhar uma linha de asteriscos usando uma solução recursiva. Para começar, pense no ato de desenhar asteriscos como uma tarefa. Após um asterisco ser desenhado, o que falta fazer? Resposta: desenhar os outros asteriscos. Em outras palavras, a tarefa de desenhar  $N$  asteriscos pode ser dividida na tarefa de desenhar um asterisco seguida pela tarefa de desenhar os outros  $N-1$  asteriscos. É claro que a tarefa de desenhar  $N-1$  asteriscos é simplesmente o processo de desenhar um asterisco seguido pela tarefa de desenhar os outros  $N-2$  asteriscos. O processo é então repetido até que não haja mais asteriscos para desenhar. Essa ideia geral é a essência da recursão. Claro que o processo deve terminar em algum momento. No caso do desenho de asteriscos, ele termina quando  $N$  é igual a 1 e o último asterisco é desenhado.

Agora examinemos como o processo é implementado em código Java. Primeiro, **drawStars()** recebe o número de asteriscos a serem desenhados em seu parâmetro **n**. Dentro do método, se **n** for igual a 1, **drawStars()** desenhárá um asterisco e retornará. Logo, quando **n** é igual a 1, *não é feita* a chamada recursiva. O ponto em que a chamada recursiva *não é feita* se chama *caso base*. Em todos os casos em que **n** é maior do que 1, **drawStars()** desenha um asterisco e então chama a si mesmo recursivamente, passando **n-1** como argumento. Esse processo se repete até o valor passado para **n** ser 1 e as chamadas recursivas começarem a retornar. Por exemplo, uma chamada a **drawStars(3)** exibe um asterisco e chama **drawStars(2)**, que exibe um asterisco e chama **drawStars(1)**. Já que **drawStars(1)** é o caso base, o asterisco é exibido e as chamadas recursivas começam a retornar. Após a chamada inicial a **drawStars()** retornar, uma linha de **n** asteriscos terá sido desenhada.

Há mais um ponto importante que devemos ressaltar sobre **drawStars()**. As chamadas recursivas param quando o caso base é encontrado. Como explicado, em

**drawStars( )**, isso ocorre quando **n** recebe 1. Se o caso base não estivesse presente para impedir a chamada recursiva, **drawStars( )** chamaria a si mesmo infinitamente (até um erro de tempo de execução ocorrer). É por isso que o caso base é importante e todo método recursivo precisa de um.

Aqui está um programa completo que demonstra o método **drawStars( )**.

```
class StarDrawer {
 void drawStars(int n) {
 if(n == 1)
 System.out.print("*");
 else {
 System.out.print("*");
 drawStars(n-1); // uma chamada recursiva
 }
 }
}

class StarDrawingDemo {
 public static void main(String[] args) {
 StarDrawer drawer = new StarDrawer();

 drawer.drawStars(1); // apenas o caso base
 System.out.println();
 drawer.drawStars(2); // uma chamada recursiva
 System.out.println();
 drawer.drawStars(3); // duas chamadas recursivas
 System.out.println();
 drawer.drawStars(10); // nove chamadas recursivas
 System.out.println();
 }
}
```

A saída desse programa é:

```
*
**


```

Agora examinemos um exemplo de recursão um pouco mais complicado. Como regra geral, um método recursivo requer o uso de um parâmetro que ajude a determinar quando uma chamada recursiva ocorrerá. No caso de **drawStars( )**, ele era **n** e recebeu o número de asteriscos a serem desenhados. No entanto, às vezes a forma mais natural de um método não tem esse parâmetro. Por exemplo, considere um método chamado **printArray( )** que use um array de inteiros como parâmetro e exiba os elementos do array. Esse método é fácil de implementar iterativamente com o uso de um laço **for**, mas como você o implementaria recursivamente? Para fazê-lo, precisaria de um caso base que interrompesse as chamadas recursivas. Qual seria o caso base? Já que o array nunca muda, o parâmetro do método também não, e, portanto, cada chamada recursiva ao método usaria o mesmo parâmetro, resultando em uma sucessão infinita de chamadas recursivas. A única coisa que muda durante a exibição do array é o índice do próximo elemento a ser exibido.

A solução para esse tipo de problema é usar um método auxiliar que adicione um parâmetro controlador da recursão. Por exemplo, no caso de `printArray()`, o método auxiliar poderia se chamar `printArrayAux()`. Ele teria o array como seu primeiro parâmetro e, como segundo parâmetro, um inteiro fornecendo o índice do elemento do array a ser exibido a seguir. Esse método auxiliar alcançaria o caso base quando o segundo parâmetro fosse uma unidade maior que o índice do último elemento do array. Aqui está um programa completo que mostra uma implementação desses métodos e demonstra como eles funcionam:

```
class Printer {
 void printArray(int[] array) {
 printArrayAux(array, 0); // começa no elemento zero
 System.out.println();
 }

 void printArrayAux(int[] array, int index) {
 if(index == array.length)
 return; // terminamos
 else { // há mais elementos para exibir
 System.out.print(array[index] + " ");
 printArrayAux(array, index+1);
 }
 }
}

class PrinterDemo {
 public static void main(String[] args) {
 Printer printer = new Printer();
 int[] array = { 3,1,4,2,5,7,6,8 };

 printer.printArray(array);
 }
}
```

A saída desse programa é mostrada abaixo:

```
| 3 1 4 2 5 7 6 8
```

Até agora, os exemplos de métodos recursivos tiveram um tipo de retorno `void`. No entanto, um método recursivo também pode retornar um valor. Um exemplo clássico desse tipo de método é o que calcula e retorna o fatorial de um número. O *fatorial* de um número  $N$  é o produto de todos os números inteiros entre 1 e  $N$ . Por exemplo, o fatorial de 3 é  $1 \times 2 \times 3$ , ou 6. O programa a seguir mostra uma maneira recursiva de calcular o fatorial de um número. Para fins de comparação, um equivalente não recursivo também é mostrado.

```
// Um exemplo simples de recursão.
class Factorial {
 // Esta é uma função recursiva.
 int factR(int n) {
 int result;
```

```

if(n==1) return 1;
result = factR(n-1) * n;
return result;
}

// Este é um equivalente iterativo.
int factI(int n) {
 int t, result;

 result = 1;
 for(t=1; t <= n; t++) result *= t;
 return result;
}

class Recursion {
 public static void main(String[] args) {
 Factorial f = new Factorial();

 System.out.println("Factorials using recursive method.");
 System.out.println("Factorial of 3 is " + f.factR(3));
 System.out.println("Factorial of 4 is " + f.factR(4));
 System.out.println("Factorial of 5 is " + f.factR(5));
 System.out.println();

 System.out.println("Factorials using iterative method.");
 System.out.println("Factorial of 3 is " + f.factI(3));
 System.out.println("Factorial of 4 is " + f.factI(4));
 System.out.println("Factorial of 5 is " + f.factI(5));
 }
}

```

A saída do programa é mostrada abaixo:

```

Factorials using recursive method.
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

Factorials using iterative method.
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

```

A operação do método não recursivo **factI()** deve ter ficado clara. Ele usa um laço começando em 1 e multiplica progressivamente cada número pelo novo produto.

A operação do método recursivo **factR()** é um pouco mais complexa. Quando **factR()** é chamado com um argumento igual a 1, ele retorna 1; caso contrário, retorna o produto de **factR(n-1)\*n**. Para avaliar essa expressão, **factR()** é chamado com **n-1**. Esse processo se repete até **n** ser igual a 1 e as chamadas ao método começarem a retornar. Por exemplo, quando o factorial de 2 é calculado, a primeira chamada a **factR()** faz uma segunda chamada ser feita com o argumento 1. Essa chamada re-

tornará 1, que será então multiplicado por 2 (o valor original de **n**). A resposta será 2. Pode ser interessante inserir instruções **println()** em **factR()** que exibam em que nível cada chamada está e quais são os resultados intermediários.

Embora a recursão possa ser muito útil, ela tem um preço. Sempre que chamamos um método, alguma sobrecarga é adicionada ao programa. Como resultado, um método recursivo produzirá essa sobrecarga sempre que a chamada recursiva for executada. Em alguns casos, isso pode impactar o desempenho a tal ponto que uma solução iterativa seria melhor. A principal vantagem da recursão é que alguns tipos de algoritmos podem ser implementados mais clara e simplesmente de maneira recursiva do que de maneira iterativa. Por exemplo, o algoritmo de classificação rápida é bem difícil de implementar de maneira iterativa. Além disso, em alguns casos, o uso da recursão fornece a maneira mais natural de se resolver um problema.

Ao criar métodos recursivos, é importante lembrar de uma coisa: precisaremos de uma instrução condicional, como **if**, em algum local para forçar o método a retornar sem a chamada recursiva ser executada. (Em outras palavras, devemos estabelecer um caso base.) Se não o fizermos, quando chamarmos o método, ele nunca retornará. Isso resultará em um erro de tempo de execução. A recursão descontrolada é muito comum quando começamos a desenvolver métodos recursivos. Use instruções **println()** à vontade para saber o que está ocorrendo e aborte a execução se perceber que cometeu um erro.

## Pergunte ao especialista

**P** A recursão é fascinante, mas não entendo como Java pode manter todas as chamadas recursivas corretas. Os valores dos parâmetros e variáveis não ficarão misturados de uma chamada para a outra?

**R** Embora os detalhes de como a recursão é tratada por um compilador sejam assunto de um curso mais avançado de programação, vejamos uma descrição breve. Sempre que um método é chamado, o armazenamento de seus parâmetros e variáveis locais é criado, com os parâmetros recebendo os valores dos argumentos. Logo, cada chamada de um método começa com seu próprio conjunto de parâmetros e variáveis locais. Quando um método chama a si próprio, ocorre o mesmo processo. A cada chamada recursiva, um novo conjunto de parâmetros e variáveis locais recebe espaço de armazenamento e o código do método é executado desde o início. Uma chamada recursiva *não faz* uma nova cópia do método. Só os parâmetros e variáveis locais são novos. À medida que cada chamada recursiva retorna, suas variáveis locais e parâmetros são removidos do armazenamento e a execução é retomada no ponto da chamada dentro do método. Poderíamos dizer que, como em um telescópio, os métodos recursivos se expandem e se retraem.

## ENTENDENDO static

Você pode definir um membro de classe para ser usado independentemente de qualquer objeto dessa classe. Como deve saber, normalmente o membro de uma classe deve ser acessado por intermédio de um objeto de sua classe, mas é possível criar um membro para ser usado sem referência a uma instância específica. Esse membro pode

ser considerado como aplicável a uma classe como um todo. Para criar esse tipo de membro, preceda sua declaração com a palavra-chave **static**.

Quando um membro é declarado **static**, pode ser acessado antes de qualquer objeto de sua classe ser criado e sem referência a nenhum objeto. Você pode declarar tanto métodos quanto variáveis como estáticos. O exemplo mais comum de um membro **static** é **main()**. O método **main()** é declarado como **static** porque deve ser chamado pela JVM quando o programa começa. Fora da classe, para usar um membro **static**, você só tem que especificar o nome de sua classe seguido pelo operador ponto. Nenhum objeto precisa ser criado. Por exemplo, se quiser atribuir o valor 10 a uma variável **static** chamada **count** pertencente a uma classe chamada **MyTimer**, use esta linha:

```
| MyTimer.count = 10;
```

Esse formato é semelhante ao usado no acesso a variáveis de instância comuns por meio de um objeto, exceto pelo fato de o nome da classe ser usado. Um método **static** pode ser chamado da mesma maneira – com o uso do operador ponto no nome da classe.

## Variáveis estáticas

Variáveis declaradas como **static** são, basicamente, variáveis globais. Quando um objeto é criado, nenhuma cópia de uma variável **static** é feita. Em vez disso, todas as instâncias da classe compartilham a mesma variável **static**. Aqui está um exemplo que mostra as diferenças entre uma variável **static** e uma variável de instância:

```
// Usa uma variável estática.
class StaticDemo {
 int x; // uma variável de instância comum
 static int y; // uma variável estática ←———— Há uma cópia de y para todos os
 objetos compartilharem

 // retorna a soma da variável de instância x
 // e a variável estática y.
 int sum() {
 return x + y;
 }
}

class SDemo {
 public static void main(String[] args) {
 StaticDemo ob1 = new StaticDemo();
 StaticDemo ob2 = new StaticDemo();

 // Cada objeto tem sua própria cópia de uma variável de instância.
 ob1.x = 10;
 ob2.x = 20;
 System.out.println("Of course, ob1.x and ob2.x " +
 "are independent.");
 System.out.println("ob1.x: " + ob1.x +
 "\nob2.x: " + ob2.x);
 System.out.println();
```

```

// Cada objeto compartilha uma cópia de uma variável estática.
System.out.println("The static variable y is shared.");
StaticDemo.y = 19;
System.out.println("Set StaticDemo.y to 19.");

System.out.println("ob1.sum(): " + ob1.sum());
System.out.println("ob2.sum(): " + ob2.sum());
System.out.println();

StaticDemo.y = 100;
System.out.println("Change StaticDemo.y to 100");
System.out.println("ob1.sum(): " + ob1.sum());
System.out.println("ob2.sum(): " + ob2.sum());
System.out.println();
}
}

```

A saída do programa é mostrada abaixo:

```

Of course, ob1.x and ob2.x are independent.
ob1.x: 10
ob2.x: 20

The static variable y is shared.
Set StaticDemo.y to 19.
ob1.sum(): 29
ob2.sum(): 39

Change StaticDemo.y to 100
ob1.sum(): 110
ob2.sum(): 120

```

Como a saída mostra, a variável **static** **y** é compartilhada tanto por **ob1** quanto por **ob2**. Logo, **sum()** adiciona o mesmo valor de **y** à variável **x** de cada objeto. Além disso, a alteração de **y** afeta todos os objetos (isto é, a classe inteira) e não apenas uma instância específica. Preste atenção em como **y** é acessada por intermédio do nome de sua classe, como mostrado aqui:

```
| StaticDemo.y = 19;
```

Uma vez que **y** é compartilhada por todos os objetos, ela é acessada pelo nome de sua classe e não por uma referência de objeto.

Como as variáveis **static** não dependem de um objeto específico, elas são úteis quando precisamos manter informações que sejam aplicáveis a uma classe inteira. Vejamos um exemplo simples. Ele usa um membro **static** chamado **count** para manter uma contagem do número de objetos **MyClass** que foram criados.

```

// Conta instâncias.
class MyClass {
 // Essa variável estática será incrementada
 // sempre que um objeto MyClass for criado.
 static int count = 0;
}

```

```
MyClass() {
 count++; // incrementa a contagem
}
}

class UseStatic {
 public static void main(String[] args) {
 for(int i=0; i < 3; i++) {
 MyClass obj = new MyClass();
 System.out.println("Number of objects created: " + MyClass.count);
 }
 }
}
```

A saída é mostrada aqui:

```
Number of objects created: 1
Number of objects created: 2
Number of objects created: 3
```

Sempre que um objeto **MyClass** é criado, a variável **count** é incrementada. Já que ela é **static**, é usada por todas as instâncias de **MyClass**. Como resultado, armazena uma contagem progressiva do número de objetos **MyClass** que foram instanciados. Não há como fazer isso usando uma variável de instância porque cada objeto tem sua própria cópia de cada variável de instância. Uma variável **static** é aplicável à classe inteira.

## Métodos estáticos

Métodos declarados como **static** são, essencialmente, métodos globais. Eles são chamados independentemente de qualquer objeto. Em vez disso, um método **static** é chamado com o uso do nome de sua classe. Este é um exemplo que cria um método **static**. Observe como ele é chamado dentro de **main()**.

```
// Usa um método estático.
class StaticMeth {
 static int val = 1024; // uma variável estática

 // um método estático.
 static int valDiv2() {
 return val/2;
 }
}

class SDemo2 {
 public static void main(String[] args) {

 System.out.println("val is " + StaticMeth.val);
 System.out.println("StaticMeth.valDiv2(): " +
 StaticMeth.valDiv2());
```

```

 StaticMeth.val = 4;
 System.out.println("val is " + StaticMeth.val);
 System.out.println("StaticMeth.valDiv2(): " +
 StaticMeth.valDiv2());
}
}

```

A saída é mostrada aqui:

```

val is 1024
StaticMeth.valDiv2(): 512
val is 4
StaticMeth.valDiv2(): 2

```

Como o programa mostra, já que **valDiv2()** é declarado como **static**, pode ser chamado sem nenhuma instância de sua classe, **StaticMeth**, ser criada.

Os métodos estáticos ajudam principalmente na criação de métodos utilitários que executem funções úteis não relacionadas a um objeto específico. Vários exemplos são encontrados na classe **Math** padrão. Ela define um grande número de métodos **static** que executam vários cálculos matemáticos. Você viu um exemplo: o método **sqrt()**. Outros incluem funções trigonométricas como **cos()**, **sin()** e **tan()**, o método **abs()**, que retorna o valor absoluto, e **log()**, que retorna o logaritmo natural de um valor. Posteriormente você encontrará outros exemplos de métodos **static** da biblioteca Java.

Métodos declarados como **static** têm várias restrições:

- Só podem chamar diretamente outros métodos **static**.
- Só podem acessar diretamente dados **static**.
- Não têm uma referência **this**.

Por exemplo, na classe a seguir, o método **static valDivDenom()** é inválido:

```

class StaticError {
 int denom = 3; // uma variável de instância comum
 static int val = 1024; // uma variável estática

 /* Erro! Não pode acessar uma variável não estática
 de dentro de um método estático. */
 static int valDivDenom() {
 return val/denom; // não será compilado
 }
}

```

Aqui, **denom** é uma variável de instância comum que não pode ser acessada dentro de um método **static**.

## Blocos estáticos

Uma classe pode precisar de algum tipo de inicialização antes de estar pronta para criar objetos. Por exemplo, ela pode ter que estabelecer uma conexão com um site remoto. Também pode ter que inicializar certas variáveis **static** antes de seus métodos **static** serem usados. Para tratar esses tipos de situações, Java permite que você

declare um bloco **static**. Um bloco **static** é executado quando a classe é carregada pela primeira vez. Portanto, ele é executado antes de a classe poder ser usada para qualquer outro fim. Aqui está um exemplo de um bloco **static**:

```
// Usa um bloco estático
class StaticBlock {
 static double rootOf2;
 static double rootOf3;

 static { ←
 System.out.println("Inside static block.");
 rootOf2 = Math.sqrt(2.0);
 rootOf3 = Math.sqrt(3.0);
 }

 StaticBlock(String msg) {
 System.out.println(msg);
 }
}

class SDemo3 {
 public static void main(String[] args) {
 StaticBlock ob = new StaticBlock("Inside Constructor");

 System.out.println("Square root of 2 is " +
 StaticBlock.rootOf2);
 System.out.println("Square root of 3 is " +
 StaticBlock.rootOf3);
 }
}
```

Esse bloco é executado quando a classe é carregada.

A saída é mostrada abaixo:

```
Inside static block.
Inside Constructor
Square root of 2 is 1.4142135623730951
Square root of 3 is 1.7320508075688772
```

Como podemos ver, o bloco **static** é executado antes de qualquer objeto ser construído.

## Verificação do progresso

1. Defina recursão.
2. Explique a diferença entre variáveis **static** e variáveis de instância.
3. Quando um bloco **static** é executado?

**Respostas:**

1. Recursão é o processo de um método chamar a si mesmo.
2. Cada objeto de uma classe tem sua própria cópia das variáveis de instância definidas pela classe e compartilha uma cópia de uma variável **static**.
3. Um bloco **static** é executado quando sua classe é carregada pela primeira vez, antes de ser usada.

### TENTE ISTO 6-3 A classificação rápida

`QSDemo.java`

No Capítulo 5, você viu um método de classificação simples chamado classificação de bolha. Foi mencionado naquele momento que existem classificações significativamente melhores. Aqui, você desenvolverá uma versão de uma das melhores: a classificação rápida (*quicksort*), inventada por C.A.R. Hoare. Não a mostramos no Capítulo 5 porque a melhor implementação da classificação rápida se baseia na recursão. A versão que desenvolveremos classifica um array de caracteres, mas a lógica pode ser adaptada para classificar qualquer tipo de objeto.

A classificação rápida se baseia na ideia de partições. O procedimento geral envolve a seleção de um valor, chamado *comparando*, e depois é feita a divisão do array em duas seções. Todos os elementos maiores ou iguais ao comparando são inseridos em um lado e os menores são inseridos no outro. Esse processo é repetido para cada seção remanescente até o array estar classificado. Por exemplo, dado o array **fedacb** e usando o valor **d** como comparando, a primeira passagem da classificação rápida reorganizaria o array como mostrado a seguir:

|            |             |
|------------|-------------|
| Inicial    | f e d a c b |
| Passagem 1 | b c a d e f |

Esse processo é então repetido para cada seção – isto é, **bca** e **def**. Como você pode ver, o processo é essencialmente recursivo em sua natureza, e é por isso que a implementação mais limpa da classificação rápida é recursiva.

Você pode selecionar o valor do comparando de duas maneiras. Pode selecioná-lo aleatoriamente ou achando a média de um pequeno conjunto de valores tirados do array. Para obter uma classificação ótima, deve selecionar um valor que esteja exatamente no meio do intervalo de valores. No entanto, não é fácil fazer isso na maioria dos conjuntos de dados. O pior caso é quando o valor selecionado está em uma extremidade. Mesmo assim, a classificação rápida será executada corretamente. A versão da classificação rápida que desenvolveremos seleciona o elemento do meio do array como comparando.

#### PASSO A PASSO

1. Crie um arquivo chamado **QSDemo.java**.
2. Primeiro, crie a classe **Quicksort** mostrada aqui:

```
// Tente isto 6-3: Uma versão simples da classificação rápida.
class Quicksort {

 // Define uma chamada ao método real de classificação rápida.
 static void qsort(char[] items) {
 qs(items, 0, items.length-1);
 }

 // Uma versão recursiva da classificação rápida para caracteres.
 private static void qs(char[] items, int left, int right)
```

```

{
 int i, j;
 char x, y;

 i = left; j = right;
 x = items[(left+right)/2];

 do {
 while((items[i] < x) && (i < right)) i++;
 while((x < items[j]) && (j > left)) j--;

 if(i <= j) {
 y = items[i];
 items[i] = items[j];
 items[j] = y;
 i++; j--;
 }
 } while(i <= j);

 if(left < j) qs(items, left, j);
 if(i < right) qs(items, i, right);
}
}

```

Para manter simples a interface da classificação rápida, a classe **Quicksort** fornece o método **qs()**, que define uma chamada ao método real de classificação rápida, **qs()**. Isso permite que a classificação rápida seja chamada apenas com o nome do array a ser classificado, sem ser preciso fornecer uma partição inicial. Já que **qs()** só é usado internamente, é especificado como **private**.

3. Para usar **Quicksort**, só precisamos chamar **Quicksort.qsort()**. Já que **qsort()** é especificado como **static**, pode ser chamado por intermédio de sua classe em vez de em um objeto. Portanto, não há necessidade de criar um objeto **Quicksort**. Após a chamada retornar, o array estará classificado. Lembre-se, essa versão só funciona para arrays de caracteres, mas você pode adaptar a lógica para classificar qualquer tipo de array.
4. Aqui está um programa que demonstra **Quicksort**:

```

// Tente isto 6-3: Uma versão simples da classificação rápida.

class Quicksort {

 // Define uma chamada ao método real de classificação rápida.
 static void qsort(char[] items) {
 qs(items, 0, items.length-1);
 }

 // Uma versão recursiva da classificação rápida para caracteres.
 private static void qs(char[] items, int left, int right)
 {
 int i, j;
 char x, y;
 }
}

```

```

 i = left; j = right;
 x = items[(left+right)/2];

 do {
 while((items[i] < x) && (i < right)) i++;
 while((x < items[j]) && (j > left)) j--;

 if(i <= j) {
 y = items[i];
 items[i] = items[j];
 items[j] = y;
 i++; j--;
 }
 } while(i <= j);

 if(left < j) qs(items, left, j);
 if(i < right) qs(items, i, right);
}
}

class QSDemo {
 public static void main(String[] args) {
 char[] a = { 'd', 'x', 'a', 'r', 'p', 'j', 'i' };
 int i;

 System.out.print("Original array: ");
 for(i=0; i < a.length; i++)
 System.out.print(a[i]);

 System.out.println();

 // agora, classifica o array
 Quicksort.qsort(a);

 System.out.print("Sorted array: ");
 for(i=0; i < a.length; i++)
 System.out.print(a[i]);
 }
}

```

## INTRODUÇÃO ÀS CLASSES ANINHADAS E INTERNAS

Em Java você pode definir uma *classe aninhada*. Trata-se de uma classe que é declarada dentro de outra. Uma classe aninhada não existe independentemente da classe que a contém. Logo, o escopo da classe aninhada é limitado por sua classe externa. Uma classe aninhada que é declarada diretamente dentro do escopo de sua classe externa é membro dessa classe. Também é possível declarar uma classe aninhada que seja local de um bloco.

Há dois tipos gerais de classes aninhadas: as que são precedidas pelo modificador **static** e as que não o são. O único tipo em que estamos interessados neste livro é o não estático. Esse tipo de classe aninhada também é chamado de *classe interna*. Ela

tem acesso a todas as variáveis e métodos de sua classe externa e pode referenciá-los diretamente, como fazem outros membros não **static** da classe externa.

Às vezes, uma classe interna é usada para fornecer um conjunto de serviços que só é usado por sua classe externa. Aqui está um exemplo que usa uma classe interna para calcular valores para sua classe externa:

```
// Usa uma classe interna.
class Outer {
 int[] nums;

 Outer(int[] n) {
 nums = n;
 }

 void analyze() {
 Inner inOb = new Inner();

 System.out.println("Minimum: " + inOb.min());
 System.out.println("Maximum: " + inOb.max());
 System.out.println("Average: " + inOb.avg());
 }
}

// Esta é uma classe interna.
class Inner { ← Uma classe interna.
 // Retorna o valor mínimo.
 int min() {
 int m = nums[0];

 for(int i=1; i < nums.length; i++)
 if(nums[i] < m) m = nums[i];

 return m;
 }

 // Retorna o valor máximo.
 int max() {
 int m = nums[0];
 for(int i=1; i < nums.length; i++)
 if(nums[i] > m) m = nums[i];

 return m;
 }

 // Retorna a média.
 int avg() {
 int a = 0;
 for(int i=0; i < nums.length; i++)
 a += nums[i];
 }
}
```

```

 return a / nums.length;
 }
}

class NestedClassDemo {
 public static void main(String[] args) {
 int[] x = { 3, 2, 1, 5, 6, 9, 7, 8 };
 Outer outOb = new Outer(x);

 outOb.analyze();
 }
}

```

A saída do programa é mostrada abaixo:

```

Minimum: 1
Maximum: 9
Average: 5

```

Nesse exemplo, a classe interna **Inner** calcula diversos valores a partir do array **nums**, que é membro de **Outer**. Como explicado, uma classe interna tem acesso aos membros de sua classe externa, logo, é perfeitamente aceitável **Inner** acessar o array **nums** diretamente. No entanto, o contrário não é verdade. Por exemplo, não seria possível **analyze()** chamar o método **min()** diretamente, sem a criação de um objeto **Inner**.

Como mencionado, podemos aninhar uma classe dentro de um escopo de bloco. Isso simplesmente cria uma classe localizada que não é conhecida fora de seu bloco. O exemplo a seguir adapta a classe **BitOut** desenvolvida na seção Tente isto 5-3 para uso como classe local.

```

// Usa BitOut como classe local.
class LocalClassDemo {
 public static void main(String[] args) {

 // Uma versão de BitOut como classe interna.
 class BitOut { ← Uma classe local alinhada
 int numBits;

 BitOut(int n) {
 if(n < 1) n = 1;
 if(n > 64) n = 64;
 numBits = n;
 }
 void show(long val) {
 long mask = 1;

```

```
// desloca uma unidade para a esquerda para a posição apropriada
mask <= numBits-1;

int spacer = 8 - (numBits % 8);
for(; mask != 0; mask >>= 1) {
 if((val & mask) != 0) System.out.print("1");
 else System.out.print("0");
 spacer++;
 if((spacer % 8) == 0) {
 System.out.print(" ");
 spacer = 0;
 }
}
System.out.println();
}

for(byte b = 0; b < 10; b++) {
 BitOut byteval = new BitOut(8);

 System.out.print(b + " in binary: ");
 byteval.show(b);
}
}
```

A saída dessa versão do programa é mostrada aqui:

```
0 in binary: 00000000
1 in binary: 00000001
2 in binary: 00000010
3 in binary: 00000011
4 in binary: 00000100
5 in binary: 00000101
6 in binary: 00000110
7 in binary: 00000111
8 in binary: 00001000
9 in binary: 00001001
```

Nesse exemplo, a classe **BitOut** não é conhecida fora de **main()** e qualquer tentativa de acessá-la feita por um método que não for **main()** resultará em erro.

Um último ponto: você pode criar uma classe interna sem nome. É a chamada *classe interna anônima*. Um objeto de uma classe interna anônima é instanciado quando a classe é declarada com o uso de **new**. As classes internas anônimas serão discutidas com mais detalhes na Parte II deste livro, quando o tratamento de eventos com Swing for descrito.

## Pergunte ao especialista

**P** O que torna uma classe aninhada **static** diferente de uma não **static**?

**R** Uma classe aninhada **static** é aquela à qual o modificador **static** é aplicado. Por ser **static**, ela só pode acessar diretamente outros membros **static** da classe onde está contida e deve acessar os membros da sua classe externa por uma referência de objeto.

## Verificação do progresso

1. Uma classe interna tem acesso aos outros membros de sua classe externa. Verdadeiro ou falso?
2. Uma classe aninhada não existe independentemente de sua classe externa. Verdadeiro ou falso?

## VARARGS: ARGUMENTOS EM QUANTIDADE VARIÁVEL

Em algumas situações, podemos querer criar um método que use um número variável de argumentos, de acordo com a sua aplicação exata. Por exemplo, um método que abre uma conexão com a Internet pode receber um nome de usuário, uma senha, um nome de arquivo, um protocolo e assim por diante, mas fornecer padrões se alguma dessas informações não for passada. Nessa situação, seria conveniente passar apenas os argumentos aos quais os padrões não sejam aplicáveis. Um método assim exige alguma maneira de criarmos uma lista de argumentos de tamanho variável em vez de fixo.

No passado, métodos que requeriam uma lista de argumentos de tamanho variável podiam ser tratados de duas maneiras, nenhuma particularmente amigável. Em primeiro lugar, se o número máximo de argumentos fosse pequeno e conhecido, você poderia criar versões sobrecarregadas do método, uma para cada maneira de ele ser chamado. Embora isso funcione e seja adequado para algumas situações, só é aplicável a uma pequena categoria delas. Em casos em que o número máximo de possíveis argumentos era maior, ou desconhecido, uma segunda abordagem era usada na qual os argumentos eram inseridos em um array e este era passado para o método. Para sermos sinceros, geralmente essas duas abordagens resultavam em soluções desajeitadas, e sabia-se que uma melhor abordagem era necessária.

A partir do JDK 5, essa necessidade foi atendida pela inclusão de um recurso que simplificou a criação de métodos que demandam um número variável de argumentos. Esse recurso se chama *varargs*, que é a abreviação de ‘variable-length arguments’. Um método que recebe um número variável de argumentos é chamado de método de aridade variável, ou simplesmente *método varargs*. A lista de parâmetros de um método varargs não é fixa, mas sim em variável tamanho. Portanto, um método varargs pode receber um número de argumentos variável.

Respostas:

1. Verdadeiro.
2. Verdadeiro.

## Aspectos básicos dos varargs

Uma lista de argumentos de tamanho variável é especificada por três pontos (...). Por exemplo, veja como criar um método chamado **vaTest()** que recebe um número de argumentos variável:

```
// vaTest() usa um vararg.
static void vaTest(int ... v) { ← Declara uma lista de argumentos de tamanho variável.
 System.out.println("Number of args: " + v.length);
 System.out.println("Contents: ");

 for(int i=0; i < v.length; i++)
 System.out.println(" arg " + i + ": " + v[i]);

 System.out.println();
}
```

Observe que **v** é declarado como mostrado aqui:

```
| int ... v
```

Essa sintaxe diz ao compilador que **vaTest()** pode ser chamado com zero ou mais argumentos. Além disso, faz **v** ser declarado implicitamente como um array de tipo **int[]**. Portanto, dentro de **vaTest()**, **v** é acessado com o uso da sintaxe comum dos arrays.

Este é um programa completo que demonstra **vaTest()**:

```
// Demonstra argumentos em quantidade variável.
class VarArgs {

 // vaTest() usa um vararg.
 static void vaTest(int ... v) {
 System.out.println("Number of args: " + v.length);
 System.out.println("Contents: ");

 for(int i=0; i < v.length; i++)
 System.out.println(" arg " + i + ": " + v[i]);

 System.out.println();
 }

 public static void main(String[] args)
 {

 // Observe como vaTest() pode ser chamado
 // com um número de argumentos variável.
 vaTest(10); // 1 argumento
 vaTest(1, 2, 3); // 3 argumentos
 vaTest(); // nenhum argumento
 }
}
```

A saída do programa é mostrada aqui:

```
Number of args: 1
Contents:
 arg 0: 10
```

```
Number of args: 3
Contents:
 arg 0: 1
 arg 1: 2
 arg 2: 3
```

```
Number of args: 0
Contents:
```

Há duas coisas importantes que devemos observar nesse programa. Em primeiro lugar, como explicado, dentro de `vaTest()`, `v` é tratado como um array. Isso ocorre porque `v` é *um array*. A sintaxe ... diz ao compilador que um número de argumentos variável será usado e que esses argumentos serão armazenados no array referenciado por `v`. Em segundo lugar, em `main()`, `vaTest()` é chamado com números de argumentos diferentes, inclusive sem argumentos. Os argumentos são inseridos automaticamente em um array e passados para `v`. No caso da ausência de argumentos, o tamanho do array é zero.

Um método pode ter parâmetros “comuns” junto com um parâmetro em quantidade variável. No entanto, o parâmetro de tamanho variável deve ser o último declarado pelo método. Por exemplo, a seguinte declaração de método é perfeitamente aceitável:

```
| int doit(int a, int b, double c, int ... vals) {
```

Nesse caso, os três primeiros argumentos usados em uma chamada a `doIt()` serão trazidos para os três primeiros parâmetros. Qualquer argumento restante será considerado pertencente a `vals`.

Aqui está uma versão retrabalhada do método `vaTest()` que recebe um argumento comum e um de tamanho variável:

```
// Usa varargs com argumentos padrão.
class VarArgs2 {

 // Aqui, msg é um parâmetro comum e v é um
 // parâmetro varargs.
 static void vaTest(String msg, int ... v) { ← Um parâmetro “comum” e
 System.out.println(msg + v.length); um vararg.
 System.out.println("Contents: ");

 for(int i=0; i < v.length; i++)
 System.out.println(" arg " + i + ": " + v[i]);

 System.out.println();
 }

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

```
{
 vaTest("One vararg: ", 10);
 vaTest("Three varargs: ", 1, 2, 3);
 vaTest("No varargs: ");
}
```

A saída do programa é esta:

```
One vararg: 1
Contents:
 arg 0: 10

Three varargs: 3
Contents:
 arg 0: 1
 arg 1: 2
 arg 2: 3

No varargs: 0
Contents:
```

Lembre-se de que o parâmetro varargs deve ser o último. Por exemplo, a declaração a seguir está incorreta:

```
| int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Erro!
```

Nesse caso, há uma tentativa de declarar um parâmetro comum após o parâmetro varargs, o que é inválido. Há mais uma restrição que devemos conhecer: só pode haver um parâmetro varargs. Por exemplo, a declaração seguinte também é inválida:

```
| int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Erro!
```

A tentativa de declarar o segundo parâmetro varargs é inválida.

## Verificação do progresso

1. Mostre como declarar um método chamado **sum()** que receba um número variável de argumentos **int**. (Use um tipo de retorno **int**.)
2. Dada esta declaração,

```
| void m(double ... x)
```

o parâmetro **x** é declarado implicitamente como um \_\_\_\_\_.

## Sobrecarregando métodos varargs

Podemos sobrecarregar um método que use um argumento de tamanho variável. Por exemplo, o programa a seguir sobrecarrega **vaTest()** três vezes:

**Respostas:**

1. `int sum(int ... n)`
2. `array double`

```
// Varargs e a sobrecarga.
class VarArgs3 {
 static void vaTest(int ... v) { ← Primeira versão de vaTest()
 System.out.println("vaTest(int ...): " +
 "Number of args: " + v.length);
 System.out.println("Contents: ");

 for(int i=0; i < v.length; i++)
 System.out.println(" arg " + i + ": " + v[i]);

 System.out.println();
 } ← Segunda versão de vaTest()

 static void vaTest(boolean ... v) { ←
 System.out.println("vaTest(boolean ...): " +
 "Number of args: " + v.length);
 System.out.println("Contents: ");

 for(int i=0; i < v.length; i++)
 System.out.println(" arg " + i + ": " + v[i]);

 System.out.println();
 } ← Terceira versão de vaTest()

 static void vaTest(String msg, int ... v) { ←
 System.out.println("vaTest(String, int ...): " +
 msg + v.length);
 System.out.println("Contents: ");

 for(int i=0; i < v.length; i++)
 System.out.println(" arg " + i + ": " + v[i]);

 System.out.println();
 }

 public static void main(String[] args)
 {
 vaTest(1, 2, 3);
 vaTest("Testing: ", 10, 20);
 vaTest(true, false, false);
 }
}
```

A saída produzida pelo programa é mostrada abaixo:

```
vaTest(int ...): Number of args: 3
Contents:
 arg 0: 1
 arg 1: 2
 arg 2: 3
```

```
vaTest(String, int ...): Testing: 2
Contents:
 arg 0: 10
 arg 1: 20

vaTest(boolean ...): Number of args: 3
Contents:
 arg 0: true
 arg 1: false
 arg 2: false
```

Esse programa ilustra as duas maneiras pelas quais um método varargs pode ser sobre-carregado. Em primeiro lugar, os tipos de seu parâmetro varargs podem variar. É esse o caso de `vaTest(int ...)` e `vaTest(boolean ...)`. Lembre-se, os três pontos fazem o parâmetro ser tratado como um array do tipo especificado. Portanto, da mesma forma que você pode também sobrecarregar métodos usando diferentes tipos de parâmetros de array, pode sobrecarregar métodos varargs usando diferentes tipos de varargs. Nesse caso, Java usa a diferença de tipo para determinar que método sobre-carregado será chamado.

A segunda maneira de sobre-carregar um método varargs é adicionar um ou mais parâmetros comuns. É isso que foi feito com `vaTest(String, int ...)`. Nesse caso, Java usa tanto a quantidade quanto o tipo dos argumentos para determinar que método chamar.

## Varargs e ambiguidade

Erros inesperados podem surgir na sobre-carga de um método que use um argumento de tamanho variável. Esses erros envolvem a ambiguidade, porque é possível criar uma chamada ambígua a um método varargs sobre-carregado. Por exemplo, considere o programa a seguir:

```
// Varargs, a sobre-carga e a ambiguidade.
//
// Este programa contém um erro e não será compilado!
class VarArgs4 {

 // Usa um parâmetro vararg int.
 static void vaTest(int ... v) { ← Um vararg int
 // ...
 }

 // Usa um parâmetro vararg booleano.
 static void vaTest(boolean ... v) { ← Um vararg booleano
 // ...
 }

 public static void main(String[] args)
 {
 vaTest(1, 2, 3); // OK
 vaTest(true, false, false); // OK

 vaTest(); // Erro: ambíguo! ← Ambíguo!
 }
}
```

Nesse programa, a sobrecarga de **vaTest()** está perfeitamente correta. No entanto, o programa não será compilado devido à chamada abaixo:

```
| vaTest(); // Erro: ambíguo!
```

Já que o parâmetro varargs pode estar vazio, essa chamada poderia ser convertida em uma chamada a **vaTest(int ...)** ou a **vaTest(boolean ...)**. As duas também são válidas. Logo, a chamada é inherentemente ambígua.

Aqui está outro exemplo de ambiguidade. As versões sobrecarregadas de **vaTest()** a seguir são inherentemente ambíguas, ainda que uma use um parâmetro comum:

```
| static void vaTest(int ... v) { // ...
| static void vaTest(int n, int ... v) { // ...
```

Embora as listas de parâmetros de **vaTest()** sejam diferentes, não há como o compilador resolver a chamada a seguir:

```
vaTest(1)
```

Elá representa uma chamada a **vaTest(int ...)**, com um argumento varargs, ou uma chamada a **vaTest(int, int ...)** sem argumentos varargs? Não há como o compilador responder a essa pergunta. Logo, a situação é ambígua.

Devido a erros de ambiguidade como os que acabamos de mostrar, às vezes você terá que desistir da sobrecarga e usar dois nomes de método diferentes. Em alguns casos, os erros de ambiguidade também expõem uma falha conceitual no código, que você pode remediar elaborando uma solução mais cuidadosa.

## EXERCÍCIOS

1. Dado o seguinte fragmento,

```
| class X {
| private int count;
```

o fragmento a seguir está correto?

```
| class Y {
| public static void main(String[] args) {
| X ob = new X();
|
| ob.count = 10;
| }
| }
```

2. Um modificador de acesso deve \_\_\_\_\_ a declaração de um membro.

3. Dada esta classe,

```
| class Test {
| int a;
| Test(int i) { a = i; }
| }
```

crie um método chamado **swap()** que troque o conteúdo dos objetos referenciados por duas referências de objeto **Test**.

4. O fragmento a seguir está correto?

```
class X {
 int meth(int a, int b) { ... }
 String meth(int a, int b) { ... }
```

5. Crie um método recursivo que exiba o conteúdo de um string de trás para frente.
6. Se todos os objetos de uma classe tiverem que compartilhar a mesma variável, como você deve declarar essa variável?
7. Por que você pode ter que usar um bloco **static**?
8. O que é uma classe interna?
9. Para que um membro só possa ser acessado por outros membros de sua classe, que modificador de acesso deve ser usado?
10. O nome de um método mais sua lista de parâmetros compõem a \_\_\_\_\_ do método.
11. Um argumento **int** é passado para um método com o uso da chamada por \_\_\_\_\_.
12. Crie um método varargs chamado **sum()** que some os valores **int** passados para ele. Faça-o retornar o resultado. Demonstre seu uso.
13. Um método varargs pode ser sobrecarregado?
14. Mostre um exemplo de um método varargs sobrecarregado que seja ambíguo.
15. Modifique o método **showBits()** da classe **BitOut** que vimos na seção Tente isto 5-3 do Capítulo 5 para que não exiba os bits e em vez disso retorne um **String** contendo os bits que seriam impressos. Modifique também o método **main()** da classe **ShowBitsDemo** para que teste o novo método **showBits()**.
16. Implemente um método **string2charArray()** que use um **String** como parâmetro. Ele cria e retorna um array **char** contendo os mesmos caracteres do string na mesma ordem.
17. Implemente um método **charArray2string()** que use um array **char** como parâmetro. Ele cria e retorna um **String** contendo os mesmos caracteres do array na mesma ordem.
18. Implemente um método **readString()** que use **System.in.read()** para ler uma linha de caracteres. Em seguida, ele combina os caracteres em um **String** que é retornado. O string retornado deve incluir o caractere de fim de linha ‘\n’.
19. O método **hasDuplicateValues()** mostrado abaixo deveria retornar **true** se o array tivesse algum inteiro repetido e retornar **false** se todos os inteiros fossem exclusivos. No entanto, ele não funciona corretamente. Explique por que está errado e então faça a correção.

```
boolean hasDuplicateValues(int [] data) {
 for(int i = 0; i < data.length; i++)
 for(int j = 0; j < data.length; j++)
 if(data[i] == data[j]) return true;
 return false;
}
```

20. Implemente um método **addAtEnd()** que use um array de inteiros **data** e um inteiro **x** como parâmetros. Ele cria um novo array cujo tamanho é uma unidade maior que o tamanho de **data**. Em seguida, copia todos os elementos de **data** para o novo array e, para concluir, adiciona o valor de **x** ao último elemento do array. Ele retorna o novo array.
21. Implemente um método **insert()** que use um array de inteiros **data**, um inteiro **x** e um inteiro **idx** como parâmetros. Ele cria um novo array cujo tamanho é uma unidade maior que o tamanho de **data**. Em seguida, copia **x** e todos os elementos de **data** para o novo array. O valor de **x** é inserido no novo array no índice **idx** e os valores de **data** são adicionados para preencher os elementos próximos de **x** na ordem em que estão em **data**. Ele retorna o novo array.
22. Implemente um método **remove()** que use um array de inteiros **data** e um inteiro **idx** como parâmetros. Ele cria um novo array cujo tamanho é uma unidade menor que o tamanho de **data**. Em seguida, copia todos os elementos de **data** para o novo array exceto o valor do índice **idx**. Ele retorna o novo array.
23. Suponhamos que você tivesse um método que usasse como parâmetros dois arrays de inteiros **data1** e **data2** com o mesmo tamanho e copiasse todos os dados de **data1** em **data2** para então apagar **data1** (isto é, ele configura todos os valores de **data1** com 0). O que aconteceria se alguém chamassem esse método e passasse o mesmo array como os dois argumentos em vez de passar dois arrays distintos?
24. Suponhamos que uma classe tivesse um método sobrecarregado chamado **add** com as duas implementações a seguir:

```
double add(int x, double y) { return x + y; }

double add(double x, int y) { return x + y + 1; }
```

O que seria retornado, caso algo seja, pelas chamadas de método a seguir?

- A. `add(3, 3.14)`
- B. `add(3.14, 3)`
- C. `add(3, 3)`
- D. `add(3.14, 3.14)`

25. O método **drawStars()** descrito na seção sobre recursão desenha um asterisco e então chama recursivamente a si próprio para desenhar os outros **n-1** asteriscos. Ele também poderia operar na ordem oposta? Isto é, poderia chamar a si mesmo recursivamente para desenhar **n-1** asteriscos e então desenhar o último asterisco?
26. O que aconteceria se o método **drawStars()** descrito na seção sobre recursão fosse chamado usando como argumento o inteiro **-1**? Modifique o método para que, se um inteiro negativo for passado como argumento, nada seja exibido.
27. Crie um método recursivo **countDown()** que use um inteiro **n** como parâmetro. Ele exibe regressivamente os inteiros de **n** a 0, um por linha, e então exibe “Blast off!”.

28. Crie um método recursivo **add1toN()** que use um inteiro **n** como parâmetro. Ele retorna a soma  $1 + 2 + 3 + \dots + n$ .
29. Abaixo temos o código de um método recursivo chamado **mystery**. O que é exibido quando **mystery(1, 2)** é chamado? Quantas chamadas recursivas foram feitas?

```
void mystery(int a, int b)
{
 if(a == 0 && b == 0)
 System.out.println(0);
 else if(a == 0) {
 System.out.println(b);
 mystery(a, b-1);
 }
 else {
 mystery(a-1, b);
 System.out.println(b);
 }
}
```

30. Implemente um método **equalArrays()** que use dois arrays de inteiros como parâmetros e retorne **true** se ambos tiverem o mesmo tamanho e valores iguais em índices correspondentes. Implemente-o de duas maneiras:
- iterativamente
  - recursivamente (*Dica:* crie uma função auxiliar com um parâmetro adicional.)
31. Crie um método **reverse()** que use um array de inteiros como parâmetro e inverta a ordem dos elementos do array. Implemente-o de duas maneiras:
- iterativamente
  - recursivamente (*Dica:* crie uma função auxiliar com um parâmetro adicional.)
32. Implemente um método **numTimes()** que use dois parâmetros: um array de inteiros chamado **data** e um inteiro chamado **x**. Ele retorna quantas vezes **x** aparece no array. Implemente-o de duas maneiras:
- iterativamente
  - recursivamente (*Dica:* crie uma função auxiliar com um parâmetro adicional.)
33. O código a seguir não será compilado. Explique o que está errado.
- ```
class Oops {
    int x = 3;

    static void changeX() { x = 4; }
}
```
34. Na seção Tente isto 6-3, um método de classificação rápida, **qsort()**, foi implementado e demonstrado. Ele chama um método recursivo **qs()**. Quantas vezes **qs()** será chamado se **qsort()** for chamado para classificar cada um dos arrays

a seguir? Para responder a pergunta, crie uma variável **counter** e adicione uma instrução que a incremente no começo do corpo de **qs()** para que ela registre quantas vezes **qs()** é chamado.

- A. `{'a', 'b', 'c', 'd', 'e', 'f', 'g'} // um array já classificado`
- B. `{'g', 'f', 'e', 'd', 'c', 'b', 'a'} // um array classificado na // ordem inversa`
- C. `{'a', 'c', 'e', 'g', 'i', 'k', 'm', 'o', 'q', 's', 'u', 'w', 'y', 'b', 'd', 'f', 'h', 'j', 'l', 'n', 'p', 'r', 't', 'v', 'x', 'z'}`

35. Na seção Tente isto 6-3, o método de classificação rápida recursivo **qs()** seleciona o valor do meio do array como comparando. Modifique o método para que selecione o primeiro elemento do array como comparando e então responda as mesmas perguntas do exercício anterior.
36. Se tanto uma classe interna quanto uma classe externa tiverem uma variável de instância chamada **x**, que variável estará sendo referenciada quando **x** for usada em um método da classe interna? Por quê?
37. Qual é a diferença entre os dois métodos a seguir? Mais precisamente, os corpos dos métodos são iguais, mas os parâmetros não. O que a versão varargs **addUp1** nos permite fazer diferentemente da versão de array **addUp2**?

```
int addUp1(int ... v) {
    int sum = 0;

    for(int x : v)
        sum += x;
    return sum;
}

int addUp2(int[] v) {
    int sum = 0;

    for(int x : v)
        sum += x;
    return sum;
}
```

38. Na seção Tente isto 6-2, foi implementado um novo construtor de **SimpleStack** que usa outra pilha como argumento e cria uma cópia dela. Aqui está o código:

```
// Constrói uma pilha a partir de outra.
SimpleStack(SimpleStack otherStack) {
    // o tamanho da nova pilha é igual ao de otherStack
    data = new char[otherStack.data.length];

    // configura tos com a mesma posição
    tos = otherStack.tos;
```

```
// copia o conteúdo
for(int i = 0; i < tos; i++)
    data[i] = otherStack.data[i];
}
```

O que há de errado com a implementação do construtor como descrito abaixo, que é muito mais simples?

```
// Constrói uma pilha a partir de outra.
SimpleStack(SimpleStack otherStack) {
    // configura data & tos com as variáveis data & tos de otherStack
    data = otherStack.data;
    tos = otherStack.tos;
}
```

Herança

PRINCIPAIS HABILIDADES E CONCEITOS

- Entender os aspectos básicos da herança
- Chamar construtores de superclasses
- Usar **super** para acessar membros da superclasse
- Criar uma hierarquia de classes com vários níveis
- Saber quando os construtores são chamados
- Entender as referências da superclasse a objetos da subclasse
- Sobrepor métodos
- Usar métodos sobrepostos para dar suporte ao polimorfismo
- Usar classes abstratas
- Usar **final**
- Conhecer a classe **Object**

Herança é um dos três princípios básicos da programação orientada a objetos, porque permite a criação de classificações hierárquicas. Usando herança, você pode criar uma classe geral que defina características comuns a um conjunto de itens relacionados. Essa classe poderá então ser herdada por outras classes mais específicas, cada uma adicionando suas características exclusivas.

No jargão Java, a classe que é herdada se chama *superclasse*. A classe que herda se chama *subclasse*. Portanto, uma subclasse é uma versão especializada da superclasse. Ela herda todas as variáveis e métodos definidos pela superclasse e adiciona seus próprios elementos exclusivos.

ASPECTOS BÁSICOS DE HERANÇA

Java dá suporte a herança permitindo que uma classe incorpore outra em sua declaração. Isso é feito com o uso da palavra-chave **extends**. Portanto, a subclasse traz acréscimos (estende) à superclasse.

Comecemos com um exemplo curto que ilustra vários dos recursos-chave de herança. O programa a seguir cria uma superclasse chamada **TwoDShape**, que armazena a largura e a altura de um objeto bidimensional, e uma subclasse chamada **Triangle**. Observe como a palavra-chave **extends** é usada para criar uma subclasse.

```
// Uma hierarquia de classe simples.

// Uma classe para objetos de duas dimensões.
class TwoDShape {
    double width;
    double height;

    void showDim() {
        System.out.println("Width and height are " +
                           width + " and " + height);
    }
}

// Uma subclasse de TwoDShape para triângulos.
class Triangle extends TwoDShape {
    String style;   ↑————— Triangle herda TwoDShape.

    double area() {
        return width * height / 2; ←————— Triangle pode referenciar os membros de
    }                                TwoDShape como se fossem seus.

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

class Shapes {
    public static void main(String[] args) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();

        t1.width = 4.0; ←————— Todos os membros de Triangle estão disponíveis
        t1.height = 4.0; para objetos Triangle, mesmo os herdados de
        t1.style = "filled"; TwoDShape.

        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = "outlined";

        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());

        System.out.println();
    }
}
```

```

        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}

```

A saída desse programa é mostrada abaixo:

```

Info for t1:
Triangle is filled
Width and height are 4.0 and 4.0
Area is 8.0

Info for t2:
Triangle is outlined
Width and height are 8.0 and 12.0
Area is 48.0

```

Aqui, **TwoDShape** define os atributos de uma forma bidimensional “genérica”, como um quadrado, retângulo ou triângulo. A classe **Triangle** cria um tipo específico de **TwoDShape**, nesse caso, um triângulo. Ela inclui tudo que pertence a **TwoDShape** e adiciona o campo **style**, o método **area()** e o método **showStyle()**. O estilo do triângulo é armazenado em **style**. Pode ser qualquer string que descreva o triângulo, como “cheio”, “contorno”, “transparente” ou até algo como “símbolo de aviso”, “isósceles” ou “arredondado”. O método **area()** calcula e retorna a área do triângulo e **showStyle()** exibe seu estilo.

Como **Triangle** inclui todos os membros de sua superclasse, **TwoDShape**, pode acessar **width** e **height** dentro de **area()**. Além disso, dentro de **main()**, os objetos **t1** e **t2** podem referenciar **width** e **height** diretamente, como se eles fizessem parte de **Triangle**. A Figura 7-1 esquematiza conceitualmente como **TwoDShape** é incorporada a **Triangle**.

Ainda que **TwoDShape** seja a superclasse de **Triangle**, ela também é uma classe autônoma totalmente independente. Ser a superclasse de uma subclasse não significa não poder ser usada separadamente. Por exemplo, o código a seguir é perfeitamente válido.

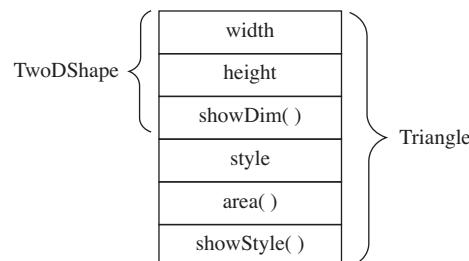


Figura 7-1 Representação conceitual da classe **Triangle**.

```
TwoDShape shape = new TwoDShape();  
  
shape.width = 10;  
shape.height = 20;  
  
shape.showDim();
```

É claro que um objeto de **TwoDShape** não conhece ou acessa nenhuma subclasse de **TwoDShape**.

A forma geral de uma declaração **class** que herda uma superclasse é mostrada aqui:

```
class nome-subclasse extends nome-superclasse {  
    // corpo da classe  
}
```

Você só pode especificar uma única superclasse para qualquer subclasse que criar. Java não dá suporte a herança de várias superclasses na mesma subclasse. No entanto, você pode criar uma hierarquia de herança em que uma subclasse passe a ser a superclasse de outra subclasse. Obviamente, nenhuma classe pode ser superclasse de si mesma.

Uma grande vantagem da herança é que, uma vez que você tenha criado uma superclasse que defina os atributos comuns a um conjunto de objetos, ela poderá ser usada para criar qualquer número de subclases mais específicas. Cada subclasse pode especificar com precisão sua própria classificação. Por exemplo, esta é outra subclasse de **TwoDShape** que encapsula retângulos:

```
// Uma subclasse de TwoDShape para retângulos.  
class Rectangle extends TwoDShape {  
    boolean isSquare() {  
        if(width == height) return true;  
        return false;  
    }  
  
    double area() {  
        return width * height;  
    }  
}
```

A classe **Rectangle** inclui **TwoDShape** e adiciona os métodos **isSquare()**, que determina se o retângulo é quadrado, e **area()**, que calcula a área de um retângulo. Observe que **Rectangle** não tem um campo **style** ou um método **showStyle()**. Embora **Triangle** adicione esses membros, isso não significa que **Rectangle** (ou qualquer outra subclasse de **TwoDShape**) também deva adicioná-los. Exceto por compartilhar a mesma superclasse, cada subclasse é independente. É claro que as subclasses podem fornecer membros semelhantes, com é o caso do método **area()**. Mesmo com uma implementação diferente, tanto **Triangle** quanto **Rectangle** fornecem esse método.

ACESSO A MEMBROS E HERANÇA

Como você aprendeu no Capítulo 6, com frequência a variável de instância de uma classe é declarada como **private** para não poder ser usada sem autorização ou adulterada. Herdar uma classe *não* invalida a restrição de acesso **private**. Logo, ainda que uma subclasse inclua todos os membros de sua superclasse, não poderá acessar

os membros declarados como **private**. Por exemplo, se, como mostrado aqui, **width** e **height** forem tornadas privadas em **TwoDShape**, **Triangle** não poderá acessá-las:

```
// Membros privados de uma superclasse não podem ser acessados por uma
// subclasse.

// Este exemplo não será compilado.

// Uma classe para objetos bidimensionais.
class TwoDShape {
    private double width; // agora esses
    private double height; // membros são privados

    void showDim() {
        System.out.println("Width and height are " +
                           width + " and " + height);
    }
}

// Uma subclasse de TwoDShape para triângulos.
class Triangle extends TwoDShape {
    String style;

    double area() {
        return width * height / 2; // Erro! não pode acessar
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}
```

Não pode acessar o membro **private** de uma superclasse.

A classe **Triangle** não será compilada, porque a referência a **width** e **height** dentro do método **area()** causa uma violação de acesso. Já que **width** e **height** foram declaradas como **private** em **TwoDShape**, só podem ser acessadas por outros membros de **TwoDShape**. As subclasses não podem acessá-las.

Lembre-se de que o membro de uma classe que foi declarado como **private** permanecerá sendo privado de sua classe. Ele não poderá ser acessado por nenhum código de fora da classe, inclusive subclasses.

À primeira vista, você pode achar que o fato de as subclasses não terem acesso aos membros privados das superclasses é uma restrição grave que impediria o uso de membros privados em muitas situações. No entanto, isso não é verdade. Como explicado no Capítulo 6, normalmente os programadores de Java usam métodos acessadores para dar acesso às variáveis de instância privadas de uma classe. Aqui está uma nova versão das classes **TwoDShape** e **Triangle** que usa métodos para acessar as variáveis de instância privadas **width** e **height**:

```
// Usa métodos acessadores para configurar e examinar membros privados.

// Uma classe para objetos bidimensionais.
class TwoDShape {
    private double width; // agora esses
```

```

private double height; // membros são privados
// Métodos acessadores para width e height.
double getWidth() { return width; }
double getHeight() { return height; } ← Métodos acessadores para
void setWidth(double w) { width = w; } width e height.
void setHeight(double h) { height = h; }

void showDim() {
    System.out.println("Width and height are " +
                       width + " and " + height);
}
}

// Uma subclasse de TwoDShape para triângulos.
class Triangle extends TwoDShape {
    String style;           Usa métodos acessadores fornecidos
                           pela superclasse.
    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

class Shapes2 {
    public static void main(String[] args) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();

        t1.setWidth(4.0);
        t1.setHeight(4.0);
        t1.style = "filled";

        t2.setWidth(8.0);
        t2.setHeight(12.0);
        t2.style = "outlined";

        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());

        System.out.println();

        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}

```

Pergunte ao especialista

P Quando devo tornar uma variável de instância privada?

R Não há regras fixas que sirvam a todas as situações, mas aqui estão dois princípios gerais: se uma variável de instância for usada apenas por métodos definidos dentro de sua classe, ela deve ser privada; se tiver que estar dentro de certos limites, deve ser privada e só estar disponível por intermédio de métodos acessadores. Dessa forma, você poderá impedir que valores inválidos sejam atribuídos. Além disso, o uso de métodos acessadores no acesso a dados permitirá que você altere mais facilmente a implementação da classe sem afetar seus usuários.

Verificação do progresso

1. Na criação de uma subclasse, que palavra-chave é usada para incluir uma superclasse?
2. Uma subclasse inclui os membros de sua superclasse?
3. Uma subclasse tem acesso aos membros privados de sua superclasse?

CONSTRUTORES E HERANÇA

Em uma hierarquia, é possível que tanto as superclasses quanto as subclasses tenham seus próprios construtores. Isso levanta uma questão importante: que construtor é responsável pela construção de um objeto da subclasse – o da superclasse, o da subclasse ou ambos? A resposta é esta: o construtor da superclasse constrói a parte do objeto referente à superclasse e o construtor da subclasse constrói a parte da subclasse. Faz sentido, porque a superclasse não conhece ou acessa elementos de uma subclasse. Portanto, sua construção deve ser separada. Os exemplos anteriores usaram os construtores padrão criados automaticamente por Java, logo, essa questão não foi um problema. Na prática, porém, a maioria das classes terá construtores explícitos. Agora você verá como tratar essa situação.

Quando só a subclasse define um construtor, o processo é simples: construimos apenas o objeto da subclasse. A parte do objeto referente à superclasse é construída automaticamente com o uso de seu construtor padrão. Por exemplo, aqui está uma versão retrabalhada de **Triangle** que define um construtor. Também torna **style** privada, já que agora ela é configurada pelo construtor.

```
// Adiciona um construtor a Triangle.  
// Uma classe para objetos bidimensionais.
```

Respostas:

1. **extends**
2. Sim.
3. Não.

```
class TwoDShape {
    private double width; // agora esses
    private double height; // membros são privados

    // Métodos acessadores para width e height.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Width and height are " +
                           width + " and " + height);
    }
}

// Uma subclasse de TwoDShape para triângulos.
class Triangle extends TwoDShape {
    private String style;

    // Construtor
    Triangle(String s, double w, double h) {
        setWidth(w); ← Inicializa a parte do objeto referente a TwoDShape.
        setHeight(h);

        style = s;
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

class Shapes3 {
    public static void main(String[] args) {
        Triangle t1 = new Triangle("filled", 4.0, 4.0);
        Triangle t2 = new Triangle("outlined", 8.0, 12.0);

        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());

        System.out.println();
        System.out.println("Info for t2: ");
    }
}
```

```
t2.showStyle();
t2.showDim();
System.out.println("Area is " + t2.area());
}
```

Nesse caso, o construtor de `Triangle` inicializa os membros herdados de `TwoDShape` junto com seu campo `style`.

Quando tanto a superclasse quanto a subclasse definem construtores, o processo é um pouco mais complicado, porque os dois construtores devem ser executados. Se isso ocorrer, você deve usar outra das palavras-chave Java, **super**, que tem duas formas gerais. A primeira chama um construtor da superclasse. A segunda é usada para acessar um membro da superclasse ocultado pelo membro de uma subclasse. Aqui, examinaremos seu primeiro uso.

USANDO super PARA CHAMAR CONSTRutores DA SUPERCLASSE

Uma subclasse pode chamar um construtor definido por sua superclasse usando a forma de **super** a seguir:

super(*lista-parâmetros*);

Lista-parâmetros especifica qualquer parâmetro requerido pelo construtor na superclasse. A primeira instrução executada dentro do construtor de uma subclasse deve ser sempre `super()`. Para ver como `super()` é usada, considere a versão de **TwoDSShape** do programa abaixo. Ela define um construtor que inicializa `width` e `height`.

```
// Subclasse de TwoDShape para triângulos.
class Triangle extends TwoDShape {
    private String style;

    Triangle(String s, double w, double h) {
        super(w, h); // chama construtor da superclasse
        ↑
        style = s;
    }                                Usa super() para executar o construtor de TwoDShape.

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

class Shapes4 {
    public static void main(String[] args) {
        Triangle t1 = new Triangle("filled", 4.0, 4.0);
        Triangle t2 = new Triangle("outlined", 8.0, 12.0);

        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());

        System.out.println();

        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}
```

Aqui, `Triangle()` chama `super()` com os parâmetros `w` e `h`. Isso faz o construtor `TwoDShape()` ser chamado e inicializar `width` e `height` com esses valores. A classe `Triangle` não os inicializa mais, só precisa inicializar o valor que é exclusivo dela: `style`. Assim, `TwoDShape` fica livre para construir sua parte da maneira que quiser. Além disso, pode adicionar funcionalidades sobre as quais as subclasses não tenham conhecimento, impedindo que o código existente seja danificado.

Toda forma de construtor definida pela superclasse pode ser chamada por `super()`. O construtor executado será o que tiver os argumentos correspondentes. Por exemplo, estas são versões expandidas tanto de `TwoDShape` quanto de `Triangle` que incluem construtores padrão e construtores que recebem um argumento:

```
// Adiciona mais construtores a TwoDShape.
class TwoDShape {
```

```

private double width;
private double height;

// Um construtor padrão.
TwoDShape() {
    width = height = 0.0;
}

// Construtor parametrizado.
TwoDShape(double w, double h) {
    width = w;
    height = h;
}

// Constrói o objeto com altura e largura iguais.
TwoDShape(double x) {
    width = height = x;
}

// Métodos acessadores para width e height.
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }

void showDim() {
    System.out.println("Width and height are " +
                       width + " and " + height);
}
}

// Subclasse de TwoDShape para triângulos.
class Triangle extends TwoDShape {
    private String style;

    // Construtor padrão.
    Triangle() {
        super(); ←
        style = "none";
    }

    // Construtor
    Triangle(String s, double w, double h) {
        super(w, h); // chama construtor da superclasse ←
        style = s;
    }

    // Construtor com um argumento.
    Triangle(double x) {
        super(x); // chama construtor da superclasse ←
        // O padrão de style é filled
        style = "filled";
    }
}

```

Usa **super()** para chamar as
várias formas do construtor de
TwoDShape.

```
}

double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    System.out.println("Triangle is " + style);
}
}

class Shapes5 {
    public static void main(String[] args) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle("outlined", 8.0, 12.0);
        Triangle t3 = new Triangle(4.0);

        t1 = t2;

        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());

        System.out.println();

        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());

        System.out.println();
    }
}
```

Veja a saída dessa versão:

```
Info for t1:
Triangle is outlined
Width and height are 8.0 and 12.0
Area is 48.0

Info for t2:
Triangle is outlined
```

```

Width and height are 8.0 and 12.0
Area is 48.0

Info for t3:
Triangle is filled
Width and height are 4.0 and 4.0
Area is 8.0

```

Revisemos os conceitos-chave de **super()**. Quando uma subclasse chama **super()**, está chamando o construtor de sua superclasse imediata. Portanto, **super()** sempre referencia a superclasse imediatamente acima da classe chamadora, o que é verdade mesmo em uma hierarquia de vários níveis. Além disso, **super()** deve ser sempre a primeira instrução executada dentro de um construtor de subclasse.

Verificação do progresso

1. Como uma subclasse executa o construtor de sua superclasse?
2. Os parâmetros podem ser passados via **super()**?
3. Uma chamada a **super()** pode estar em qualquer local do construtor de uma subclasse?

USANDO **super** PARA ACESSAR MEMBROS DA SUPERCLASSE

Há uma segunda forma de **super** que age um pouco como **this**, exceto por referenciar sempre a superclasse da subclasse em que é usada. Essa aplicação tem a forma geral a seguir:

`super.membro`

Aqui, *membro* pode ser um método ou uma variável de instância. Essa forma de **super** é mais aplicável a situações em que os nomes dos membros de uma subclasse ocultam membros com o mesmo nome na superclasse. Considere a seguinte hierarquia de classes simples:

```

// Usando super para resolver o problema de ocultação de nomes.
class A {
    int i;
}

// Cria uma subclasse estendendo a classe A.
class B extends A {
    int i; // essa variável i oculta a variável i de A

    B(int a, int b) {

```

Respostas:

1. Chama **super()**.
2. Sim.
3. Não, deve ser a primeira instrução executada.

```

super.i = a; // i de A ← Aqui, super.i referencia a variável i de A.
i = b; // i in B
}

void show() {
    System.out.println("i in superclass: " + super.i);
    System.out.println("i in subclass: " + i);
}
}

class UseSuper {
    public static void main(String[] args) {
        B subOb = new B(1, 2);

        subOb.show();
    }
}

```

O programa exibe o seguinte:

```
i in superclass: 1
i in subclass: 2
```

Embora a variável de instância **i** de **B** oculte a variável **i** de **A**, **super** permite o acesso à variável **i** definida na superclasse. Para chamar métodos ocultos por uma subclasse, **super** também pode ser usada.

TENTE ISTO 7-1 Estendendo a classe Vehicle

`TruckDemo.java`

Para ilustrar o poder de herança, estenderemos a classe **Vehicle** desenvolvida no Capítulo 4. Como você deve lembrar, **Vehicle** encapsula informações sobre veículos, inclusive o número de passageiros que eles podem levar, sua capacidade de armazenamento de combustível e sua taxa de consumo de combustível. Podemos usar a classe **Vehicle** como ponto de partida a partir do qual classes mais especializadas serão desenvolvidas. Por exemplo, um caminhão é um tipo de veículo. Um atributo importante de um caminhão é sua capacidade de transportar carga. Logo, para criar uma classe **Truck**, podemos estender **Vehicle**, adicionando uma variável de instância que armazene a capacidade de transporte de carga. Este projeto mostra como. No processo, as variáveis de instância de **Vehicle** serão tornadas **private** e métodos acessadores serão fornecidos para a verificação e a configuração de seus valores.

PASSO A PASSO

1. Crie um arquivo chamado **TruckDemo.java** e copie nele a última implementação de **Vehicle** do Capítulo 4.

2 Crie a classe **Truck** como mostrado abaixo:

```
// Estende Vehicle para criar a especialização Truck.
class Truck extends Vehicle {
    private int cargoCap; // capacidade de transporte de carga em
                          // libras

    // Construtor para Truck.
    Truck(int p, int f, int m, int c) {
        /* Inicializa os membros de Vehicle usando
         o construtor da classe. */
        super(p, f, m);

        cargoCap = c;
    }

    // Métodos acessadores para cargoCap.
    int getCargo() { return cargoCap; }
    void putCargo(int c) { cargoCap = c; }
}
```

Aqui, **Truck** herda **Vehicle**, adicionando **cargoCap**, **getCargo()** e **putCargo()**. Portanto, **Truck** inclui todos os atributos gerais dos veículos definidos por **Vehicle** e só precisa adicionar os itens que são exclusivos de sua própria classe.

3. Agora, torne as variáveis de instância de **Vehicle** privadas, como mostrado a seguir:

```
private int passengers; // número de passageiros
private int fuelCap;   // capacidade de armazenamento de
                      // combustível em galões
private int mpg;       // consumo de combustível em milhas por galão
```

4. Já que agora as variáveis de instância de **Vehicle** são privadas, você precisará dos métodos acessadores a seguir para configurar ou obter seus valores.

```
int getPassengers() { return passengers; }
void setPassengers(int p) { passengers = p; }

int getFuelCap() { return fuelCap; }
void setFuelCap(int f) { fuelCap = f; }

int getMpg() { return mpg; }
void setMpg(int m) { mpg = m; }
```

5. Aqui está um programa inteiro que demonstra a classe **Truck** e as alterações em **Vehicle**.

```
// Tente isto 7-1
//
// Constrói uma subclasse de Vehicle para caminhões.
class Vehicle {
    private int passengers; // número de passageiros
    private int fuelCap;   // capacidade de armazenamento de
                          // combustível em galões
```

```
private int mpg;           // consumo de combustível em milhas por
                         // galão

// Este é um construtor para Vehicle.
Vehicle(int p, int f, int m) {
    passengers = p;
    fuelCap = f;
    mpg = m;
}

// Retorna a autonomia.
int range() {
    return mpg * fuelCap;
}

// Calcula o combustível necessário para cobrir uma determinada
// distância.
double fuelNeeded(int miles) {
    return (double) miles / mpg;
}

// Métodos de acesso de variáveis de instância.
int getPassengers() { return passengers; }
void setPassengers(int p) { passengers = p; }

int getFuelCap() { return fuelCap; }
void setFuelCap(int f) { fuelCap = f; }

int getMpg() { return mpg; }
void setMpg(int m) { mpg = m; }
}

// Estende Vehicle para criar a especialização Truck.
class Truck extends Vehicle {
    private int cargoCap; // capacidade de carga em libras

    // Este é um construtor para Truck.
    Truck(int p, int f, int m, int c) {
        /* Inicializa membros de Vehicle usando o
           construtor de Vehicle. */
        super(p, f, m);

        cargoCap = c;
    }

    // Métodos acessadores para cargoCap.
    int getCargo() { return cargoCap; }
    void putCargo(int c) { cargoCap = c; }
}
```

```

class TruckDemo {
    public static void main(String[] args) {

        // constrói alguns caminhões
        Truck semi = new Truck(2, 200, 7, 44000);
        Truck pickup = new Truck(3, 28, 15, 2000);
        double gallons;
        int dist = 252;

        gallons = semi.fuelNeeded(dist);

        System.out.println("Semi can carry " + semi.getCargo() +
                           " pounds.");
        System.out.println("To go " + dist + " miles semi needs " +
                           gallons + " gallons of fuel.\n");

        gallons = pickup.fuelNeeded(dist);

        System.out.println("Pickup can carry " + pickup.getCargo() +
                           " pounds.");
        System.out.println("To go " + dist + " miles pickup needs " +
                           gallons + " gallons of fuel.");
    }
}

```

6. A saída desse programa é mostrada abaixo:

```

Semi can carry 44000 pounds.
To go 252 miles semi needs 36.0 gallons of fuel.

Pickup can carry 2000 pounds.
To go 252 miles pickup needs 16.8 gallons of fuel.

```

7. Muitos outros tipos de classes podem ser derivados de **Vehicle**. Por exemplo, o esboço a seguir cria uma classe off-road que armazena a distância entre o veículo e o solo.

```

// Cria uma classe de veículo off-road
class OffRoad extends Vehicle {
    private int groundClearance; // distância do solo em polegadas

    // ...
}

```

O ponto-chave é que, quando você tiver criado uma superclasse que defina os aspectos gerais de um objeto, ela poderá ser herdada para formar classes especializadas. Cada subclasse adicionará apenas seus próprios atributos exclusivos. Essa é a essência da herança.

CRIANDO UMA HIERARQUIA DE VÁRIOS NÍVEIS

Até agora, usamos hierarquias de classes simples compostas apenas por uma superclasse e uma subclasse. No entanto, podemos construir hierarquias contendo quantas

camadas de herança quisermos. Como mencionado, é perfeitamente aceitável usar uma subclasse como superclasse de outra subclasse. Por exemplo, dadas três classes chamadas **A**, **B** e **C**, **C** pode ser subclasse de **B**, que é subclasse de **A**. Quando ocorre esse tipo de situação, cada subclasse herda todas as características encontradas em todas as suas superclasses. Nesse caso, **C** herda todos os aspectos de **B** e **A**.

Para ver como uma hierarquia de vários níveis pode ser útil, considere o programa a seguir. Nele, a subclasse **Triangle** é usada como superclasse para criar a subclasse chamada **ColorTriangle**. **ColorTriangle** herda todas as características de **Triangle** e **TwoDShape** e adiciona um campo chamado **color**, que contém a cor do triângulo.

```
// Hierarquia de vários níveis.
class TwoDShape {
    private double width;
    private double height;

    // Construtor padrão.
    TwoDShape() {
        width = height = 0.0;
    }

    // Construtor parametrizado.
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    // Constrói objeto com largura e altura iguais.
    TwoDShape(double x) {
        width = height = x;
    }

    // Métodos acessadores para width e height.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Width and height are " +
                           width + " and " + height);
    }
}

// Estende TwoDShape.
class Triangle extends TwoDShape {
    private String style;

    // Construtor padrão.
    Triangle() {
```

```

        super();
        style = "none";
    }

Triangle(String s, double w, double h) {
    super(w, h); // chama construtor da superclasse

    style = s;
}

// Construtor com um argumento.
Triangle(double x) {
    super(x); // chama construtor da superclasse

    // padrão de style é filled
    style = "filled";
}

double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    System.out.println("Triangle is " + style);
}
}

// Estende Triangle.
class ColorTriangle extends Triangle {
    private String color; ↑
    ColorTriangle(String c, String s,
                  double w, double h) {
        super(s, w, h);

        color = c;
    }

    String getColor() { return color; }

    void showColor() {
        System.out.println("Color is " + color);
    }
}

class Shapes6 {
    public static void main(String[] args) {
        ColorTriangle t1 =
            new ColorTriangle("Blue", "outlined", 8.0, 12.0);
        ColorTriangle t2 =
            new ColorTriangle("Red", "filled", 2.0, 2.0);
    }
}

```

ColorTriangle herda **Triangle**, que é descendente de **TwoDShape**, portanto, **ColorTriangle** inclui todos os membros de **Triangle** e **TwoDShape**.

```
System.out.println("Info for t1: ");
t1.showStyle();
t1.showDim();
t1.showColor();
System.out.println("Area is " + t1.area());

System.out.println();

System.out.println("Info for t2: ");
t2.showStyle(); ← Um objeto ColorTriangle pode chamar métodos
t2.showDim(); definidos por ele próprio e suas superclasses.
t2.showColor();
System.out.println("Area is " + t2.area());
}
}
```

A saída desse programa é mostrada aqui:

```
Info for t1:
Triangle is outlined
Width and height are 8.0 and 12.0
Color is Blue
Area is 48.0

Info for t2:
Triangle is filled
Width and height are 2.0 and 2.0
Color is Red
Area is 2.0
```

Devido à herança, **ColorTriangle** pode fazer uso das classes **Triangle** e **TwoDShape** definidas anteriormente, adicionando apenas as informações extras de seu uso específico. Isso é parte do valor da herança: ela permite a reutilização de código.

O exemplo ilustra outro ponto importante: **super()** sempre referencia o construtor da superclasse mais próxima. A instrução **super()** de **ColorTriangle** chama o construtor de **Triangle**. A instrução **super()** de **Triangle** chama o construtor de **TwoDShape**. Em uma hierarquia de classes, se o construtor de uma superclasse precisar de parâmetros, todas as subclasses devem passá-los “para cima na hierarquia”. Isso é verdade quer a subclasse precise de parâmetros ou não.

QUANDO OS CONSTRUTORES SÃO EXECUTADOS?

Na discussão anterior sobre herança e hierarquias de classes, uma pergunta importante pode ter lhe ocorrido: quando o objeto de uma subclasse é criado, o construtor de quem é executado primeiro, o da subclasse ou o definido pela superclasse? Por exemplo, dada uma subclasse chamada **B** e uma superclasse chamada **A**, o construtor de **A** é executado antes do de **B** ou o contrário? A resposta é que em uma hierarquia de classes, os construtores concluem sua execução em ordem de derivação, da superclasse para a subclasse. Além disso, já que **super()** deve ser a primeira instrução

executada no construtor de uma subclasse, essa ordem é a mesma independentemente de **super()** ser ou não usada. Se **super()** não for usada, o construtor padrão (sem parâmetros) de cada superclasse será executado. O programa a seguir ilustra quando os construtores são executados:

```
// Demonstra quando os construtores são executados.

// Cria uma superclasse.
class A {
    A() {
        System.out.println("Constructing A.");
    }
}

// Cria uma subclasse estendendo a classe A.
class B extends A {
    B() {
        System.out.println("Constructing B.");
    }
}

// Cria outra subclasse estendendo B.
class C extends B {
    C() {
        System.out.println("Constructing C.");
    }
}

class OrderOfConstruction {
    public static void main(String[] args) {
        C c = new C(); ←———— Constrói um objeto C.
    }
}
```

A saída desse programa é mostrada aqui:

```
Constructing A.
Constructing B.
Constructing C.
```

Como você pode ver, os construtores são chamados em ordem de derivação.

Se pensarmos bem, faz sentido os construtores serem executados em ordem de derivação. Já que uma superclasse não tem conhecimento das subclasses, qualquer inicialização que ela precisar executar será separada e possivelmente pré-requisito de uma inicialização executada pela subclasse. Logo, ela deve ser executada antes.

REFERÊNCIAS DA SUPERCLASSE E OBJETOS DA SUBCLASSE

Como você sabe, Java é uma linguagem fortemente tipada. Além das conversões padrão e das promoções automáticas aplicadas aos seus tipos primitivos, a compatibilidade de tipos é imposta rigorosamente. Logo, normalmente uma variável de referê-

cia de um tipo de classe não pode referenciar um objeto de outro tipo de classe. Por exemplo, considere o programa abaixo:

```
// Este código não será compilado.
class X {
    int a;

    X(int i) { a = i; }

}

class Y {
    int a;

    Y(int i) { a = i; }

}

class IncompatibleRef {
    public static void main(String[] args) {
        X x = new X(10);
        X x2;
        Y y = new Y(5);

        x2 = x; // Correto, as duas são do mesmo tipo

        x2 = y; // Erro, não são do mesmo tipo
    }
}
```

Aqui, ainda que a classe **X** e a classe **Y** sejam fisicamente iguais, não é possível atribuir a uma variável **X** a referência a um objeto **Y**, porque elas têm tipos diferentes. Em geral, uma variável de referência de objeto só pode referenciar objetos de seu tipo.

No entanto, há uma exceção importante à imposição rigorosa de tipos em Java. A variável de referência de uma superclasse pode receber a referência a um objeto de qualquer subclasse derivada dessa superclasse. Em outras palavras, uma referência da superclasse pode referenciar um objeto da subclasse. Veja um exemplo:

```
// Uma referência de superclasse pode referenciar um objeto da subclasse.
class X {
    int a;

    X(int i) { a = i; }

}

class Y extends X {
    int b;

    Y(int i, int j) {
        super(j);
```

```

        b = i;
    }
}

class SupSubRef {
    public static void main(String[] args) {
        X x = new X(10);
        X x2;
        Y y = new Y(5, 6);

        x2 = x; // Correto, as duas são do mesmo tipo
        System.out.println("x2.a: " + x2.a); Correto porque Y é subclasse de
                                         X, logo, x2 pode referenciar y.

        ↓
        x2 = y; // ainda está correto porque Y é derivada de X
        System.out.println("x2.a: " + x2.a);

        // Referências de X só conhecem membros de X
        x2.a = 19; // OK
//      x2.b = 27; // Erro, X não tem um membro b
    }
}

```

Aqui, Y é derivada de X, logo, é permitido que x2 receba uma referência a um objeto Y.

É importante entender que é o tipo da variável de referência – e não o tipo do objeto que ela referencia – que determina os membros que podem ser acessados. Isto é, quando uma referência a um objeto da subclasse for atribuída a uma variável de referência da superclasse, você só terá acesso às partes do objeto definidas pela superclasse. É por isso que x2 não pode acessar b mesmo quando referencia um objeto Y. Se você pensar bem, faz sentido, porque a superclasse não tem conhecimento do que uma subclasse adiciona a ela. Por essa razão, a última linha de código do programa foi desativada por um comentário.

Embora a discussão anterior possa parecer um pouco etérea, ela tem algumas aplicações práticas importantes. Uma delas será descrita agora. A outra discutiremos posteriormente neste capítulo, quando a sobreposição de métodos for abordada.

Um local importante em que referências de subclasse são atribuídas a variáveis da superclasse é quando os construtores são chamados em uma hierarquia de classes. Como você sabe, é comum uma classe definir um construtor que recebe um objeto da classe como parâmetro. Isso permite que a classe construa uma cópia de um objeto. As subclasses de uma classe como essa podem se beneficiar desse recurso. Por exemplo, considere as versões a seguir de **TwoDShape** e **Triangle**. As duas adicionam construtores que recebem um objeto como parâmetro.

```

class TwoDShape {
    private double width;
    private double height;

    // Um construtor padrão.
    TwoDShape() {
        width = height = 0.0;
    }
}

```

```
// Construtor parametrizado.
TwoDShape(double w, double h) {
    width = w;
    height = h;
}

// Constrói um objeto com largura e altura iguais.
TwoDShape(double x) {
    width = height = x;
}

// Constrói um objeto a partir de outro.
TwoDShape(TwoDShape ob) { ← Constrói um objeto a partir de outro.
    width = ob.width;
    height = ob.height;
}

// Métodos acessadores para width e height.
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }

void showDim() {
    System.out.println("Width and height are " +
                       width + " and " + height);
}
}

// Subclasse de TwoDShape para triângulos.
class Triangle extends TwoDShape {
    private String style;

    // Construtor padrão.
    Triangle() {
        super();
        style = "none";
    }

    // Construtor de Triangle.
    Triangle(String s, double w, double h) {
        super(w, h); // chama construtor da superclasse

        style = s;
    }

    // Construtor com um argumento.
    Triangle(double x) {
        super(x); // chama construtor da superclasse
```

```

    // padrão de style é filled
    style = "filled";
}
// Constrói um objeto a partir de outro.
Triangle(Triangle ob) {
    super(ob); // passa o objeto para o construtor de TwoDShape
    style = ob.style; ↑———— Passa uma referência Triangle
}                                para o construtor de TwoDShape.

double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    System.out.println("Triangle is " + style);
}
}

class Shapes7 {
    public static void main(String[] args) {
        Triangle t1 =
            new Triangle("outlined", 8.0, 12.0);

        // faz uma cópia de t1
        Triangle t2 = new Triangle(t1);

        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());

        System.out.println();

        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}
}

```

Nesse programa, **t2** é construída a partir de **t1** e, portanto, é idêntica. A saída é mostrada aqui:

```

Info for t1:
Triangle is outlined
Width and height are 8.0 and 12.0
Area is 48.0

Info for t2:
Triangle is outlined
Width and height are 8.0 and 12.0
Area is 48.0

```

Preste atenção neste construtor de **Triangle**:

```
// Constrói um objeto a partir de outro.
Triangle(Triangle ob) {
    super(ob); // passa o objeto para o construtor de TwoDShape
    style = ob.style;
}
```

Ele recebe um objeto de tipo **Triangle** e o passa (por intermédio de **super**) para este construtor de **TwoDShape**:

```
// Constrói um objeto a partir de outro.
TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
}
```

O ponto-chave é que **TwoDShape()** está esperando um objeto **TwoDShape**. No entanto, **Triangle()** passa para ele um objeto **Triangle**. Isso funciona porque, como explicado, uma referência da superclasse pode referenciar um objeto da subclasse. Logo, é perfeitamente aceitável passar para **TwoDShape()** a referência a um objeto de uma classe derivada de **TwoDShape**. Já que o construtor **TwoDShape()** só está inicializando as partes do objeto da subclasse que são membros de **TwoDShape**, não importa se o objeto contém outros membros adicionados por classes derivadas.

Verificação do progresso

1. Uma subclasse pode ser usada como superclasse de outra subclasse?
2. Em uma hierarquia de classes, em que ordem os construtores são executados?
3. Dado que **Jet** estende **Airplane**, uma referência **Airplane** pode apontar para um objeto **Jet**?

SOBREPOSIÇÃO DE MÉTODOS

Em uma hierarquia de classes, quando um método de uma subclasse tem o mesmo tipo de retorno e assinatura de um método de sua superclasse, diz-se que o método da subclasse *sobrepõe* o método da superclasse. Quando um método sobreposto é chamado de dentro de uma subclasse, a referência é sempre à versão definida pela

Respostas:

1. Sim.
2. Os construtores são executados em ordem de derivação.
3. Sim. Em todos os casos, uma referência da superclasse pode apontar para um objeto da subclasse, mas não o contrário.

subclasse. A versão do método definida pela superclasse será ocultada. Considere o seguinte:

```
// Sobreposição de métodos.
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }

    // exibe i e j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // exibe k - esta versão sobrepuja show() em A
    void show() { ←———— Este método show() de B sobrepuja o definido por A.
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String[] args) {
        B subOb = new B(1, 2, 3);

        subOb.show(); // Essa instrução chama show() de B
    }
}
```

A saída produzida por esse programa é mostrada aqui:

```
|k: 3
```

Quando **show()** é chamado em um objeto de tipo **B**, a versão definida dentro de **B** é usada. Isto é, a versão de **show()** de **B** sobrepuja a versão declarada em **A**.

Se quiser acessar a versão de um método sobreposto definida pela superclasse, você pode fazer isso usando **super**. Por exemplo, nessa versão de **B**, a versão de **show()** da superclasse é chamada dentro da versão da subclasse. Isso permite que todas as variáveis de instância sejam exibidas.

```

class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    void show() {↓
        super.show(); // essa instrução chama o método show() de A
        System.out.println("k: " + k);
    }
}

```

Usa **super** para chamar a versão de **show()** definida pela superclasse A.

Se você usar essa versão de **show()** no programa anterior, verá a saída a seguir:

```
i and j: 1 2
k: 3
```

Aqui, **super.show()** chama a versão de **show()** da superclasse.

A sobreposição de métodos só ocorre quando as assinaturas dos dois métodos são idênticas. Se não forem, os dois métodos serão apenas sobrecarregados. Por exemplo, considere uma versão modificada do exemplo anterior:

```

/* Métodos com assinaturas diferentes são
   sobrecarregados e não sobrepostos. */
class A {
    int i, j;

    A(int a, int b) {
        i = a;
        j = b;
    }

    // exibe i e j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

// Cria uma subclasse estendendo a classe A.
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // sobrecarrega show()
    void show(String msg) {←

```

Já que as assinaturas diferem,
o método **show()** apenas
sobrecreve o da superclasse A.

```

        System.out.println(msg + k);
    }
}

class Overload {
    public static void main(String[] args) {
        B subOb = new B(1, 2, 3);

        subOb.show("This is k: "); // chama show() em B
        subOb.show(); // chama show() em A
    }
}

```

A saída produzida pelo programa é mostrada abaixo:

```

| This is k: 3
| i and j: 1 2

```

A versão de **show()** definida por **B** recebe um parâmetro tipo string. Isso torna sua assinatura diferente da existente em **A**, que não recebe parâmetros. Logo, não ocorre sobreposição (ou ocultação de nomes).

MÉTODOS SOBREPOSTOS DÃO SUPORTE AO POLIMORFISMO

Embora os exemplos da seção anterior demonstrem a mecânica da sobreposição de métodos, eles não mostram seu poder. Na verdade, se não houvesse nada mais na sobreposição de métodos além de uma convenção de espaço de nome, então ela seria, no máximo, uma curiosidade interessante, mas de pouco valor real. No entanto, não é esse o caso. A sobreposição de métodos forma a base de um dos conceitos mais poderosos em Java: *o despacho dinâmico de métodos*. Despacho dinâmico de métodos é o mecanismo pelo qual a chamada a um método sobreposto é resolvida no tempo de execução em vez de no tempo de compilação. O despacho dinâmico é importante porque é assim que Java implementa o polimorfismo no tempo de execução.

Comecemos reafirmando um princípio importante: uma variável de referência da superclasse pode referenciar um objeto da subclasse. Java usa esse fato para resolver chamadas a métodos sobrepostos no tempo de execução. Veja como: quando um método sobreposto é chamado por uma referência da superclasse, Java determina a versão desse método que será executada com base no tipo do objeto sendo referenciado no momento em que a chamada ocorre. Portanto, essa escolha é feita no tempo de execução. Quando diferentes tipos de objetos são referenciados, versões distintas de um método sobreposto são chamadas. Em outras palavras, *é o tipo do objeto referenciado* (e não o tipo da variável de referência) que determina a versão de um método sobreposto que será executada. Logo, se uma superclasse tiver um método sobreposto por uma subclasse, quando diferentes tipos de objetos da subclasse forem referenciados por uma variável de referência da superclasse, versões distintas do método serão executadas.

Aqui está um exemplo que ilustra o despacho dinâmico de métodos:

```
// Demonstra o despacho dinâmico de métodos.

class Sup {
    void who() {
        System.out.println("who() in Sup");
    }
}

class Sub1 extends Sup {
    void who() {
        System.out.println("who() in Sub1");
    }
}

class Sub2 extends Sup {
    void who() {
        System.out.println("who() in Sub2");
    }
}

class DynDispDemo {
    public static void main(String[] args) {
        Sup superOb = new Sup();
        Sub1 subOb1 = new Sub1();
        Sub2 subOb2 = new Sub2();

        Sup supRef;

        supRef = superOb;
        supRef.who();           ← Em cada caso,
                               a versão de
                               who() a ser
                               chamada é
                               determinada
                               no tempo de
                               execução pelo
                               tipo de objeto
                               referenciado.

        supRef = subOb1;
        supRef.who();           ←
        supRef = subOb2;
        supRef.who();           ←
    }
}
```

A saída do programa é mostrada aqui:

```
| who() in Sup
| who() in Sub1
| who() in Sub2
```

Esse programa cria uma superclasse chamada **Sup** com duas subclasses chamadas **Sub1** e **Sub2**. **Sup** declara um método chamado **who()** e as subclasses o sobrepõem. Dentro do método **main()**, objetos de tipo **Sup**, **Sub1** e **Sub2** são declarados. Além disso, uma referência de tipo **Sup**, chamada **supRef**, é declarada. O programa atribui

então uma referência de cada tipo de objeto a **supRef** e usa essa referência para chamar **who()**. Como a saída mostra, a versão de **who()** executada é determinada pelo tipo de objeto referenciado no momento da chamada, e não pelo tipo de classe de **supRef**.

Pergunte ao especialista

P Há alguma outra linguagem de programação orientada a objetos que dê suporte à sobreposição de métodos? Ou ela só ocorre em Java?

R A sobreposição de métodos não ocorre só em Java. Por exemplo, tanto C++ quanto C# dão suporte à sobreposição. Em C++, ela ocorre com o uso de funções virtuais. Em C#, com o uso de métodos virtuais. No entanto, em ambas, o efeito é semelhante ao encontrado em Java.

POR QUE SOBREPOR MÉTODOS?

Como mencionado anteriormente, os métodos sobrepostos permitem que Java dê suporte ao polimorfismo no tempo de execução. O polimorfismo é essencial para a programação orientada a objetos por uma razão: permite que uma classe geral especifique métodos que serão comuns a todos os seus derivados, permitindo também que as subclasses definam a implementação específica de alguns desses métodos ou de todos eles. Os métodos sobrepostos são outra maneira de Java implementar o aspecto “uma interface, vários métodos” do polimorfismo. Parte do segredo para a aplicação bem-sucedida do polimorfismo é entender que as superclasses e subclasses formam uma hierarquia que se move da menor para a maior especialização. Quando usada corretamente, a superclasse fornece todos os elementos que uma subclasse pode usar diretamente. Também especifica os métodos que a classe derivada deve implementar por conta própria. Isso dá à subclasse flexibilidade para definir seus próprios métodos, sem deixar de impor a consistência da interface. Logo, combinando herança com os métodos sobrepostos, uma superclasse pode definir a forma geral dos métodos que serão usados por todas as suas subclasses.

Aplicando a sobreposição de métodos a **TwoDShape**

Para demonstrar melhor o poder da sobreposição de métodos, ela será aplicada à classe **TwoDShape**. Nos exemplos anteriores, cada classe derivada de **TwoDShape** define um método chamado **area()**. Ou seja, pode ser mais adequado tornar **area()** parte da classe **TwoDShape** e permitir que cada subclasse o sobreponha, definindo como a área é calculada para o tipo de forma que a classe encapsula. O programa a seguir age desse modo. Por conveniência, ele também adiciona um campo de nome a **TwoDShape**. (Isso facilita a criação de programas de demonstração.)

```
// Usa o despacho dinâmico de métodos.
class TwoDShape {
    private double width;
    private double height;
    private String name;

    // Construtor padrão.
```

```
TwoDShape() {
    width = height = 0.0;
    name = "none";
}

// Construtor parametrizado.
TwoDShape(double w, double h, String n) {
    width = w;
    height = h;
    name = n;
}

// Constrói objeto com largura e altura iguais.
TwoDShape(double x, String n) {
    width = height = x;
    name = n;
}

// Constrói um objeto a partir de outro.
TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
}

// Métodos acessadores para width e height.
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }

String getName() { return name; }

void showDim() {
    System.out.println("Width and height are " +
                       width + " and " + height);
}

double area() { ←
    System.out.println("area() must be overridden");
    return 0.0;
}

// Uma subclasse de TwoDShape para triângulos.
class Triangle extends TwoDShape {
    private String style;

    // Construtor padrão.
    Triangle() {
        super();
    }
}
```

Método **area()** definido por **TwoDShape**.

```
    style = "none";
}

// Construtor para Triangle.
Triangle(String s, double w, double h) {
    super(w, h, "triangle");

    style = s;
}

// Construtor com um argumento.
Triangle(double x) {
    super(x, "triangle"); // chama construtor da superclasse

    // padrão de style é filled
    style = "filled";
}

// Constrói um objeto a partir de outro.
Triangle(Triangle ob) {
    super(ob); // passa objeto para o construtor de TwoDShape
    style = ob.style;
}

// Sobrepõe area() para Triangle.
double area() { ← Sobrepondo area() para Triangle.
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    System.out.println("Triangle is " + style);
}

// Subclasse de TwoDShape para retângulos.
class Rectangle extends TwoDShape {
    // Construtor padrão.
    Rectangle() {
        super();
    }

    // Construtor para Rectangle.
    Rectangle(double w, double h) {
        super(w, h, "rectangle"); // chama construtor da superclasse
    }

    // Constrói um quadrado.
    Rectangle(double x) {
        super(x, "rectangle"); // chama construtor da superclasse
    }
}
```

```
// Constrói um objeto a partir de outro.
Rectangle(Rectangle ob) {
    super(ob); // passa o objeto para o construtor de TwoDShape
}

boolean isSquare() {
    if(getWidth() == getHeight()) return true;
    return false;
}

// Sobrepõe area() para Rectangle.
double area() { ← Sobrepõe area() para Rectangle.
    return getWidth() * getHeight();
}

class DynShapes {
    public static void main(String[] args) {
        TwoDShape[] shapes = new TwoDShape[5];

        shapes[0] = new Triangle("outlined", 8.0, 12.0);
        shapes[1] = new Rectangle(10);
        shapes[2] = new Rectangle(10, 4);
        shapes[3] = new Triangle(7.0);
        shapes[4] = new TwoDShape(10, 20, "generic");           A versão apropriada de
                                                               area() é chamada para
                                                               cada forma.

        for(TwoDShape shape : shapes) {
            System.out.println("object is " + shape.getName());
            System.out.println("Area is " + shape.area()); ←
            System.out.println();
        }
    }
}
```

A saída do programa é mostrada abaixo:

```
object is triangle
Area is 48.0

object is rectangle
Area is 100.0

object is rectangle
Area is 40.0

object is triangle
Area is 24.5

object is generic
area() must be overridden
Area is 0.0
```

Examinemos esse programa em detalhes. Em primeiro lugar, como explicado, agora `area()` faz parte da classe `TwoDShape` e é sobreposto por `Triangle` e `Rectangle`. Dentro de `TwoDShape`, `area()` ganha uma implementação de espaço reservado que apenas informa ao usuário que esse método deve ser sobreposto por uma subclasse. Cada sobreposição de `area()` fornece uma implementação que é adequada ao tipo de objeto encapsulado pela subclasse. Logo, se você implementasse uma classe de elipse, por exemplo, `area()` teria que calcular a área de uma elipse.

Há outro recurso importante no programa anterior. Observe que `shapes` é declarada em `main()` como um array de objetos `TwoDShape`. No entanto, os elementos desse array recebem referências `Triangle`, `Rectangle` e `TwoDShape`. Isso é válido porque, como explicado, uma referência da superclasse pode referenciar um objeto da subclasse. O programa percorre então o array, exibindo informações sobre cada objeto. Embora muito simples, esse caso ilustra o poder tanto da herança quanto da sobreposição de métodos. O tipo de objeto referenciado por uma variável de referência da superclasse é determinado no tempo de execução e tratado de acordo. Se um objeto for derivado de `TwoDShape`, sua área poderá ser obtida com uma chamada a `area()`. A interface dessa operação é a mesma, não importando o tipo de forma usado.

Verificação do progresso

1. O que é sobreposição de métodos?
2. Por que a sobreposição de métodos é importante?
3. Quando um método sobreposto é chamado por uma referência da superclasse, que versão do método é executada?

USANDO CLASSES ABSTRATAS

Você pode querer criar uma superclasse que defina uma forma generalizada para ser compartilhada por todas as suas subclasses, deixando que cada subclasse insira os detalhes. Esse tipo de classe determina a natureza dos métodos que as subclasses devem implementar, mas não fornece uma implementação de um ou mais desses métodos. Uma maneira de essa situação ocorrer é quando uma superclasse não pode criar uma implementação significativa de um método. É esse o caso da versão de `TwoDShape` usada no exemplo anterior. A definição do método `area()` é apenas um espaço reservado. Ele não calculará e exibirá a área de nenhum tipo de objeto.

Como você verá ao criar suas próprias hierarquias de classes, é comum um método não ter definição significativa no contexto de sua superclasse. Você pode

Respostas:

1. A sobreposição de métodos ocorre quando uma subclasse define um método que tem a mesma assinatura e tipo de retorno de um método de sua superclasse.
2. A sobreposição de métodos permite que Java dê suporte ao polimorfismo.
3. A versão de um método sobreposto que será executada é determinada pelo tipo do objeto referenciado no momento da chamada. Logo, essa escolha é feita no tempo de execução.

tratar essa situação de duas maneiras. Uma maneira, como mostrado no exemplo anterior, é exibir uma mensagem de aviso. Embora essa abordagem possa ser útil em certas situações – como a depuração –, geralmente não é apropriada. Você pode ter métodos que precisem ser sobrepostos pela subclasse para que esta tenha algum sentido. Considere a classe **Triangle**. Ela ficaria incompleta se **area()** não fosse definido. Nesse caso, você quer alguma maneira de assegurar que uma subclasse sobreponha realmente todos os métodos necessários. A solução Java para esse problema é o *método abstrato*.

O método abstrato é criado pela especificação do modificador de tipo **abstract**. Ele não contém corpo e, portanto, não é implementado pela superclasse. Logo, uma subclasse deve sobrepor-lo – ela não pode apenas usar a versão definida na superclasse. Para declarar um método abstrato, use esta forma geral:

```
abstract tipo nome(lista-parâmetros);
```

Como ficou claro, não há um corpo de método presente. O modificador **abstract** só pode ser usado em métodos comuns. Ele não pode ser aplicado a métodos **static** ou a construtores.

Uma classe que contém um ou mais métodos abstratos também deve ser declarada como abstrata precedendo sua declaração **class** com o modificador **abstract**. Já que uma classe abstrata não define uma implementação completa, não podem existir objetos dessa classe. Logo, tentar criar um objeto de uma classe abstrata usando **new** resultará em um erro de tempo de compilação.

Quando uma subclasse herda uma classe abstrata, ela deve implementar todos os métodos abstratos da superclasse. Se não implementar, também deve ser especificada como **abstract**. Portanto, o atributo **abstract** é herdado até uma implementação completa ser obtida.

Usando uma classe abstrata, você pode melhorar a classe **TwoDShape**. Como não há um conceito significativo de área para uma figura bidimensional indefinida, a versão a seguir do programa anterior declara **area()** como **abstract** dentro de **TwoDShape**, e **TwoDShape** como **abstract**. Ou seja, é claro que todas as classes derivadas de **TwoDShape** devem sobrepor **area()**.

```
// Cria uma classe abstrata.
abstract class TwoDShape { ←———— Agora TwoDShape é abstrata.
    private double width;
    private double height;
    private String name;

    // Um construtor padrão.
    TwoDShape() {
        width = height = 0.0;
        name = "none";
    }

    // Construtor parametrizado.
    TwoDShape(double w, double h, String n) {
        width = w;
        height = h;
        name = n;
    }
}
```

```

}

// Constrói objeto com largura e altura iguais.
TwoDShape(double x, String n) {
    width = height = x;
    name = n;
}

// Constrói um objeto a partir de outro.
TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
}

// Métodos acessadores para width e height.
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }

String getName() { return name; }

void showDim() {
    System.out.println("Width and height are " +
                       width + " and " + height);
}

// Agora, area() é abstrato.
abstract double area(); ←———— Transforma area() em um método abstrato.
}

// Uma subclasse de TwoDShape para triângulos.
class Triangle extends TwoDShape {
    private String style;

    // Construtor padrão.
    Triangle() {
        super();
        style = "none";
    }

    // Construtor para Triangle.
    Triangle(String s, double w, double h) {
        super(w, h, "triangle");

        style = s;
    }

    // Construtor com um argumento.
    Triangle(double x) {
}

```

```
super(x, "triangle"); // chama construtor da superclasse

// padrão de style é filled
style = "filled";
}

// Constrói um objeto a partir de outro.
Triangle(Triangle ob) {
    super(ob); // passa objeto para construtor de TwoDShape
    style = ob.style;
}

double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    System.out.println("Triangle is " + style);
}
}

// Subclasse de TwoDShape para retângulos.
class Rectangle extends TwoDShape {
    // Construtor padrão.
    Rectangle() {
        super();
    }

    // Construtor para Rectangle.
    Rectangle(double w, double h) {
        super(w, h, "rectangle"); // chama construtor da superclasse
    }

    // Constrói um quadrado.
    Rectangle(double x) {
        super(x, "rectangle"); // chama construtor da superclasse
    }

    // Constrói um objeto a partir de outro.
    Rectangle(Rectangle ob) {
        super(ob); // passa objeto para construtor de TwoDShape
    }

    boolean isSquare() {
        if(getWidth() == getHeight()) return true;
        return false;
    }

    double area() {
        return getWidth() * getHeight();
    }
}
```

```

    }

class AbsShape {
    public static void main(String[] args) {
        TwoDShape[] shapes = new TwoDShape[4];

        shapes[0] = new Triangle("outlined", 8.0, 12.0);
        shapes[1] = new Rectangle(10);
        shapes[2] = new Rectangle(10, 4);
        shapes[3] = new Triangle(7.0);

        for(TwoDShape shape : shapes) {
            System.out.println("object is " + shape.getName());
            System.out.println("Area is " + shape.area());
            System.out.println();
        }
    }
}

```

Como o programa ilustra, todas as subclasses de **TwoDShape** devem sobrepor **area()**. Para confirmar isso, tente criar uma classe que não sobreponha **area()**. Você verá um erro de tempo de compilação. Certamente ainda é possível criar uma referência de objeto de tipo **TwoDShape**, o que o programa faz. Contudo, não é mais possível declarar objetos de tipo **TwoDShape**. Portanto, em **main()** o array **shapes** foi diminuído para 4, e não é mais criado um objeto **TwoDShape**.

Um último ponto: observe que **TwoDShape** ainda inclui os métodos **showDim()** e **getName()** e que eles não são modificados por **abstract**. É perfeitamente aceitável – na verdade, muito comum – uma classe abstrata conter métodos concretos que uma classe possa usar da forma em que se encontram. Só os métodos declarados como **abstract** têm que ser sobrepostos pelas subclasses.

Verificação do progresso

1. O que é um método abstrato? Como ele é criado?
2. Quando uma classe deve ser declarada abstrata?
3. Podemos instanciar um objeto de uma classe abstrata?

Respostas:

1. Um método abstrato é aquele que não tem um corpo. Logo, é composto por um tipo de retorno, um nome e uma lista de parâmetros e é precedido pela palavra-chave **abstract**.
2. Quando contém pelo menos um método abstrato.
3. Não.

USANDO final

Mesmo com a sobreposição de métodos e a herança sendo tão poderosas, podemos querer evitar que ocorram. Por exemplo, podemos ter uma classe que encapsule o controle de algum dispositivo de hardware. Além disso, essa classe pode dar ao usuário a oportunidade de inicializar o dispositivo, fazendo uso de informações privadas. Nesse caso, não vamos querer que os usuários de nossa classe possam sobrepor o método de inicialização. Qualquer que seja a razão, em Java, com o uso da palavra-chave **final**, é fácil impedir que um método seja sobreposto ou que uma classe seja herdada.

A palavra-chave final impede a sobreposição

Para impedir que um método seja sobreposto, especifique **final** como modificador no início de sua declaração. Métodos declarados como **final** não podem ser sobrepostos. O fragmento a seguir ilustra **final**:

```
class A {  
    final void meth() {  
        System.out.println("This is a final method.");  
    }  
}  
  
class B extends A {  
    void meth() { // ERRO! Não pode sobrepor.  
        System.out.println("Illegal!");  
    }  
}
```

Uma vez que **meth()** é declarado como **final**, não pode ser sobreposto em **B**. Se você tentar fazê-lo, ocorrerá um erro de tempo de compilação.

A palavra-chave final impede a herança

Você pode impedir que uma classe seja herdada precedendo sua declaração com **final**. A declaração de uma classe como **final** também declara implicitamente todos os seus métodos como **final**. Como era de se esperar, é inválido declarar uma classe como **abstract** e **final**, uma vez que uma classe abstrata é individualmente incompleta e depende de suas subclasses para fornecer implementações completas.

Aqui está um exemplo de uma classe **final**:

```
final class A {  
    // ...  
}  
  
// A classe seguinte é inválida.  
class B extends A { // ERRO! Não pode criar uma subclasses de A  
    // ...  
}
```

Como os comentários sugerem, é inválido **B** herdar **A**, já que **A** é declarada como **final**.

Usando final com membros de dados

Além dos usos que acabamos de mostrar, **final** também pode ser aplicada a variáveis membros para criar o que seriam constantes nomeadas. Se você preceder o nome da variável de uma classe com **final**, seu valor não poderá ser alterado durante todo o tempo de vida do programa. Você pode, claro, dar a essa variável um valor inicial. Por exemplo, no Capítulo 6, uma classe simples de gerenciamento de erros chamada **ErrorMsg** foi mostrada. Essa classe mapeava um string legível por humanos para um código de erro. Aqui, a versão original da classe será melhorada pelo acréscimo de constantes **final**, que representam os erros. Agora, em vez de passar para **getErrorMsg()** um número como 2, você pode passar a constante int nomeada **DISKERR**.

```
// Retorna um objeto String.
class ErrorMsg {
    // Códigos de erro.
    final int OUTERR    = 0;
    final int INERR     = 1; ← Declara constantes final.
    final int DISKERR   = 2;
    final int INDEXERR = 3;

    String[] msgs = {
        "Output Error",
        "Input Error",
        "Disk Full",
        "Index Out-Of-Bounds"
    };

    // Retorna a mensagem de erro.
    String getErrorMsg(int i) {
        if(i >=0 & i < msgs.length)
            return msgs[i];
        else
            return "Invalid Error Code";
    }
}

class FinalD {
    public static void main(String[] args) {
        ErrorMsg err = new ErrorMsg();
        System.out.println(err.getErrorMsg(err.OUTERR));
        System.out.println(err.getErrorMsg(err.DISKERR));
    }
}
```

Observe como as constantes **final** são usadas em **main()**. Uma vez que são membros da classe **ErrorMsg**, devem ser acessadas via um objeto dessa classe. (Consulte a próxima seção Pergunte ao Especialista para ver uma abordagem alternativa.)

É claro que também podem ser herdadas pelas subclasses e acessadas diretamente dentro delas.

Por uma questão estilística, muitos programadores de Java usam identificadores maiúsculos em constantes **final**, como no exemplo anterior, mas essa não é uma regra fixa.

Pergunte ao especialista

P Variáveis membros **final** podem ser tornadas **static**? A palavra-chave **final** pode ser usada em parâmetros de métodos e variáveis locais?

R A resposta às duas perguntas é sim. Transformar uma variável membro **final** em **static** permite que você referencie a constante pelo nome de sua classe em vez de por um objeto. Por exemplo, se as constantes de **ErrorMsg** fossem modificadas por **static**, as instruções **println()** de **main()** teriam esta aparência:

```
| System.out.println(err.getErrorMsg(ErrorMsg.OUTERR)) ;
| System.out.println(err.getErrorMsg(ErrorMsg.DISKERR)) ;
```

Com frequência, o uso de uma variável **static final** é uma abordagem melhor para esses tipos de contantes, e normalmente é a abordagem que vemos.

A declaração de um parâmetro **final** impede que ele seja alterado dentro do método. A declaração de uma variável local **final** impede que ela receba um valor mais de uma vez.

Verificação do progresso

1. Como podemos impedir que um método seja sobreposto?
2. Se uma classe for declarada como **final**, ela pode ser herdada?

A CLASSE Object

Java define uma classe especial chamada **Object**, que é uma superclasse implícita de todas as outras classes. Em outras palavras, todas as outras classes são subclasses de **Object**. Ou seja, uma variável de referência de tipo **Object** pode referenciar um objeto de qualquer outra classe. Além disso, uma vez que os arrays são implementados como classes, uma variável de tipo **Object** também pode referenciar qualquer array.

Respostas:

1. Precedendo sua declaração com a palavra-chave **final**.
2. Não.

Object define os métodos a seguir, portanto, eles estão disponíveis em todos os objetos:

Método	Finalidade
Object clone()	Cria um novo objeto igual ao objeto que está sendo clonado.
boolean equals(Object <i>objeto</i>)	Determina se um objeto é igual ao outro.
void finalize()	Chamado antes de um objeto não usado ser reciclado.
Class<?>getClass	Obtém a classe de um objeto no tempo de execução.
int hashCode()	Retorna o código hash associado ao objeto chamador.
void notify()	Retoma a execução de uma thread que está esperando no objeto chamador.
void notifyAll()	Retoma a execução de todas as threads que estão esperando no objeto chamador.
String toString()	Retorna um string que descreve o objeto.
void wait() void wait(long <i>milissegundos</i>) void wait(long <i>milissegundos</i> , int <i>nanossegundos</i>)	Espera outra thread de execução.

Os métodos **getClass()**, **notify()**, **notifyAll()** e **wait()** são declarados como **final**. Você pode sobrepor os outros. Vários desses métodos serão descritos posteriormente no livro. No entanto, veremos dois agora: **equals()** e **toString()**. O método **equals()** compara dois objetos. Ele retorna **true** se os objetos forem equivalentes, caso contrário, retorna **false**. A implementação de **Object** para **equals()** apenas verifica se a referência chamadora aponta para o mesmo objeto que foi passado como argumento. No entanto, geralmente **equals()** é sobreposto para determinar se dois objetos são iguais em seu conteúdo. O método **toString()** retorna um string contendo a descrição do objeto em que é chamado. Além disso, esse método é chamado automaticamente quando um objeto é exibido com o uso de **println()**. Muitas classes o sobreponem. Isso permite que personalizem uma descrição especificamente para os tipos de objetos que criam.

Um último ponto: observe a sintaxe incomum no tipo de retorno de **getClass()**. Ela pertence aos *tipos genéricos* Java. Os genéricos permitem que o tipo de dado usado por uma classe ou método seja especificado como parâmetro. Eles serão discutidos no Capítulo 14.

EXERCÍCIOS

1. Uma superclasse tem acesso aos membros de uma subclasse? E a subclasse pode acessar os membros de uma superclasse?
2. Crie uma subclasse de **TwoDShape** chamada **Circle**. Inclua um método **area()** que calcule a área do círculo e um construtor que use **super** para inicializar a parte referente a **TwoDShape**.

3. Como impedir que uma subclasse tenha acesso a um membro de uma superclasse?
4. Descreva a finalidade e a aplicação das duas versões de **super**.
5. Dada a hierarquia a seguir:

```
class Alpha { ...  
  
class Beta extends Alpha { ...  
  
Class Gamma extends Beta { ...
```

Em que ordem os construtores dessas classes são executados quando um objeto **Gamma** é instanciado?

6. Uma referência da superclasse pode referenciar um objeto da subclasse. Explique por que isso é importante no âmbito da sobreposição de métodos.
7. O que é uma classe abstrata?
8. Como impedir que um método seja sobreposto? E que uma classe seja herdada?
9. Explique como a herança, a sobreposição de métodos e as classes abstratas são usadas para dar suporte ao polimorfismo.
10. Que classe é superclasse de todas as outras classes?
11. Uma classe que contém pelo menos um método abstrato deve ser declarada como abstrata. Verdadeiro ou falso?
12. Que palavra-chave é usada para criar uma constante nomeada?
13. Suponhamos que uma classe **A** tivesse os métodos **m1()**, **m2()** e **m3()** e uma subclasse **B** tivesse os métodos **m4()** e **m5()**.
 - A. Quais dos cinco métodos podem ser acessados por objetos da classe **A**?
 - B. Quais dos cinco métodos podem ser acessados por objetos da classe **B**?
 - C. Quais dos cinco métodos podem ser chamados com o uso de uma variável de tipo **A** que refencie um objeto de tipo **B**?
 - D. Suponhamos que **m1()** fosse declarado como **private**. Agora quais dos cinco métodos podem ser acessados por objetos da classe **B**?
14. Crie uma classe **Person** com variáveis de instância privadas para o nome e a data de nascimento da pessoa. Adicione métodos acessadores apropriados para essas variáveis. Em seguida, crie uma subclasse **CollegeGraduate** com variáveis de instância privadas para a média das notas (GPA, grade point average) e o ano de graduação do aluno e acessadores apropriados para essas variáveis. Lembre-se de incluir construtores apropriados para suas classes. Depois crie uma classe com um método **main()** que as demonstre.
15. A prática padrão em muitas situações é declarar como **private** todas as variáveis de instância não finais e fornecer métodos acessadores apropriados para o acesso a elas. Para algumas variáveis de instância é apropriado que haja um método “getter” para a obtenção do valor da variável, mas não um método “setter” que altere o seu valor. Pense em um exemplo de uma classe e uma variável de instância em que essa situação ocorra e explique por que não é apropriado que haja um método “setter” para a variável.

16. No Capítulo 6, uma classe **FailSoftArray** foi criada para impedir a ocorrência de erros de limite quando o array fosse acessado. No entanto, a classe não é totalmente à prova de falhas devido ao fato de a variável de instância **length** ser **public**. Se quiséssemos fazer mau uso intencional da classe atribuindo a **length** um valor maior do que o tamanho do array subjacente, uma tentativa de acessar o array além de seus limites não seria impedida. O que devemos fazer à variável **length** para impedir que esse tipo de má utilização ocorra? Lembre-se: **length** tem que permanecer **public** para que os usuários possam saber qual é o tamanho do array.
17. Sobreponha o método **toString()** herdado pela classe **FailSoftArray** do Capítulo 6 para que retorne um string exibindo o conteúdo do array. Por exemplo, se o array tiver 1, 2, 3 e 4, o método **toString()** deve retornar o string “{1, 2, 3, 4}”. (Para obter mais informações sobre a sobreposição de **toString()**, consulte o Capítulo 22.)
18. Suponhamos que quiséssemos melhorar nossa modelagem de veículos criando objetos **Tire** para representar seus pneus. Como a classe **Tire** deve estar relacionada à classe **Vehicle**? Deve ser subclasse de **Vehicle**? Por quê?
19. Suponhamos que você tivesse as classes **Boat**, **House**, **HouseBoat** e **BoatHouse** para representar barcos, casas, barcos-moradia (uma casa flutuante) e casas de barco (uma construção para guardar barcos), respectivamente. Alguma das classes deve ser subclasse das outras? Por quê?
20. O segmento de código a seguir é válido em Java? Por quê?

```
| int [] array = {1,2,3};  
| Object [] data = {"hello", new Object(), array};
```

21. Suponhamos que você estivesse criando um programa Java que fizesse muitos cálculos matemáticos envolvendo π . Se só precisasse de duas casas decimais de precisão, poderia usar 3,14 em todos os locais em que π fosse necessário em seu código (por exemplo, em fórmulas como `area = 3,14*r*r`) ou declarar uma constante **PI** igual a 3,14 e usá-la em vez de 3,14 onde π fosse necessário (usando fórmulas como `area = PI*r*r`).
- Mostre como seria a declaração dessa constante **PI** em Java.
 - Dê pelo menos duas razões para o uso da constante **PI** ser melhor do que usar 3,14.
22. Um dos usos da herança é para eliminarmos a duplicação de código. Por exemplo, suponhamos que você tivesse duas classes, **A** e **B**, e ambas tivessem métodos **getData()** idênticos que extraíssem dados de arquivos, mas as classes fizessem coisas diferentes com os dados depois de sua extração. Descreva uma maneira de a herança ser usada para evitar o código duplicado.
23. Suponhamos que uma classe **A** declarasse um método **equals()** que usasse um parâmetro de tipo **A**. Essa classe também herdará o método **equals()** da classe **Object**. O método **equals()** da classe **A** sobrepõe o método **equals()** herdado ou o método **equals()** é sobreescrito na classe **A**?

8

Interfaces

PRINCIPAIS HABILIDADES E CONCEITOS

- Entender os aspectos básicos da interface
- Saber a forma geral de uma interface
- Implementar uma interface
- Aplicar referências de interface
- Usar constantes de interface
- Estender interfaces
- Aninhar interfaces

Na programação orientada a objetos, às vezes é útil definir o que uma classe deve fazer, mas não como ela o fará. Já vimos uma maneira de fazer isso: o método abstrato. Um método abstrato define a assinatura de um método, mas não fornece implementação. Uma subclasse deve fornecer sua própria implementação de cada método abstrato definido por sua superclasse. Portanto, um método abstrato especifica a *interface* do método, mas não a *implementação*. Embora as classes e métodos abstratos sejam úteis, podemos levar esse conceito um passo adiante. Em Java, podemos separar totalmente a interface de uma classe de sua implementação usando a palavra-chave **interface**. Esse é o assunto deste capítulo.

ASPECTOS BÁSICOS DA INTERFACE

Em Java, uma **interface** define um conjunto de métodos que será implementado por uma classe. Sintaticamente, as interfaces são semelhantes às classes abstratas, exceto pelos métodos não poderem incluir um corpo. Isto é, uma interface não fornece implementação dos métodos que define. Logo, ela apenas especifica o que deve ser feito, mas não como. Colocado de maneira mais formal, uma interface é uma estrutura que descreve a funcionalidade sem especificar uma implementação.

Quando uma interface é definida, não há limite para o número de classes que pode implementá-la. Isso permite que duas ou mais classes forneçam a mesma funcionalidade, porém de maneiras diferentes. Além disso, uma classe pode implementar qualquer número de interfaces. Portanto, a mesma classe pode fornecer várias funcionalidades bem definidas.

Para implementar uma interface, a classe deve fornecer corpos (implementações) para os métodos descritos nela. Cada classe é livre para determinar os detalhes de sua própria implementação. Duas classes podem implementar os métodos definidos por uma interface diferentemente, mas ambas darão suporte ao mesmo conjunto de métodos. Logo, um código que souber da existência da interface poderá usar objetos das duas classes, porque a interface dos objetos é a mesma. Ao fornecer a interface, Java permite que você utilize plenamente o aspecto “uma interface, vários métodos” do polimorfismo.

Uma interface é declarada com o uso da palavra-chave **interface**. Esta é a forma geral simplificada de uma declaração de interface:

```
acesso interface nome {
    tipo-ret nome-método1(lista-param);
    tipo-ret nome-método2(lista-param);
    // ...
    tipo-ret nome-métodoN(lista-param);
}
```

Aqui, *acesso* pode ser **public** ou não usado. Quando não é incluído um modificador de acesso, isso resulta no acesso padrão. Embora o acesso padrão seja apropriado em muitas aplicações, com frequência as interfaces são declaradas como **public** porque isso as torna acessíveis para um conjunto maior de códigos. (Quando uma **interface** é declarada como **public**, deve ficar em um arquivo de mesmo nome.) O nome da interface é especificado por *nome* e pode ser qualquer identificador válido.

Em uma **interface**, os métodos são declarados com o uso apenas de seu tipo de retorno e assinatura. São basicamente métodos abstratos. Como explicado, nenhum método da interface pode ter implementação. É responsabilidade de cada classe que inclua uma **interface** fornecer as implementações. Os métodos da interface são implicitamente **public**.

CRIANDO UMA INTERFACE

Como acabamos de explicar, a principal finalidade de uma interface é especificar o que deve ser feito, mas não como. Esse é um conceito poderoso em programação. Para entender por que, examinemos um exemplo simples. Suponhamos que você quisesse criar um conjunto de classes que gerassem diferentes tipos de séries numéricas. Essas séries poderiam ser de números pares começando em 2, números aleatórios ou o conjunto de números primos, para citar apenas algumas. No entanto, em todos os casos, você quer que as classes funcionem da mesma maneira e que todas tenham métodos para obter o próximo número da série, levar novamente a série ao seu início e especificar um valor inicial. Dessa forma, o código que usar um gerador de série numérica poderá facilmente ser alterado para usar um diferente. Esse tipo de situação é ideal para uma interface, porque ela pode declarar os métodos que cada gerador de série numérica usará, mas cada classe poderá implementar sua própria série numérica específica.

Para colocarmos o caso anterior em bases concretas, aqui está um exemplo de uma definição simples de **interface**. Ela especifica uma interface chamada **Series** que descreve os métodos usados na geração de uma série de números.

```
public interface Series {  
    int getNext(); // retorna o próximo número da série  
    void reset(); // reinicia  
    void setStart(int x); // define o valor inicial  
}
```

Observe que **Series** define três métodos. O primeiro é **getNext()**, que obtém o próximo número da série. O segundo é **reset()**, que retorna a série ao seu valor inicial. O último é **setStart()**, que define o ponto inicial. Todas as classes que implementarem **Series** devem fornecer esses três métodos. Portanto, poderão ser usadas da mesma forma, chamando o mesmo conjunto de métodos. Outra coisa: aqui, **Series** é declarada como **public**, logo, deve ser mantida em um arquivo chamado **Series.java**.

IMPLEMENTANDO UMA INTERFACE

Quando uma **interface** tiver sido definida, uma ou mais classes poderão implementá-la. Para implementar uma interface, siga estas duas etapas:

1. Em uma declaração de classe, inclua uma cláusula **implements** que especifique a interface que está sendo implementada.
2. Dentro da classe, implemente os métodos definidos pela interface.

A cláusula **implements** especifica o nome da interface que a classe implementará. A forma geral de uma classe que inclui a cláusula **implements** é esta:

```
class nomeclasse extends superclasse implements interface {  
    // corpo-classe  
}
```

Na implementação de mais de uma interface, as interfaces são separadas por uma vírgula. Obviamente, a cláusula **extends** é opcional, mas seu uso nos permite herdar uma classe e implementar uma ou mais interfaces ao mesmo tempo.

Dentro da classe, você deve definir todos os métodos especificados pela interface que está sendo implementada. Os métodos que implementam uma interface devem ser declarados como **public**. Além disso, o tipo de retorno e a assinatura do método implementador deve coincidir exatamente com o tipo de retorno e a assinatura especificados na declaração da **interface**.

Este é um exemplo que mostra uma implementação da interface **Series** vista anteriormente. Ele cria uma classe chamada **ByTwos**, que gera uma série de números, cada um duas unidades maior do que o anterior. Observe o uso da cláusula **implements**.

```
// Implementa Series.  
class ByTwos implements Series {  
    int start;           ↑  
    int val;             ————— Implementa a interface Series.
```

```

ByTwos() {
    start = 0;
    val = 0;
}

// Implementa os métodos especificados por Series.
public int getNext() {
    val += 2;
    return val;
}

public void reset() {
    val = start;
}

public void setStart(int x) {
    start = x;
    val = x;
}
}

```

Observe que os métodos **getNext()**, **reset()** e **setStart()** são declarados como **public**. Como explicado, isso é necessário. Sempre que você implementar um método definido por uma interface, ele deve ser implementado como **public**, porque todos os métodos especificados por uma interface são implicitamente **public**.

Aqui está uma classe que demonstra **ByTwos**:

```

// Demonstra o uso de Series.
class SeriesDemo {
    public static void main(String[] args) {
        ByTwos ob = new ByTwos();

        for(int i=0; i < 5; i++)
            System.out.println("Next value is " + ob.getNext());

        System.out.println("\nResetting");
        ob.reset();
        for(int i=0; i < 5; i++)
            System.out.println("Next value is " + ob.getNext());

        System.out.println("\nStarting at 100");
        ob.setStart(100);
        for(int i=0; i < 5; i++)
            System.out.println("Next value is " + ob.getNext());
    }
}

```

A saída desse programa é mostrada abaixo:

```

| Next value is 2
| Next value is 4

```

```
Next value is 6
Next value is 8
Next value is 10
Resetting
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Starting at 100
Next value is 102
Next value is 104
Next value is 106
Next value is 108
Next value is 110
```

É permitido e comum classes que implementam uma interface definirem membros adicionais. Por exemplo, a versão a seguir de **ByTwos** adiciona o método **getPriorVal()**, que retorna o valor anterior gerado:

```
// Implementa Series e adiciona getPriorVal().
class ByTwos implements Series {
    int start;
    int val;
    int priorVal;

    ByTwos() {
        start = 0;
        val = 0;
        priorVal = -2;
    }

    // Implementa os métodos especificados por Series.
    public int getNext() {
        priorVal = val;
        val += 2;
        return val;
    }

    public void reset() {
        val = start;
        priorVal = start - 2;
    }

    public void setStart(int x) {
        start = x;
        val = x;
        priorVal = x - 2;
    }

    // Retorna o valor anterior. Esse método não é definido por Series.
```

```

    int getPriorVal() { ← Adiciona um método não definido por Series.
        return priorVal;
    }
}

```

Ainda que **ByTwos** tenha adicionado o método **getPriorVal()**, isso não muda o fato de que ela implementa **Series**. A única obrigação que uma classe tem quando implementa uma interface é fornecer os métodos definidos pela interface. Ela *não* fica limitada a fornecer *apenas* esses métodos. A classe pode fornecer qualquer funcionalidade adicional desejada. Logo, você pode usar a mesma classe **SeriesDemo** com a nova versão de **ByTwos**.

Há algo mais que devemos destacar nessa versão de **ByTwos()**. A inclusão de **getPriorVal()** demandou uma alteração nas implementações dos métodos definidos por **Series**. No entanto, já que a interface dos métodos permaneceu igual, a alteração ocorreu normalmente e não prejudicou nenhum código preexistente. Essa é uma das principais vantagens das interfaces.

Um número ilimitado de classes pode implementar uma **interface**. Por exemplo, esta é uma classe chamada **ByThrees** que gera uma série composta por múltiplos de três. Logo, ela implementa uma série diferente.

```

// Implementa Series de uma maneira diferente.
class ByThrees implements Series { ← Implementa Series de uma maneira diferente.
    int start;
    int val;

    ByThrees() {
        start = 0;
        val = 0;
    }

    // Implementa os métodos especificados por Series.
    public int getNext() {
        val += 3;
        return val;
    }

    public void reset() {
        val = start;
    }

    public void setStart(int x) {
        start = x;
        val = x;
    }
}

```

Embora a implementação de **ByThrees** seja diferente da de **ByTwos**, as duas implementam a mesma interface **Series**. Ou seja, as duas classes podem ser usadas da mesma maneira. Por exemplo, nos dois casos, **getNext()** obtém o próximo elemento da série.

Como regra geral, uma classe deve definir todos os métodos especificados pela interface que ela está implementando. Contudo, se uma classe implementar uma interface mas não definir todos os métodos, ela deve ser declarada como **abstract**. Não poderão ser criados objetos dessa classe, mas ela poderá ser usada como superclasse abstrata, permitindo que subclasses forneçam a implementação completa.

Verificação do progresso

1. O que é uma interface? Que palavra-chave é usada para declarar uma?
2. Para que serve **implements**?

USANDO REFERÊNCIAS DE INTERFACES

Com você sabe, quando definimos uma classe, estamos criando um novo tipo de referência. O mesmo ocorre com as interfaces. Uma declaração de interface também cria um novo tipo de referência. Quando uma classe implementa uma interface, está adicionando o tipo da interface ao seu tipo. Como resultado, uma instância de uma classe que implementa uma interface também é uma instância desse tipo de interface. Por exemplo, uma instância de **ByTwos** também é uma instância de **Series**.

Já que uma interface define um tipo, você pode declarar uma variável de referência de um tipo de interface. Em outras palavras, você pode criar uma variável de referência de interface. Uma variável assim tem uma propriedade muito importante: pode referenciar qualquer objeto que implemente a interface. (Ou seja, pode referenciar qualquer instância de seu tipo.) Quando você chamar um método em um objeto por intermédio de uma referência de interface, a versão do método implementada pelo objeto será executada. Esse processo é semelhante ao uso de uma referência da superclasse no acesso a um objeto da subclasse, como descrito no Capítulo 7.

O exemplo a seguir ilustra o processo. A classe **SeriesDemo2** usa a mesma variável de referência de interface para chamar métodos em objetos tanto de **ByTwos** quanto de **ByThrees**.

```
class SeriesDemo2 {
    public static void main(String[] args) {
        ByTwos twoOb = new ByTwos();
        ByThrees threeOb = new ByThrees();

        Series iRef; // uma referência de interface

        for(int i=0; i < 5; i++) {
            iRef = twoOb; // referencia um objeto ByTwos
            System.out.println("Next ByTwos value is " +
```

Respostas:

1. Uma interface define os métodos que uma classe deve implementar, mas não define uma implementação própria. Ela é declarada pela palavra-chave **interface**.
2. Para implementar uma interface, devemos incluí-la em uma classe usando a palavra-chave **implements**.

```

    iRef.getNext(); <-- Acessa um objeto
    iRef = threeOb; // referencia um objeto ByThrees
    System.out.println("Next ByThrees value is " +
    iRef.getNext(); <-- por meio de uma
}                                referência de interface.
}
}
}

```

A saída é mostrada aqui:

```

Next ByTwos value is 2
Next ByThrees value is 3
Next ByTwos value is 4
Next ByThrees value is 6
Next ByTwos value is 6
Next ByThrees value is 9
Next ByTwos value is 8
Next ByThrees value is 12
Next ByTwos value is 10
Next ByThrees value is 15

```

Em **main()**, **iRef** é declarada como referência à interface **Series**. Ou seja, pode ser usada para armazenar uma referência a qualquer objeto que implemente **Series**. Nesse caso, é usada para referenciar **twoOb** e **threeOb**, que são objetos de tipo **ByTwos** e **ByThrees**, respectivamente. Isso é possível porque ambos implementam **Series**. Sempre que um dos métodos definidos por **Series** é chamado por intermédio de **iRef**, a versão do método implementada pelo objeto que está sendo referenciado é executada.

Há algo mais que devemos observar nesse exemplo: uma variável de referência de interface só tem conhecimento dos métodos declarados por sua declaração **interface**. Logo, **iRef** não pode ser usada para acessar nenhuma outra variável ou método fornecido por uma classe implementadora. Por exemplo, se **ByTwos** incluísse o método **getPriorVal()** mostrado anteriormente, ele não poderia ser acessado por intermédio de **iRef**.

Embora o exemplo anterior mostre a mecânica da chamada de métodos por intermédio de uma referência de interface, ele não mostra um de seus benefícios mais importantes. Como você sabe, o polimorfismo é um preceito-chave da programação orientada a objetos. O princípio que o define é o de que funcionalidades relacionadas podem ser acessadas por intermédio de uma interface comum. Uma vez que você tiver entendido a interface, poderá usar qualquer implementação específica dela. Mas talvez o mais importante seja que a implementação pode mudar sem afetar o código que usa a interface. Chamar métodos de interface por intermédio de uma referência de interface o ajudará a apreender plenamente o benefício da filosofia “uma interface, vários métodos”.

Para entender por quê, considere uma classe que simule algum processo físico, como a movimentação de filas de clientes em um banco ou diferenças no lucro de colheitas baseadas na quantidade de chuva. Para fazer essa simulação, a classe pode precisar de um gerador de série numérica. No entanto, talvez você queira alterar a natureza da série para poder observar resultados diferentes. Poderia criar essa classe como mostrado aqui.

```

class Simulation {
    // numSeq referencia o gerador de série numérica
    // que será usado pela simulação.
    Series numSeq;
}

```

```
// Passa o gerador de série numérica que será usado
// pela instância de Simulation que está sendo construída.
Simulation(Series s) {
    numSeq = s;
}

// ...
}
```

Observe que o gerador de série é passado para **Simulation**, via seu construtor, como um parâmetro de tipo **Series**. Além disso, observe que uma referência a esse objeto é mantida em uma variável de instância chamada **numSeq**, que também é uma referência de tipo **Series**. Já que **Simulation** especifica o gerador de números pelo tipo de interface **Series**, em vez de embutir no código um tipo específico de gerador, você pode alterar facilmente o tipo de gerador de números usado. Por exemplo, estas duas declarações são válidas:

```
Simulation sim = new Simulation(new ByTwos());
Simulation sim2 = new Simulation(new ByThrees());
```

Na primeira, o gerador de números é uma instância de **ByTwos**. Na segunda, é um objeto **ByThrees**. É claro que ele poderia ser qualquer tipo de objeto, contanto que implementasse **Series**. Por exemplo, se uma série que refletisse uma distribuição normal (forma de sino) fosse necessária, você poderia criar uma classe chamada **NormalDist** que implementasse **Series** e passar uma instância de **NormalDist** para **Simulation**. Já que todas as implementações de **Series** são usadas da mesma forma, não seriam necessárias alterações em **Simulation**.

Resumindo: com a especificação da funcionalidade do gerador de números por seu tipo de interface (em vez de por uma implementação de tipo de classe específica), a interface nos possibilita projetar um código facilmente adaptável. Podemos simplesmente alterar o tipo de objeto passado para **Simulation** quando uma instância for construída. Nenhuma outra alteração é necessária.

O exemplo acima pode ser generalizado. A especificação da funcionalidade com o uso de uma referência de interface nos permite mudar o código elegantemente com o passar do tempo, sem prejudicá-lo. Contanto que a interface permaneça inalterada, tanto sua implementação quanto o código que a usa podem evoluir quando necessário. Com o uso da interface, nosso código ganha flexibilidade.

IMPLEMENTANDO VÁRIAS INTERFACES

Como mencionado, uma classe pode implementar mais de uma interface. Para fazê-lo, apenas especifique cada interface em uma lista separada por vírgulas. Obviamente, a classe deve implementar todos os métodos especificados por cada interface. Aqui está um exemplo simples:

```
interface IfA {
    void doSomething();
}
```

```

interface IfB {
    void doSomethingElse();
}

// Implementa tanto IfA quanto IfB.
class MyClass implements IfA, IfB {
    public void doSomething() {
        System.out.println("Doing something.");
    }

    public void doSomethingElse() {
        System.out.println("Doing something else.");
    }
}

```

Nesse exemplo, **MyClass** especifica tanto **IfA** quanto **IfB** em sua cláusula **implements**. Em seguida, implementa o método especificado por cada uma.

Em aplicativos do mundo real, é comum uma classe implementar mais de uma interface. Isso permite que ela forneça várias funcionalidades bem definidas sem ter que usar a herança de classes. Como você sabe, uma classe só pode herdar diretamente outra classe. Como resultado, pode ser difícil especificar funcionalidades adicionais sem recorrer a hierarquias com um topo sobrecarregado. As interfaces resolvem esse problema, permitindo que a classe especifique a funcionalidade adicional sem impactar a hierarquia de herança.

Pergunte ao especialista

P Na implementação de duas interfaces, o que acontece quando ambas declaram o mesmo método? Por exemplo, e se duas interfaces especificassem um método chamado **doSomething()**?

R Se uma classe implementar duas interfaces que declaram o mesmo método, a mesma implementação do método será usada para ambas. Ou seja, só uma versão do método é definida pela classe. Por exemplo, considere essa variação do exemplo que acabamos de mostrar:

```

// Tanto IfA quanto IfB declaram o método doSomething().
interface IfA {
    void doSomething();
}

interface IfB {
    void doSomething();
}

// Implementa tanto IfA quanto IfB
class MyClass implements IfA, IfB {

    // Esse método implementa tanto IfA quanto IfB.
    public void doSomething() {
        System.out.println("Doing something.");
    }
}

```

```

    }
}

class MultiImpDemo {
    public static void main(String[] args) {
        IfA aRef;
        IfB bRef;
        MyClass obj = new MyClass();

        // As duas interfaces usam o mesmo método doSomething().
        aRef = obj;
        aRef.doSomething();

        bRef = obj;
        bRef.doSomething();
    }
}

```

A saída é mostrada abaixo:

```

Doing something.
Doing something.

```

Nesse caso, tanto **IfA** quanto **IfB** declaram o mesmo método: **doSomething()**. Quando **MyClas** implementar essas interfaces, as duas usarão o mesmo **doSomething()**. Logo, o mesmo método será executado, seja **doSomething()** chamado por meio de uma referência a **IfA** ou a **IfB**.

Verificação do progresso

1. Uma variável de referência de interface pode apontar para um objeto que implemente essa interface?
2. Uma classe só pode implementar uma interface. Verdadeiro ou falso?

TENTE ISTO 8-1 Criando uma interface de pilha simples

```

ISimpleStack.java
FixedLengthStack.java
DynamicStack.java
ISimpleStackDemo.java

```

Para entender melhor o poder das interfaces e o princípio “uma interface, vários métodos” da programação orientada a objetos, será útil examinarmos um exemplo prático. Em capítulos anteriores, desenvolvemos uma classe chamada **SimpleStack** que implementava uma pilha de tamanho fixo para caracteres. No entanto, há outras maneiras de implementar uma pilha. Por exemplo, a pilha pode

Respostas:

1. Sim.
2. Falso.

ser dinâmica, ou seja, seu tamanho será expandido quando necessário para acomodar itens adicionais. Também podemos usar uma estrutura de dados que não seja um array para armazenar o conteúdo de uma pilha. Independentemente de como a pilha for implementada, sua interface permanecerá a mesma. Em outras palavras, métodos como **push()** e **pop()** a definirão sem importar os detalhes da implementação. Portanto, uma vez que tivermos criado uma interface de pilha, todas as pilhas que a implementarem poderão ser usadas da mesma forma. Além disso, elas poderão ser usadas por intermédio de uma referência de interface, o que nos permitirá alterar a implementação específica usada sem medo de danificar códigos existentes.

Neste projeto, você criará uma interface para uma pilha simples baseada na funcionalidade fornecida pela classe **SimpleStack** desenvolvida inicialmente na seção Tente isto 5-2 e melhorada nas seções Tente isto 6-1 e 6-2. Essa interface será chamada de **ISimpleStack**. Em seguida, duas implementações de **ISimpleStack** serão desenvolvidas. A primeiro é adaptada de **SimpleStack**. Já que é uma pilha de tamanho fixo, será chamada de **FixedLengthStack**. A segunda implementação, chamada **DynamicStack**, será uma pilha dinâmica, que cresce conforme necessário quando o tamanho do array subjacente é excedido.

PASSO A PASSO

1. Já que uma interface define a funcionalidade de uma implementação, a primeira etapa é criar a interface que descreve uma pilha simples. Para fazê-lo, começaremos com os métodos definidos pela classe **SimpleStack**. Lembre-se, ela declara os quatro métodos de pilha, **push()**, **pop()**, **isFull()** e **isEmpty**. Esses métodos são declarados na interface **ISimpleStack**, mostrada aqui. Insira a interface em um arquivo chamado **ISimpleStack.java**.

```
// Uma interface para uma pilha simples que armazena caracteres.
public interface ISimpleStack {

    // Insere um caractere na pilha.
    void push(char ch);

    // Remove um caractere da pilha.
    char pop();

    // Retorna true se a pilha estiver vazia.
    boolean isEmpty();

    // Retorna true se a pilha estiver cheia.
    boolean isFull();
}
```

Essa interface descreve as operações de uma pilha simples. Cada classe que implementar **ISimpleStack** terá que implementar esses métodos.

Uma observação interessante antes de avançarmos: nesse momento, **ISimpleStack** especifica a interface de uma pilha de caracteres. No capítulo 14, veremos como adaptar **ISimpleStack** para que especifique uma pilha contendo qualquer tipo de dado.

2. Agora você criará duas pilhas que implementam **ISimpleStack**. A primeira é adaptada da versão de **SimpleStack** mostrada na seção Tente isto 6-2. Como **SimpleStack** já implementa os métodos especificados por **ISimpleStack**, só três alterações são necessárias. Em primeiro lugar, o nome deve ser mudado para **FixedLengthStack**. Em segundo lugar, uma cláusula **implements ISimpleStack** deve ser adicionada à sua declaração. Em terceiro lugar, **push()**, **pop()** e **isEmpty()** têm que ser especificados como **public**, já que agora fornecem as implementações de **ISimpleStack**. Para ficar mais claro, mostraremos a classe **FixedLengthStack** inteira aqui. Insira-a em um arquivo chamado **FixedLengthStack.java**.

```
// Pilha de tamanho fixo para caracteres.
class FixedLengthStack implements ISimpleStack {
    private char[] data; // esse array contém a pilha
    private int tos; // índice do topo da pilha

    // Constrói uma pilha vazia dado seu tamanho.
    FixedLengthStack(int size) {
        data = new char[size]; // cria o array para armazenar a pilha
        tos = 0;
    }

    // Constrói uma pilha a partir de outra.
    FixedLengthStack(FixedLengthStack otherStack) {
        // o tamanho da nova pilha é igual ao de otherStack
        data = new char[otherStack.data.length];

        // configura tos com a mesma posição
        tos = otherStack.tos;

        // copia o conteúdo
        for(int i = 0; i < tos; i++)
            data[i] = otherStack.data[i];
    }

    // Constrói uma pilha com valores iniciais.
    FixedLengthStack(char[] chrs) {
        // cria o array para armazenar os valores iniciais
        data = new char[chrs.length];
        tos = 0;

        // inicializa a pilha inserindo nela
        // o conteúdo de chrs
        for(char ch : chrs)
            push(ch);
    }

    // Insere um caractere na pilha.
    public void push(char ch) {
```

```

        if(isFull()) {
            System.out.println(" -- Stack is full.");
            return;
        }

        data[tos] = ch;
        tos++;
    }

    // Remove um caractere da pilha.
    public char pop() {
        if(isEmpty()) {
            System.out.println(" -- Stack is empty.");
            return (char) 0; // valor de espaço reservado
        }

        tos--;
        return data[tos];
    }

    // Retorna true se a pilha estiver vazia.
    public boolean isEmpty() {
        return tos==0;
    }

    // Retorna true se a pilha estiver cheia.
    public boolean isFull() {
        return tos==data.length;
    }
}

```

3. Crie a classe **DynamicStack** mostrada a seguir. Ela implementa uma pilha “expansível” que aumenta seu tamanho quando acaba o espaço. Insira-a em um arquivo chamado **DynamicaStack.java**.

```

// Uma pilha expansível para caracteres.
class DynamicStack implements ISimpleStack {
    private char[] data; // esse array contém a pilha
    private int tos; // índice do topo da pilha

    // Constrói uma pilha vazia dado seu tamanho.
    DynamicStack(int size) {
        data = new char[size]; // cria o array para armazenar a pilha
        tos = 0;
    }

    // Constrói uma pilha a partir de outra.
    DynamicStack(DynamicStack otherStack) {
        // o tamanho da nova pilha é igual ao de otherStack
        data = new char[otherStack.data.length];
    }
}

```

```
// configura tos com a mesma posição
tos = otherStack.tos;

// copia o conteúdo
for(int i = 0; i < tos; i++)
    data[i] = otherStack.data[i];
}

// Constrói uma pilha com valores iniciais.
DynamicStack(char[] chrs) {
    // Cria o array para armazenar os valores iniciais
    data = new char[chrs.length];
    tos = 0;

    // inicializa a pilha inserindo nela
    // o conteúdo de chrs
    for(char ch : chrs)
        push(ch);
}

// Insere um caractere na pilha.
public void push(char ch) {

    // se não houver mais espaço no array,
    // expande o tamanho da pilha
    if(tos == data.length) {
        // dobra o tamanho do array existente
        char[] t = new char[data.length * 2];

        // copia o conteúdo da pilha no array maior
        for(int i = 0; i < tos; i++)
            t[i] = data[i];

        // configura data para referenciar o novo array
        data = t;
    }

    data[tos] = ch;
    tos++;
}

// Remove um caractere da pilha.
public char pop() {
    if(isEmpty()) {
        System.out.println("-- Stack is empty.");
        return (char) 0; // valor de espaço reservado
    }

    tos--;
}
```

```

        return data[tos];
    }
    // Retorna true se a pilha estiver cheia.
    public boolean isEmpty() {
        return tos==0;
    }

    // Retorna true se a pilha estiver vazia. Para DynamicStack,
    // esse método sempre retorna false.
    public boolean isFull() {
        return false;
    }
}

```

Nessa implementação, quando o limite do array **data** é alcançado, uma tentativa de armazenar outro elemento faz ser alocado um novo array duas vezes maior que o original. Em seguida, o conteúdo atual da pilha é copiado para o novo array. Para concluir, uma referência ao novo array é armazenada em **data**. Há outra coisa que devemos observar na implementação de pilha dinâmica: o método **isFull()** sempre retorna **false**. Já que o tamanho da pilha será aumentado automaticamente quando necessário, a pilha nunca estará cheia. (É claro que, em algum ponto em um caso extremo, a memória acabará, resultando em um erro de tempo de execução, mas o tratamento desse tipo de erro não faz parte da discussão.)

4. Para demonstrar as duas implementações de **ISimpleStack**, insira a classe a seguir em **ISimpleStackDemo.java**. Ela usa uma referência **ISimpleStack** para acessar as duas pilhas.

```

// Demonstra ISimpleStack.
class ISimpleStackDemo {
    public static void main(String[] args) {
        int i;
        char ch;

        // cria uma variável de interface ISimpleStack
        ISimpleStack iStack;

        // Agora, constrói um FixedLengthStack e um DynamicStack
        FixedLengthStack fixedStack = new FixedLengthStack(10);
        DynamicStack dynStack = new DynamicStack(5);

        // primeiro, usa fixedStack por intermédio de iStack
        iStack = fixedStack;

        // insere caracteres em fixedStack
        for(i = 0; !iStack.isFull(); i++)
            iStack.push((char) ('A'+i));

        // remove caracteres de fixedStack

```

```

        System.out.print("Contents of fixedStack: ");
        while(!iStack.isEmpty()) {
            ch = iStack.pop();
            System.out.print(ch);
        }

        System.out.println();

        // em seguida, usa dynStack por intermédio de iStack
        iStack = dynStack;

        // insere A até Z em dynStack
        // isso resultará em aumentar três vezes o seu tamanho
        for(i = 0; i < 26; i++)
            iStack.push((char) ('A'+i));

        // remove caracteres de dynStack
        System.out.print("Contents of dynStack: ");
        while(!iStack.isEmpty()) {
            ch = iStack.pop();
            System.out.print(ch);
        }
    }
}

```

5. Compile todos os arquivos e então execute **ISimpleStackDemo**. A saída é mostrada abaixo.

```

| Contents of fixedStack: JIHGFEDCBA
| Contents of dynStack: ZYXWVUTSRQPONMLKJIHGFEBCA

```

6. Vários outros métodos poderiam ser adicionados à interface **ISimpleStack** para melhorar a funcionalidade que ela especifica. Por exemplo, você poderia adicionar um método **reset()** que reinicializasse a pilha e um método **peek()** que obtivesse, sem remover, o elemento do topo da pilha. Um método **size()** que retornasse o número de elementos da pilha também seria um acréscimo útil. A implementação desses métodos é o assunto do Exercício 12 do fim deste capítulo.

CONSTANTES EM INTERFACES

Embora a principal finalidade de uma **interface** seja a declaração de métodos que forneçam uma fronteira bem definida para a funcionalidade, a interface também pode incluir “variáveis”. No entanto, essas “variáveis” não são variáveis de instância. Em vez disso, elas são implicitamente **public**, **static** e **final** e devem ser inicializadas. Logo, são basicamente constantes. À primeira vista, você poderia pensar que haveria uma aplicação muito limitada para essas variáveis, mas é o contrário. Normalmente, programas grandes fazem uso de diversos valores constantes que descrevem coisas como o tamanho do array, limites e valores especiais. Já que um programa grande

costuma usar várias classes separadas, é preciso haver uma maneira conveniente de disponibilizar essas constantes para cada classe. Em Java, as constantes de interface oferecem uma solução.

Para definir um conjunto de constantes compartilhadas, crie uma **interface** contendo apenas as constantes, sem nenhum método. Cada classe que precisar de acesso às constantes só precisará “implementar” a interface. Isso dará visibilidade às constantes. Aqui está um exemplo simples que lhe dará uma ideia de como o processo funciona:

```
// Uma interface que contém constantes.
interface IConst {
    int MIN = 0;
    int MAX = 10;
    String ERRORMSG = "Boundary Error";
}

// Ganha acesso às constantes implementando IConst.
class IConstDemo implements IConst {
    public static void main(String[] args) {
        int[] nums = new int[MAX];

        for(int i=MIN; i < (MAX + 1); i++) {
            if(i >= MAX) System.out.println(ERRORMSG);
            else {
                nums[i] = i;
                System.out.print(nums[i] + " ");
            }
        }
    }
}
```

Estas são constantes.

Nesse exemplo, a interface **IConst** define três constantes. **MIN** e **MAX** são de tipo **int** e **ERRORMSG** é de tipo **String**. Como requerido, elas receberam valores iniciais. A classe **IConstDemo** ganha acesso a essas constantes implementando **IConst**. Ou seja, as constantes podem ser usadas diretamente pela classe **IConstDemo**, como se tivessem sido definidas por ela. Já que **IConst** pode ser implementada por qualquer número de classes, pode ser usada por qualquer classe que precisar de acesso a suas constantes.

Pergunte ao especialista

P Uma interface deve realmente definir membros? Estou perguntando isso porque ao examinar a documentação de APIs Java, vi uma interface chamada **Cloneable**. Não me pareceu que ela tivesse membros. Pode explicar?

R Não é necessário que a interface defina membros. Esse tipo de interface costuma ser chamado de “interface marcadora” porque sua única finalidade é indicar que uma classe pode dar suporte a alguma ação. A interface marcadora passa a fazer parte do tipo da classe implementadora, mesmo que nada faça. Como você verá posteriormente neste livro, é possível o tipo de um objeto ser consultado no tempo de execução, e a presença da interface marcadora pode ser verificada. No caso de **Cloneable**, significa que uma cópia exata bit a bit de um objeto dessa classe é válida. À medida que você examinar a biblioteca de APIs Java, encontrará outros exemplos de interfaces marcadoras.

INTERFACES PODEM SER ESTENDIDAS

Uma interface pode herdar outra com o uso da palavra-chave **extends**. A sintaxe é a mesma da herança de classes. Quando uma classe implementa uma interface que herda outra interface, deve fornecer implementações de todos os métodos definidos dentro da cadeia de herança das interfaces. A seguir temos um exemplo:

```
// Uma interface pode estender outra.
interface A {
    void meth1();
    void meth2();
}

// B herda meth1() e meth2() e adiciona meth3().
interface B extends A { ←
    void meth3();
}                                B herda A.

// Esta classe deve implementar tudo que pertença a A e B.
class MyClass implements B {
    public void meth1() {
        System.out.println("Implement meth1().");
    }

    public void meth2() {
        System.out.println("Implement meth2().");
    }

    public void meth3() {
        System.out.println("Implement meth3().");
    }
}

class IFExtend {
    public static void main(String[] args) {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

Nesse exemplo, a interface **A** é estendida pela interface **B**. Em seguida, **MyClass** implementa **B**. Ou seja, **MyClass** deve implementar todos os métodos definidos pelas interfaces **A** e **B**, já que **A** foi herdada por **B**. Faça um teste e tente remover a implementação de **meth1()** em **MyClass**. Isso causará um erro de tempo de compilação. Lembre-se, qualquer classe que implemente uma interface deve implementar todos os métodos definidos por ela, inclusive os herdados de outras interfaces.

INTERFACES ANINHADAS

Uma interface pode ser declarada membro de outra interface ou de uma classe. Esse tipo de interface é chamado de *interface membro* ou *interface aninhada*. Uma interface aninhada em uma classe pode usar qualquer modificador de acesso. Uma interface aninhada em outra interface é implicitamente pública. Quando uma interface aninhada é usada fora do escopo em que se encontra, deve ser qualificada pelo nome da classe ou interface da qual é membro. Logo, fora da interface ou classe em que uma classe aninhada é declarada, seu nome deve ser totalmente qualificado.

Aqui está um exemplo que demonstra uma interface aninhada:

```
// Um exemplo de interface aninhada.

// Essa interface contém uma interface aninhada.
interface A {
    // esta é uma interface aninhada
    public interface NestedIF {
        boolean isNotNegative(int x);
    }

    void doSomething();
}

// Esta classe implementa a interface aninhada.
class B implements A.NestedIF {
    public boolean isNotNegative(int x) {
        return x < 0 ? false: true;
    }
}

class NestedIFDemo {
    public static void main(String[] args) {

        // usa uma referência de interface aninhada
        A.NestedIF nif = new B();

        if(nif.isNotNegative(10))
            System.out.println("10 is not negative");
        if(nif.isNotNegative(-12))
            System.out.println("this won't be displayed");
    }
}
```

A define uma interface membro chamada **NestedIF** que é declarada como **public**. Em seguida, B implementa a interface aninhada especificando

```
| implements A.NestedIF
```

Observe que o nome é totalmente qualificado pelo nome da interface externa. Dentro do método **main()**, uma referência **A.NestedIF** chamada **nif** é criada e recebe uma referência a um objeto **B**. Já que **B** implementa **A.NestedIF**, isso é válido.

Mais uma coisa: no programa, observe que **A** também especifica um método, chamado **doSomething()**. Como **B** só implementa a interface aninhada **NestedIF**, não precisa implementar **doSomething()**.

Verificação do progresso

1. Uma “variável” declarada em uma interface cria uma constante porque é implicitamente _____, _____ e _____.
2. Uma constante de interface deve ser inicializada?

CONSIDERAÇÕES FINAIS SOBRE AS INTERFACES

Embora os exemplos mostrados neste livro não façam uso frequente de interfaces, elas são parte importante da programação Java no mundo real. Além disso, diversas interfaces são encontradas na biblioteca Java e muitas das classes padrão implementam uma ou mais das interfaces padrão. Isso permite que várias funcionalidades sejam compartilhadas entre um grande conjunto de classes. Logo, é importante que você se acostume ao seu uso.

EXERCÍCIOS

1. “Uma interface, vários métodos” é um princípio-chave de Java. Que recurso o exemplifica melhor?
2. Quantas classes podem implementar uma interface?
3. Quantas interfaces uma classe pode implementar?
4. Uma classe declara que implementa uma interface com o uso de uma
_____.
5. As interfaces podem ser estendidas?
6. Crie uma interface para a classe **Vehicle** do Capítulo 7. Chame-a de **IVehicle**.
7. As variáveis declaradas em uma interface são implicitamente **static** e **final**. Para que servem?
8. Uma interface pode ser membro de outra?
9. Dadas duas interfaces chamadas **Alpha** e **Beta**, mostre como uma classe chamada **MyClass** especificaria que as implementa.
10. Crie uma nova classe **Constants** que implemente a interface **Series** discutida no começo deste capítulo. Seu método **getNext()** retorna repetidamente o último valor passado como argumento para o método **setStart()**. Até **setStart()** ser chamado, ele retorna 0.

Respostas:

1. **public, static e final**.
2. Sim.

- 11.** Considere a classe a seguir que alega implementar a interface **ISimpleStack** definida na seção Tente isto 8-1. Ela o faz? Por quê?

```
class MockStack implements ISimpleStack {
    public char pop() { return ' '; }
    public void push(char c) { }
    public boolean isEmpty() { return false; }
    public boolean isFull() { return false; }
}
```

- 12.** Adicione os métodos a seguir à interface **ISimpleStack** fornecida na seção Tente isto 8-1. Em seguida, implemente-os nas classes **FixedLengthStack** e **DynamicStack**.
- void reset(); // esvazia a pilha
 - char peek(); // como pop() mas o caractere permanece na pilha
 - int size(); // o número de caracteres atualmente na pilha
- 13.** Verdadeiro ou falso: uma classe que implemente uma interface deve
- usar os mesmos nomes de método usados pela interface.
 - usar os mesmos tipos de retorno de método usados pela interface.
 - usar os mesmos nomes de parâmetro usados pela interface.
 - tornar **public** todos os métodos especificados pela interface.
- 14.** Uma tomada elétrica padrão é extremamente versátil quando conseguimos ligar qualquer utensílio elétrico com um plugue padrão que se encaixe nela, não importando se o utensílio é, por exemplo, uma lâmpada, uma torradeira, um aparelho de ar-condicionado ou um computador e não importando a marca do utensílio. Uma tomada muito menos versátil exigiria um plugue especial usado apenas por um utensílio de uma marca específica. O que isso tem a ver com o assunto deste capítulo?
- 15.** Verdadeiro ou falso:
- Uma interface com um corpo vazio pode estender uma interface com um corpo não vazio.
 - Uma interface com um corpo não vazio pode estender uma interface com um corpo vazio.
- 16.** Suponhamos que uma classe **Class1** estendesse uma classe **Class2** e implementasse uma interface **Interface1** que estendesse uma interface **Interface2**. Suponhamos também que **Class1** tivesse um construtor sem argumentos. Quais das instruções a seguir são válidas?
- Class1 x = new Class1();
 - Class1 x = new Class1();
 - Interface1 x = new Class1();
 - Interface2 x = new Class1();
 - Object x = new Class1();

17. Verdadeiro ou falso: se **MyInterface** for uma interface e **x** e **y** forem variáveis declaradas com o tipo de **MyInterface**, então
- x** e **y** não podem ter o valor **null**.
 - se tanto **x** quanto **y** referenciarem objetos, esses objetos devem ser instâncias da mesma classe.
18. Na seção Tente isto 8-1, uma classe **FixedLengthStack** é criada e testada com o uso da classe **ISimpleStackDemo**. A declaração de classe de **FixedLengthStack** inclui uma cláusula **implements**:

```
| class FixedLengthStack implements ISimpleStack {
```

Suponhamos que a classe implementasse todos os métodos da interface **ISimpleStack**, mas a cláusula **implements** fosse omitida.

- A classe **FixedLengthStack** ainda seria compilada?
- A classe **ISimpleStackDemo** ainda seria compilada?

Pacotes

PRINCIPAIS HABILIDADES E CONCEITOS

- Saber a finalidade de um pacote
- Criar um pacote
- Entender como os pacotes afetam o acesso
- Aplicar o modificador de acesso **protected**
- Importar pacotes
- Importar pacotes padrão Java
- Usar a importação estática

Este capítulo examina outro recurso poderoso de Java: o pacote. Um *pacote* é um grupo de classes e interfaces relacionadas. Os pacotes ajudam a organizar o código e fornecem outra camada de encapsulamento. Com o uso de pacotes você ganhará um controle maior sobre a organização do programa.

ASPECTOS BÁSICOS DOS PACOTES

Em programação, com frequência é útil agrupar partes relacionadas de um programa. Em Java, isso é feito com o uso de um pacote. O pacote serve a duas finalidades. Em primeiro lugar, fornece um mecanismo pelo qual partes relacionadas de um programa podem ser organizadas como uma unidade. Por exemplo, se você estiver organizando um sistema de entrada de pedidos para uma loja online, provavelmente vai querer criar um pacote contendo essas classes e interfaces. As classes definidas dentro de um pacote devem ser acessadas com o uso do nome de seu pacote. Logo, um pacote fornece uma maneira de nomear um conjunto de classes. Em segundo lugar, o pacote participa do mecanismo de controle de acesso Java. Classes definidas dentro de um pacote podem se tornar privadas desse pacote sem poder ser acessadas por códigos de fora dele. Portanto, o pacote fornece um meio pelo qual classes podem ser encapsuladas. Examinemos cada recurso um pouco mais detalhadamente.

Em geral, quando nomeamos uma classe, estamos alocando um nome do *espaço de nome*. Um espaço de nome define uma região declarativa. Em Java, duas classes não podem usar nomes iguais do mesmo espaço de nome. Logo, dentro de um deter-

minado espaço de nome, o nome de cada classe deve ser exclusivo. Todos os exemplos mostrados nos capítulos anteriores usaram o espaço de nome padrão ou global. Embora isso seja adequado para exemplos de programa curtos, torna-se um problema à medida que os programas crescem e o espaço de nome padrão fica abarrotado. Em programas grandes, encontrar nomes exclusivos para cada classe pode ser difícil. Também devemos evitar que os nomes colidam com os existentes em códigos criados por outros programadores que trabalhem no mesmo projeto e com os da biblioteca Java. A solução para esses problemas é o pacote, porque ele fornece uma maneira de dividir o espaço de nome. Quando uma classe é definida dentro de um pacote, o nome desse pacote é anexado ao da classe, o que evita que os nomes colidam com os de outras classes com o mesmo nome, mas de outros pacotes.

Como geralmente um pacote contém classes relacionadas, Java define direitos de acesso especiais para os códigos do pacote. Em um pacote, podemos definir um código acessado por outro código do mesmo pacote, mas não por código de fora do pacote. Isso nos permite criar grupos autônomos de classes relacionadas que mantêm sua operação privada.

Definindo um pacote

Todas as classes em Java pertencem a algum pacote. Quando um pacote não é especificado explicitamente, o pacote padrão (ou global) é usado. O pacote padrão não tem nome, o que o torna transparente. Por isso, até agora você não precisou se preocupar com os pacotes. Embora o pacote padrão seja adequado para exemplos de programa curtos, não é apropriado para aplicativos reais. Quase sempre, você definirá um ou mais pacotes para seu código.

Para criar um pacote, você usará a instrução **package**, que fica no início de um arquivo-fonte Java. Uma classe declarada dentro desse arquivo pertencerá ao pacote especificado. Uma vez que um pacote define um espaço de nome, o nome de uma classe que você inserir no pacote fará parte do espaço de nome desse pacote.

Esta é a forma geral da instrução **package**:

```
package pct;
```

Aqui, *pct* é o nome do pacote. Por exemplo, a instrução a seguir cria um pacote chamado **mypad**:

```
| package mypack;
```

Java usa o sistema de arquivos para gerenciar pacotes, com cada pacote sendo armazenado em seu próprio diretório. Por exemplo, os arquivos **.class** de qualquer classe que você declarar como parte de **mypad** devem ser armazenados em um diretório chamado **mypad**.

Como no resto em Java, há a diferenciação entre minúsculas e maiúsculas nos nomes dos pacotes. Ou seja, o diretório em que um pacote é armazenado deve ter nome exatamente igual ao do pacote. Se você tiver problemas ao testar os exemplos deste capítulo, lembre-se de verificar com cuidado os nomes de pacotes e diretórios. Geralmente são usadas minúsculas nos nomes de pacotes.

Mais de um arquivo pode incluir a mesma instrução **package**. Essa instrução especifica apenas a que pacote pertence o arquivo. Ela não impede que classes de

outros arquivos façam parte do mesmo pacote. A maioria dos pacotes do mundo real se estende por muitos arquivos.

Você pode criar uma hierarquia de pacotes. Para fazê-lo, apenas separe cada nome de pacote do nome que fica acima dele usando um ponto. A forma geral de uma instrução de pacote de vários níveis é mostrada aqui:

```
package pacote1.pacote2.pacote3...pacoteN;
```

Obviamente, você deve criar diretórios que deem suporte à hierarquia de pacotes existente. Por exemplo, a hierarquia

```
| package alpha.beta.gamma;
```

deve ser armazenada em .../alpha/beta/gamma, onde ... indica o caminho dos diretórios especificados.

Encontrando pacotes e CLASSPATH

Como acabamos de explicar, os pacotes são espelhados pelos diretórios. Isso levanta uma questão importante: como o sistema de tempo de execução Java saberá onde procurar os pacotes que você criar? A resposta tem três partes. Em primeiro lugar, por padrão, o sistema de tempo de execução usa o diretório de trabalho atual como seu ponto de partida. Logo, se seu pacote estiver em um subdiretório do diretório atual, ele será encontrado. Em segundo lugar, você pode especificar um caminho ou caminhos de diretório configurando a variável de ambiente **CLASSPATH**. Em terceiro lugar, você pode usar a opção **-classpath** com **java** e **javac** para especificar o caminho de suas classes.

Por exemplo, consideremos a especificação de pacote a seguir:

```
| package mypack;
```

Para um programa encontrar **mypadck**, uma entre três coisas deve ser verdadeira: o programa deve poder ser executado a partir de um diretório imediatamente acima de **mypadck**, ou **CLASSPATH** deve ser configurada para incluir o caminho de **mypadck**, ou a opção **-classpath** deve especificar o caminho de **mypadck** quando o programa for executado via **java**.

A maneira mais fácil de testar os exemplos mostrados neste livro é criando os diretórios dos pacotes abaixo de seu diretório de desenvolvimento atual, inserindo os arquivos **.class** nos diretórios apropriados e então executando os programas a partir do diretório de desenvolvimento. Essa é a abordagem usada nos próximos exemplos.

Um último ponto: para evitar problemas, é melhor manter todos os arquivos **.java** e **.class** associados a um pacote no diretório desse pacote. Além disso, compile cada arquivo a partir do diretório acima do diretório do pacote.

Exemplo breve de pacote

Lembrando da discussão anterior, teste este exemplo curto de pacote. Ele cria um banco de dados de livros simples que fica contido dentro de um pacote chamado **bookpack**.

```
// Demonstração breve dos pacotes.
package bookpack; ← Este arquivo faz parte do pacote bookpack.

class Book { ← Logo, Book faz parte de bookpack.
    private String title;
```

```
private String author;
private int pubDate;
Book(String t, String a, int d) {
    title = t;
    author = a;
    pubDate = d;
}

void show() {
    System.out.println(title);
    System.out.println(author);
    System.out.println(pubDate);
}
}

class BookDemo { ←———— BookDemo também faz parte de bookpack.
    public static void main(String[] args) {
        Book[] books = new Book[5];

        books[0] = new Book("The Art of Computer Programming, Vol 3",
                            "Knuth", 1973);
        books[1] = new Book("Moby Dick",
                            "Melville", 1851);
        books[2] = new Book("Thirteen at Dinner",
                            "Christie", 1933);
        books[3] = new Book("Red Storm Rising",
                            "Clancy", 1986);
        books[4] = new Book("On the Road",
                            "Kerouac", 1955);

        for(int i=0; i < books.length; i++) {
            books[i].show();
            System.out.println();
        }
    }
}
```

Chame esse arquivo de **BookDemo.java** e insira-o em um diretório chamado **bookpack**.

Em seguida, compile o arquivo. Você pode fazer isso especificando

```
| javac bookpack/BookDemo.java
```

a partir do diretório imediatamente acima de **bookpack**. Agora, tente executar a classe usando a linha de comando a seguir:

```
| java bookpack.BookDemo
```

Lembre-se, você tem que estar no diretório acima de **bookpack** quando executar esse comando. (Ou use uma das duas outras opções descritas na seção anterior para especificar o caminho de **bookpack**.)

Como explicado, agora **BookDemo** e **Book** fazem parte do pacote **bookpack**. Ou seja, **BookDemo** não pode ser executada separadamente, portanto, você não pode usar a seguinte linha de comando:

```
| java BookDemo
```

Em vez disso, **BookDemo** deve ser qualificada com o nome de seu pacote.

Verificação do progresso

1. O que é um pacote?
2. Mostre como declarar um pacote chamado **tools**.
3. O que é **CLASSPATH**?

PACOTES E O ACESSO A MEMBROS

Os capítulos anteriores introduziram os aspectos básicos do controle de acesso, inclusive os modificadores **private** e **public**, mas não contaram a história toda. Isso ocorreu porque os pacotes também participam do mecanismo de controle de acesso Java e uma discussão completa tinha que esperar até eles serem abordados.

A visibilidade de um elemento é determinada por sua especificação de acesso – **private**, **public**, **protected** ou padrão – e pelo pacote em que ele reside. Logo, a visibilidade de um elemento é determinada por sua visibilidade dentro de uma classe e sua visibilidade dentro de um pacote. Essa abordagem do controle de acesso em várias camadas dá suporte a um rico conjunto de privilégios de acesso. A Tabela 9-1 resume os vários níveis de acesso. Examinaremos cada opção individualmente.

Se o membro de uma classe não tiver um modificador de acesso explícito, poderá ser visto dentro de seu pacote, mas não fora dele. Portanto, você usará a especificação de acesso padrão para elementos que quiser manter privados para o pacote, mas públicos dentro dele.

Membros declarados explicitamente como **public** podem ser vistos em todos os locais, inclusive classes e pacotes diferentes. Não há restrição quanto ao seu uso ou acesso. Um membro **private** só pode ser acessado por outros membros de sua classe. Ele não é afetado por sua associação a um pacote. Um membro especificado como **protected** pode ser acessado dentro de seu pacote e por todas as subclasses, inclusive subclasses de outros pacotes.

A Tabela 9-1 só se aplica a membros de classes. Uma classe de nível superior tem apenas dois níveis de acesso possíveis: padrão e público. Quando uma classe é declarada como **public**, pode ser acessada por qualquer código. Se a classe tiver acesso padrão, só poderá ser acessada por um código do mesmo pacote. O mesmo ocorre com as interfaces. Além disso, a classe ou interface declarada como **public** deve residir em um arquivo de mesmo nome.

Respostas:

1. Um pacote é um contêiner para classes. Ele desempenha um papel organizador e de encapsulamento.
2. package tools;
3. **CLASSPATH** é a variável de ambiente que especifica o caminho das classes.

Tabela 9-1 Acesso a membros de classe

	Membro privado	Membro padrão	Membro protegido	Membro público
Visível dentro da mesma classe	Sim	Sim	Sim	Sim
Visível dentro do mesmo pacote pela subclasse	Não	Sim	Sim	Sim
Visível dentro do mesmo pacote por não subclasses	Não	Sim	Sim	Sim
Visível dentro de pacote diferente pela subclasse	Não	Não	Sim	Sim
Visível dentro de pacote diferente por não subclasses	Não	Não	Não	Sim

Verificação do progresso

1. Se o membro de uma classe tiver acesso padrão dentro de um pacote, poderá ser acessado por outros pacotes?
2. O que **protected** faz?
3. Um membro **private** pode ser acessado por subclasses dentro de seus pacotes. Verdadeiro ou falso?

Exemplo de acesso a pacote

No exemplo de **package** mostrado anteriormente, tanto **Book** quanto **BookDemo** estavam no mesmo pacote, logo, não havia problema em **BookDemo** usar **Book**, porque o privilégio de acesso padrão concede acesso a todos os membros do mesmo pacote. No entanto, se **Book** estivesse em um pacote e **BookDemo** em outro, a situação seria diferente. Nesse caso, o acesso a **Book** seria negado. Para disponibilizar **Book** para outros pacotes, você deve fazer três alterações. Em primeiro lugar, **Book** deve ser declarada como **public**. Isso a tornará visível fora de **bookpack**. Em segundo lugar, seu construtor deve ser tornado **public**, e para concluir, seu método **show()** tem que ser **public**. Isso permitirá que eles também possam ser vistos fora

Respostas:

1. Não.
2. Permite que um membro seja acessado por outros códigos de seu pacote e por todas as subclasses, não importando em que pacote elas estejam.
3. Falso.

de **bookpack**. Portanto, para **Book** ser usada por outros pacotes, deve ser recodificada como mostrado aqui.

```
// Book recodificada para acesso público.
package bookpack;

public class Book { ← Book e seus membros devem ser public
    private String title;      para serem usados por outros pacotes.
    private String author;
    private int pubDate;

    // Agora é público.
    public Book(String t, String a, int d) {
        title = t;
        author = a;
        pubDate = d;
    }

    // Agora é público.
    public void show() {
        System.out.println(title);
        System.out.println(author);
        System.out.println(pubDate);
    }
}
```

Para usar **Book** a partir de outro pacote, você deve雇用 a instrução **import** descrita posteriormente neste capítulo ou qualificar totalmente seu nome para que inclua a especificação de pacote completa. Por exemplo, esta é uma classe chamada **UseBook**, que está contida em um pacote diferente, chamado **mypack**. Ela qualifica **Book** totalmente para usá-la.

```
// Esta classe está no pacote mypack.
package mypack;

// Usa a classe Book a partir de bookpack.
class UseBook { ← Qualifica Book com o nome
    public static void main(String[] args) {
        bookpack.Book[] books = new bookpack.Book[5]; ← de seu pacote: bookpack.

        books[0] = new bookpack.Book("The Art of Computer Programming, Vol 3",
                                     "Knuth", 1973);
        books[1] = new bookpack.Book("Moby Dick",
                                     "Melville", 1851);
        books[2] = new bookpack.Book("Thirteen at Dinner",
                                     "Christie", 1933);
        books[3] = new bookpack.Book("Red Storm Rising",
                                     "Clancy", 1986);
        books[4] = new bookpack.Book("On the Road",
                                     "Kerouac", 1955);

        for(int i=0; i < books.length; i++) {
```

```
        books[i].show();
        System.out.println();
    }
}
```

Observe como cada uso de **Book** é precedido pelo qualificador **bookpack**. Sem essa especificação, **Book** não seria encontrada quando você tentasse compilar **UseBook**.

Entendendo os membros protegidos

Às vezes, iniciantes em Java ficam confusos com o significado e o uso de **protected**. Como explicado, o modificador **protected** cria um membro que pode ser acessado dentro de seu pacote e por subclasses de outros pacotes. Logo, um membro **protected** fica disponível para ser usado por todas as subclasses, mas continua protegido contra o acesso arbitrário de códigos de fora de seu pacote.

Para entender melhor os efeitos de **protected**, usemos um exemplo. Primeiro, altere a classe **Book** para que suas variáveis de instância sejam **protected**, como mostrado abaixo.

```
// Torna as variáveis de instância de Book protegidas.  
package bookpack;  
  
public class Book {  
    // agora essas variáveis são protected  
    protected String title; ← Agora são protected.  
    protected String author; ← Agora são protected.  
    protected int pubDate; ← Agora são protected.  
  
    public Book(String t, String a, int d) {  
        title = t;  
        author = a;  
        pubDate = d;  
    }  
  
    public void show() {  
        System.out.println(title);  
        System.out.println(author);  
        System.out.println(pubDate);  
    }  
}
```

Em seguida, crie uma subclasse de `Book`, chamada `ExtBook`, e uma classe chamada `ProtectDemo` que use `ExtBook`. `ExtBook` adiciona um campo que armazena a condição do livro e vários métodos acessadores. Essas duas classes ficam em seu próprio pacote chamado `bookpackext`. Elas são mostradas aqui.

```
// Demonstra protected.  
package bookpackext;  
  
class ExtBook extends bookpack.Book {  
    private String condition;
```

```

public ExtBook(String t, String a, int d, String c) {
    super(t, a, d);
    condition = c;
}

public void show() {
    super.show();
    System.out.print("Condition is " + condition);
    System.out.println();
}

public String getCondition() { return condition; }
public void setCondition(String c) { condition = c; }

/* Estas instruções estão corretas porque subclasses podem acessar
   um membro protegido. */
public String getTitle() { return title; }
public void setTitle(String t) { title = t; }
public String getAuthor() { return author; }
public void setAuthor(String a) { author = a; }
public int getPubDate() { return pubDate; }
public void setPubDate(int d) { pubDate = d; }
}

class ProtectDemo {
    public static void main(String[] args) {
        ExtBook[] books = new ExtBook[5];

        books[0] = new ExtBook("The Art of Computer Programming, Vol 3",
                              "Knuth", 1973, "well used");
        books[1] = new ExtBook("Moby Dick",
                              "Melville", 1851, "like new");
        books[2] = new ExtBook("Thirteen at Dinner",
                              "Christie", 1933, "fair");
        books[3] = new ExtBook("Red Storm Rising",
                              "Clancy", 1986, "good");
        books[4] = new ExtBook("On the Road",
                              "Kerouac", 1955, "fair");

        for(int i=0; i < books.length; i++) {
            books[i].show();
            System.out.println();
        }

        // Encontra a condição de Moby Dick.
        System.out.print("Condition of Moby Dick is ");
        for(int i=0; i < books.length; i++)
            if(books[i].getTitle() == "Moby Dick")
                System.out.println(books[i].getCondition());
    }
}

```

O acesso a membros de **Book** é permitido a subclasses.

```
//      books[0].title = "test title"; // Erro - não pode ser acessado
}
}           ↑
                  O acesso a um campo protected não é permitido a não subclasses.
```

Veja primeiro o código de **ExtBook**. Como **ExtBook** estende **Book**, ela tem acesso aos membros **protected** de **Book** mesmo estando em um pacote diferente. Logo, pode acessar **title**, **author** e **pubDate** diretamente, como faz nos métodos acessadores que cria para essas variáveis. No entanto, em **ProtectDemo**, o acesso às variáveis é negado, porque **ProtectDemo** não é subclasse de **Book**. Por exemplo, se você remover o símbolo de comentário da linha a seguir, o programa não será compilado.

```
//      books[0].title = "test title"; // Erro - não pode ser acessado
```

Pergunte ao especialista

P Há alguma restrição à maneira como uma subclasse pode acessar um membro **protected**?

R Como o exemplo **ProtectDemo** mostra, uma subclasse de um pacote diferente tem acesso a um membro **protected** de sua superclasse para estender a superclasse. No entanto, ela *não* pode acessar um membro **protected** de sua superclasse por intermédio de um objeto dessa superclasse. Por exemplo, se o método a seguir for adicionado a **ExtBook**,

```
void wontWork() {
    bookpack.Book b = new bookpack.Book("sometitle", "someauthor", 1961);
    b.title = "newtitle"; // Erro!
}
```

a tentativa de acessar **title** por intermédio de **b** falhará. Se você pensar bem, essa restrição faz sentido. Uma subclasse pode usar um membro **protected** para ser implementada, mas não como um meio de burlar a limitação de acesso **protected**.

IMPORTANDO PACOTES

Ao usar uma classe de outro pacote, você pode qualificar totalmente o nome da classe com o nome do pacote, como fizeram os exemplos anteriores. No entanto, essa abordagem pode se tornar cansativa e incômoda, principalmente se as classes qualificadas estiverem aninhadas em um nível muito profundo de uma hierarquia de pacotes. Como Java foi inventada por programadores para programadores – e programadores não gostam de estruturas entediantes –, não deve surpreender o fato de existir um método mais conveniente para o uso do conteúdo de pacotes: a instrução **import**. Usando **import** você pode dar visibilidade a um ou mais membros de um pacote. Isso lhe permitirá usar esses membros diretamente, sem uma qualificação de pacote explícita.

Esta é a forma geral da instrução **import**:

```
import pct.nomeclasse;
```

Aqui, *pct* é o nome do pacote, que pode incluir seu caminho completo, e *nomeclasse* é o nome da classe que está sendo importada. Se quiser importar o conteúdo inteiro

de um pacote, use um asterisco (*) como nome da classe. Veja exemplos das duas formas:

```
| import mypack.MyClass;
| import mypack.*;
```

No primeiro caso, a classe **MyClass** é importada de **mypack**. No segundo, todas as classes de **mypack** são importadas. Em um arquivo-fonte Java, as instruções **import** ocorrem imediatamente após a instrução **package** (se ela existir) e antes de qualquer definição de classe.

Você pode usar **import** para dar visibilidade ao pacote **bookpack** e a classe **Book** poder ser usada sem qualificação. Para fazê-lo, simplesmente adicione esta instrução **import** ao início de qualquer arquivo que use **Book**.

```
| import bookpack.*;
```

Por exemplo, aqui está a classe **UseBook** recodificada para usar **import**:

```
// Demonstra import.
package mypack;
import bookpack.*; ← Importa bookpack.

// Usa a classe Book a partir de bookpack.
class UseBook {
    public static void main(String[] args) {
        Book[] books = new Book[5]; ← Agora, você pode referenciar Book
                                     diretamente, sem qualificação.
        books[0] = new Book("The Art of Computer Programming, Vol 3",
                           "Knuth", 1973);
        books[1] = new Book("Moby Dick",
                           "Melville", 1851);
        books[2] = new Book("Thirteen at Dinner",
                           "Christie", 1933);
        books[3] = new Book("Red Storm Rising",
                           "Clancy", 1986);
        books[4] = new Book("On the Road",
                           "Kerouac", 1955);

        for(int i=0; i < books.length; i++) {
            books[i].show();
            System.out.println();
        }
    }
}
```

Observe que você não precisa mais qualificar **Book** com o nome do pacote.

Importando pacotes Java padrão

Como explicado anteriormente neste livro, Java define várias classes padrão que estão disponíveis para todos os programas. Essa biblioteca de classes costuma ser chamada de API (Application Programming Interface) Java. A API Java fica armazenada em

pacotes. No topo da hierarquia de pacotes está o pacote **java**. Há vários subpacotes que descendem do pacote **java**, entre eles:

Subpacote	Descrição
java.lang	Contém várias classes de uso geral
java.io	Contém as classes de I/O
java.net	Contém as classes que dão suporte à rede
java.applet	Contém classes de criação de applets
java.awt	Contém classes que dão suporte ao Abstract Window Toolkit
java.util	Contém várias classes utilitárias, mais o Collections Framework

Desde o começo deste livro, temos usado o pacote **java.lang**. Ele contém, entre muitas outras, a classe **System**, que usamos na exibição de saídas por meio de **println()**. O pacote **java.lang** é único, porque é importado automaticamente para cada programa Java. É por isso que não tivemos que importar **java.lang** nos exemplos de programa anteriores.

No entanto, devemos importar explicitamente os outros pacotes da API. Os pacotes padrão são importados da mesma forma que os mostrados nos exemplos anteriores. Por exemplo, para importar todo o pacote **java.net**, use a instrução a seguir:

```
| import java.net.*;
```

Posteriormente neste livro, vários pacotes da API Java serão examinados. Como você verá, a API oferece um vasto conjunto de funcionalidades predefinidas que seu programa poderá acessar, simplesmente importando o pacote relevante.

Verificação do progresso

1. Como podemos incluir outro pacote em um arquivo-fonte?
2. Mostre como incluir todas as classes de um pacote chamado **toolpack**.
3. É preciso incluir o pacote **java.lang** explicitamente?

TENTE ISTO 9-1 Movendo uma classe para outro pacote

```
Dog.java  
Owner.java  
DogOwnerDemo.java
```

Este projeto mostra o que envolve a transferência de uma classe para outro pacote. Como você deve ter imaginado, requer mais do que apenas alterar a instrução de pacote no começo do arquivo que contém a classe. Usaremos três classes no projeto: **Owner**, **Dog** e **DogOwnerDemo**. Inicialmente, as classes **Owner** e **Dog**

Respostas:

1. Usando a instrução **import**.
2. `import toolpack.*;`
3. Não.

estarão em um pacote **owner** e a classe **DogOwnerDemo** estará em seu próprio pacote. Então, moveremos a classe **Dog** para o novo pacote **dog**.

PASSO A PASSO

1. Em seu diretório de trabalho atual, crie os três diretórios a seguir: **owner**, **dogownerdemo** e **dog**.
2. No diretório **owner**, crie um arquivo chamado **Dog.java**. Em **Dog.java**, adicione o código abaixo:

```
package owner;

public class Dog {
    String name;

    public Dog(String n) { name = n; }

    public String toString() {
        return name;
    }
}
```

3. No diretório **owner**, crie um arquivo chamado **Owner.java**. Em **Owner.java**, adicione o código abaixo:

```
package owner;

public class Owner {
    String name;
    Dog dog;

    public Owner(String n, Dog d) {
        name = n; dog = d;
    }

    public String toString() {
        return name + " owns " + dog;
    }
}
```

4. No diretório **dogownerdemo**, crie um arquivo chamado **DogOwnerDemo.java**. Em **DogOwnerDemo.java**, adicione o programa a seguir:

```
package dogownerdemo;

import owner.*;

class DogOwnerDemo {
    public static void main(String[] args) {
        Owner owner = new Owner("Fred", new Dog("Sam"));
        System.out.println(owner);
    }
}
```

5. Compile todos os arquivos do diretório de trabalho atual usando esta sequência:

```
| javac owner/Dog.java
| javac owner/Owner.java
| javac dogownerdemo/DogOwnerDemo.java
```

Em seguida, use esta linha

```
| java dogownerdemo.DogOwnerDemo
```

para executar a classe **DogOwnerDemo** e verificar se tudo está funcionando apropriadamente. Você deve ver a saída abaixo.

```
| Fred owns Sam
```

6. Agora, moveremos a classe **Dog** para o novo pacote **dog**. São três etapas:
 A. Alterar a primeira linha de **Dog.java** para que apresente **package dog;** em vez de **package owner;**
 B. Mover **Dog.java** para o diretório chamado **dog**. (Certifique-se de remover do diretório **owner** tanto o arquivo-fonte quanto o arquivo de classe.)
 C. Adicione a linha **import dog.*;** aos dois outros arquivos, já que ambos usam a classe **Dog**.
7. Compile todos os arquivos como acabamos de mostrar, exceto que, agora, **Dog.java** é compilado com o uso desta linha:

```
| javac dog/Dog.java
```

Em seguida, execute novamente **DogOwnerDemo** como mostrado. Você deve obter a mesma saída.

8. É importante entender que foi relativamente simples mover **Dog** de um pacote para outro devido a vários fatores. Em primeiro lugar, a instrução **import** facilitou que outras classes continuassem acessando a classe **Dog**, mesmo depois que ela foi movida para outro pacote – só tivemos que adicionar uma nova linha de código a essas classes. Em segundo lugar, o fato de a classe **Dog** e seu método e o construtor serem **public** possibilitou que a classe fosse movida e continuasse sendo acessada por outras classes. Porém, se outras classes do pacote **owner** acessavam diretamente a variável de instância **name** de **Dog**, que não é **public**, após a transferência elas não poderão mais fazê-lo. Logo, uma alteração em seus códigos seria necessária. Da mesma forma, se a classe **Dog** acessava algum membro não público de uma classe do pacote **owner**, não poderá mais fazer isso após a transferência e, portanto, seu código também precisaria ser alterado. Resumindo, mover uma classe para um novo pacote pode trazer vários problemas. A chave para evitá-los é passar algum tempo projetando com cuidado suas classes e pacotes para reduzir quantas vezes um pacote terá que ser alterado.

IMPORTAÇÃO ESTÁTICA

Java dá suporte a um uso expandido da palavra-chave **import**. Se colocarmos a palavra-chave **static** depois de **import**, uma instrução **import** poderá ser usada para importar os membros estáticos de uma classe ou interface. Isso se chama *importação estática* e foi adicionado a Java pelo JDK 5. Quando a importação estática é usada, podemos referenciar membros estáticos diretamente por seus nomes, sem a necessidade de qualificá-los com o nome de sua classe. Esse método simplifica e encurta a sintaxe necessária ao uso de um membro estático.

Para entender a utilidade da importação estática, começemos com um exemplo que *não* a usa. O programa a seguir calcula as soluções de uma equação quadrática, que tem esta forma:

$$ax^2 + bx + c = 0$$

O programa usa dois métodos estáticos da classe interna Java **Math** de cálculos matemáticos, que faz parte de **java.lang**. O primeiro é **Math.pow()**, que retorna um valor elevado a uma potência especificada. O segundo é **Math.sqrt()**, que retorna a raiz quadrada de seu argumento.

```
// Encontra as soluções de uma equação quadrática.
class Quadratic {
    public static void main(String[] args) {

        // a, b e c representam os coeficientes
        // da equação quadrática: ax2 + bx + c = 0
        double a, b, c, x;

        // Resolve 4x2 + x - 3 = 0 para achar x.
        a = 4;
        b = 1;
        c = -3;

        // Encontra a primeira solução.
        x = (-b + Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
        System.out.println("First solution: " + x);

        // Encontra a segunda solução.
        x = (-b - Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
        System.out.println("Second solution: " + x);
    }
}
```

Já que **pow()** e **sqrt()** são métodos estáticos, devem ser chamados com o uso do nome de sua classe, **Math**, o que resulta em uma expressão um pouco confusa:

```
| x = (-b + Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
```

Além disso, pode ser tedioso ter de especificar o nome da classe sempre que **pow()** ou **sqrt()** (ou qualquer um dos outros métodos matemáticos Java, como **sin()**, **cos()** e **tan()**) for usado.

Você pode eliminar o incômodo de especificar o nome da classe usando a importação estática, como mostrado na versão a seguir do programa anterior:

```
// Usa a importação estática para tornar sqrt() e pow() visíveis.  
import static java.lang.Math.sqrt; ← Usa a importação estática para tornar  
import static java.lang.Math.pow; ← sqrt() e pow() visíveis.  
  
class Quadratic {  
    public static void main(String[] args) {  
  
        // a, b e c representam os coeficientes  
        // da equação quadrática: ax2 + bx + c = 0  
        double a, b, c, x;  
  
        // Resolve 4x2 + x - 3 = 0 para achar x.  
        a = 4;  
        b = 1;  
        c = -3;  
  
        // Encontra a primeira solução.  
        x = (-b + sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);  
        System.out.println("First solution: " + x);  
  
        // Encontra a segunda solução.  
        x = (-b - sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);  
        System.out.println("Second solution: " + x);  
    }  
}
```

Nessa versão, os nomes **sqrt** e **pow** ganham visibilidade por intermédio das instruções de importação estática abaixo:

```
import static java.lang.Math.sqrt;  
import static java.lang.Math.pow;
```

Depois das instruções, não é mais necessário qualificar **sqrt()** e **pow()** com o nome de sua classe. Logo, a expressão pode ser especificada de maneira mais conveniente, como mostrado aqui:

```
| x = (-b + sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
```

Como você pode ver, essa forma é consideravelmente menor e mais fácil de ler.

Há duas formas gerais da instrução **import static**. A primeira, que é usada pelo exemplo anterior, torna visível um único nome. Sua forma geral é mostrada abaixo:

```
import static pct.nome-tipo.nome-membro-estático;
```

Aqui, *nome-tipo* é o nome da classe ou interface que contém o membro estático desejado. O nome completo do pacote é especificado por *pct*. O nome do membro é especificado por *nome-membro-estático*.

O segundo tipo de importação estática importa todos os membros estáticos. Sua forma geral é mostrada abaixo:

```
import static pct.nome-tipo.*;
```

Se você utilizar muitos campos ou métodos estáticos definidos por uma classe, essa forma lhe permitirá torná-los visíveis sem ser preciso especificar cada um individualmente. Logo, o programa anterior poderia ter usado apenas essa instrução **import** para dar visibilidade tanto a **pow()** quanto a **sqrt()** (e a *todos os outros* membros estáticos de **Math**):

```
| import static java.lang.Math.*;
```

É claro que, o uso da importação estática não está restrito apenas à classe **Math** ou aos métodos. Por exemplo, esta instrução dá visibilidade ao campo estático **System.out**:

```
| import static java.lang.System.out;
```

Depois dessa instrução, você pode exibir a saída no console sem que seja necessário qualificar **out** com **System**, como mostrado aqui:

```
| out.println("After importing System.out, you can use out directly.");
```

Se a importação de **System.out** como acabamos de mostrar é uma boa ideia, é algo que se presta a debate. Embora encurte a instrução, não está mais imediatamente claro para alguém que leia o programa que o **out** que está sendo referenciado é **System.out**.

A importação estática pode ser conveniente, mas é importante não usá-la de maneira abusiva. Lembre-se, uma razão para Java organizar suas bibliotecas em pacotes é evitar colisões de espaço de nome. Quando você importar membros estáticos, estará trazendo-os para o espaço de nome global. Logo, estará aumentando a possibilidade de ocorrência de conflitos de espaço de nome e a inadvertida ocultação de outros nomes. Se estiver usando um membro estático uma ou duas vezes no programa, é melhor não importá-lo. Além disso, alguns nomes estáticos, como **System.out**, são tão conhecidos que talvez seja preferível não importá-los. A importação estática foi projetada para situações em que você esteja usando um membro estático repetidamente, como na execução de uma série de cálculos matemáticos. Em resumo, você deve usar esse recurso, mas sem abusar.

Pergunte ao especialista

P A importação estática é para ser usada apenas com classes da biblioteca Java ou posso usá-la com as classes que eu criar?

R Você pode utilizar a importação estática para importar os membros estáticos das classes e interfaces que criar. Isso será particularmente conveniente quando definir vários membros estáticos usados com frequência em todo um programa grande. Por exemplo, se uma classe definir várias constantes **static final** para estabelecer limites, o uso da importação estática para lhes dar visibilidade evitaria muita digitação tediosa. No entanto, o mesmo cuidado tomado anteriormente é aplicável: use esse recurso, mas não abuse.

EXERCÍCIOS

1. Usando o código da seção Tente isto 8-1, insira a interface **ISimpleStack** e suas duas implementações em um pacote chamado **stackpack**. Mantendo a classe de demonstração de pilha **ISimpleStackDemo** no pacote padrão, mostre como importar e usar as classes de **stackpack**.
2. O que é espaço de nome? Por que é importante Java permitir que você divida o espaço de nome?
3. Os pacotes são armazenados em _____.
4. Explique a diferença entre **protected** e o acesso padrão.
5. Explique as duas maneiras pelas quais os membros de um pacote podem ser acessados por outros pacotes.
6. Um pacote é, basicamente, um contêiner para classes. Verdadeiro ou falso?
7. Que pacote Java padrão é importado automaticamente para um programa?
8. Diga em suas próprias palavras o que faz a importação estática.
9. O que esta instrução faz?

```
| import static somepack.SomeClass.myMethod;
```

10. A importação estática foi projetada para situações especiais ou é boa prática dar visibilidade a todos os membros estáticos de todas as classes?
11. É adequado a biblioteca Java dividir classes e interfaces em pacotes, mas por que você tem que fazer isso em seus programas? O que há de errado em apenas dar a todas as classes nomes exclusivos para que nunca haja um conflito de nomes?
12. Verdadeiro ou falso:
 - A. Se você não incluir uma instrução **package** em um arquivo-fonte Java, todas as classes declaradas no arquivo não pertencerão a nenhum pacote.
 - B. Se uma classe **A** estiver em um pacote **pkg** e uma segunda classe **B** estiver em um subpacote **pkg.subpkg**, a classe **A** terá automaticamente acesso a todos os membros da classe **B** que usem o modificador **protected**.
13. Coloque os três modificadores de acesso **public**, **private** e **protected** e o acesso padrão em ordem do mais restritivo ao menos restritivo.
14. Suponhamos que uma classe **A** tivesse quatro variáveis de instância com quatro níveis de acesso diferentes, como descrito a seguir:

```
class A {  
    private int x;  
    public int y;  
    protected int z;  
    int w;  
}
```

e suponhamos que uma classe **B** tentasse acessar essas quatro variáveis assim:

```
class B {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.x);
        System.out.println(a.y);
        System.out.println(a.z);
        System.out.println(a.w);
    }
}
```

Quais das quatro chamadas a **println()** no método **main()** da classe **B** serão válidas se:

- A. as classes **A** e **B** estiverem no mesmo pacote?
- B. as classes **A** e **B** estiverem em pacotes diferentes?
- C. a classe **B** for subclasse da classe **A** e estiver no mesmo pacote?
- D. a classe **B** for subclasse da classe **A** e estiver em um pacote diferente?

15. Suponhamos que uma interface **MyConstants** fosse definida assim:

```
package mypackage;

public interface MyConstants {
    public static final int ANSWER = 42;
}
```

Há duas maneiras de dar visibilidade à constante **ANSWER** para que ela seja usada em outra classe: (1) fazer a classe implementar a interface **MyConstants** e (2) usar uma instrução de importação estática. Demonstre como usar cada uma das abordagens para que o código abaixo seja compilado:

```
class MyClass {
    public static void main(String[] args) {
        System.out.println(ANSWER);
    }
}
```

- 16.** Suponhamos que você tivesse um arquivo **MyClass.java** contendo a declaração da classe **MyClass** e quisesse mover **MyClass** para um pacote diferente. Que duas alterações devem ser feitas em **MyClass.java**?
- 17.** Quando você importa uma classe, está adicionando seu nome ao espaço de nome atual. E se já houver uma classe com o mesmo nome? Por exemplo, suponhamos que você definisse a classe a seguir. O que acontecerá quando tentar compilá-la?

```
import java.util.Date;

public class Date {
    public static String getDate() { return "Jan 1, 1970"; }
}
```

18. Suponhamos que um pacote **pkg1** tivesse uma classe chamada **MyClass** e outro pacote **pkg2** tivesse uma classe diferente também chamada **MyClass**. O que acontecerá quando você importar as duas classes? Especificamente, o que aconteceria se você tentasse compilar o código abaixo?

```
import pkg1.MyClass;
import pkg2.MyClass;

public class AnotherClass {
    public static void main(String[] args) {
        MyClass c = new MyClass();
    }
}
```

19. Quando você importa um pacote inteiro em vez de apenas uma classe ou interface, está adicionando todos os nomes de classes e interfaces desse pacote ao espaço de nome atual. E se já houver uma classe ou interface com um dos nomes importados? Por exemplo, suponhamos que você definisse as classes a seguir. O que acontecerá quando tentar compilar e executar a classe **Test**? Observe que o pacote **java.util** já tem uma classe **Date**.

```
import java.util.*;

public class Date {
    public static String getDate() { return "Jan 1, 1970"; }
}

class Test {
    public static void main(String[] args) {
        System.out.println(Date.getDate());
    }
}
```

10

Tratamento de exceções

PRINCIPAIS HABILIDADES E CONCEITOS

- Conhecer a hierarquia de exceções
- Usar **try e catch**
- Entender os efeitos de uma exceção não capturada
- Usar várias cláusulas **catch**
- Capturar exceções de subclasse
- Aninhar blocos **try**
- Lançar uma exceção
- Saber os membros de **Throwable**
- Usar **finally**
- Usar **throws**
- Conhecer as exceções internas Java
- Criar classes de exceção personalizadas

Este capítulo discutirá o tratamento de exceções. Uma exceção é um erro que ocorre no tempo de execução. Usando o subsistema Java de tratamento de exceções, você pode tratar erros de tempo de execução de uma maneira estruturada e controlada. A principal vantagem do tratamento de exceções é que ela automatiza grande parte do código de tratamento de erros que antigamente tinha que ser inserido “à mão” em qualquer programa grande. Por exemplo, em algumas linguagens de computador mais antigas, os códigos de erro são retornados quando um método falha e esses valores devem ser verificados manualmente sempre que o método é chamado. Essa abordagem é ao mesmo tempo tediosa e propensa a erros. O tratamento de exceções otimiza o tratamento de erros permitindo que o programa defina um bloco de código, chamado *tratador de exceções*, que é executado automaticamente quando um erro ocorre. Não é necessário verificar manualmente o sucesso ou a falha de cada chamada de método ou operação específica. Se um erro ocorrer, ele será processado pelo tratador de exceções.

Outra razão que torna o tratamento de exceções importante é Java definir exceções padrão para erros que são comuns nos programas, como a divisão por zero ou um índice fora dos limites de um array. Para reagir a esses erros, seu programa deve estar alerta e tratá-los. Além disso, a biblioteca de APIs Java usa intensamente

exceções. No fim das contas, ser um programador bem-sucedido de Java significa ser plenamente capaz de navegar no subsistema de tratamento de exceções Java.

HIERARQUIA DE EXCEÇÕES

Em Java, todas as exceções são representadas por classes e todas as classes de exceções são derivadas de uma classe chamada **Throwable**. Logo, quando uma exceção ocorre em um programa, um objeto de algum tipo de classe de exceção é gerado. Há duas subclasses diretas de **Throwable**: **Exception** e **Error**. As exceções de tipo **Error** estão relacionadas a erros que não podemos controlar, como os que ocorrem na própria máquina virtual Java. Geralmente os programas não lidam com eles. Portanto, esses tipos de exceções não serão descritos aqui.

Erros que resultam da atividade do programa são representados por subclasses de **Exception**. Por exemplo, erros de divisão por zero, que excedem os limites do array e de I/O se enquadram nessa categoria. Em geral, os programas devem tratar exceções desses tipos. Uma subclasse importante de **Exception** é **RuntimeException**, que é usada para representar vários tipos comuns de erros de tempo de execução.

FUNDAMENTOS DO TRATAMENTO DE EXCEÇÕES

O tratamento de exceções Java é gerenciado por cinco palavras-chave: **try**, **catch**, **throw**, **throws** e **finally**. Elas formam um subsistema interligado em que o uso de uma implica o uso de outra. No decorrer deste capítulo, examinaremos cada palavra-chave com detalhes. No entanto, é útil termos desde o início uma compreensão geral do papel que cada uma desempenha no tratamento de exceções. Resumidamente, veja como funcionam.

As instruções de programa cujas exceções você quiser monitorar ficarão em um bloco **try**. Se uma exceção ocorrer dentro do bloco **try**, ela será *lançada*. Seu código poderá capturar essa exceção usando **catch** e tratá-la de alguma maneira racional. Exceções geradas pelo sistema são lançadas automaticamente pelo sistema de tempo de execução Java. Para lançar manualmente uma exceção, use a palavra-chave **throw**. Em alguns casos, uma exceção que é lançada para fora de um método deve ser especificada como tal por uma cláusula **throws**. Qualquer código que deva ser executado ao sair de um bloco **try** deve ser inserido em um bloco **finally**.

Pergunte ao especialista

P Para não deixar dúvidas, você poderia descrever novamente as condições que fazem uma exceção ser gerada?

R Exceções são geradas de três maneiras diferentes. Em primeiro lugar, a JVM ou o sistema de suporte do tempo de execução pode gerar uma exceção em resposta a algum erro interno sobre o qual não tenhamos controle. Normalmente, o programa não trata esses tipos de exceções. Em segundo lugar, exceções padrão, como as correspondentes à divisão por zero ou índices fora dos limites de um array, são geradas por erros no código do programa. Temos que tratar essas exceções. Em terceiro lugar, podemos gerar manualmente uma exceção usando a instrução **throw**. Independentemente de como uma exceção for gerada, ela será capturada da mesma maneira.

Usando try e catch

As palavras-chave **try** e **catch** formam a base do tratamento de exceções. Elas funcionam em conjunto, ou seja, você não pode ter um **catch** sem ter um **try**. Esta é a forma geral dos blocos **try/catch** de tratamento de exceções:

```
try {
    // bloco de código cujos erros estão sendo monitorados
}
catch (TipoExceç1 obEx){
    // tratador de TipoExceç1
}
catch(TipoExceç2 obEx){
    // tratador de TipoExceç2
}
.
.
```

Aqui, *TipoExceç* é o tipo de exceção que ocorreu. Quando uma exceção é lançada, ela é capturada pela cláusula **catch** correspondente, que então a processa. Como a forma geral mostra, podemos ter mais de uma cláusula **catch** associada a uma instrução **try**. O tipo da exceção determina qual instrução **catch** será executada. Isto é, se o tipo de exceção especificado por uma instrução **catch** coincidir com o da exceção ocorrida, essa cláusula **catch** será executada (e todas as outras serão ignoradas). Quando uma exceção é capturada, *obEx* recebe seu valor.

Agora um ponto importante: se nenhuma exceção for lançada, o bloco **try** terminará normalmente e todas as suas cláusulas **catch** serão ignoradas. A execução será retomada na primeira instrução após o último **catch**. Logo, as cláusulas **catch** só são executadas quando uma exceção é lançada.

***Nota:** JDK 7 adicionou uma nova forma de instrução try que dá suporte ao gerenciamento automático de recursos. Essa nova forma de try se chama try-with-resources. Ela é descrita no Capítulo 11, no contexto do gerenciamento de fluxos de I/O (como os conectados a um arquivo), porque os fluxos são um dos recursos mais usados.*

Exemplo de exceção simples

Este é um exemplo simples que ilustra como monitorar uma exceção e capturá-la. Como você sabe, é um erro tentar indexar um array além de seus limites. Quando isso ocorre, a JVM lança uma **ArrayIndexOutOfBoundsException**. O programa a seguir gera intencionalmente essa exceção e então a captura:

```
// Demonstra o tratamento de exceções.
class ExcDemo1 {
    public static void main(String[] args) {
        int[] nums = new int[4];
        try { ←———— Cria um bloco try.
```

```

System.out.println("Before exception is generated.");

// gera uma exceção de índice fora dos limites
nums[7] = 10; ← Tenta indexar excedendo o
                limite de nums.
System.out.println("this won't be displayed");
}

catch (ArrayIndexOutOfBoundsException exc) { ← Captura erros nos limites
    // captura a exceção
    System.out.println("Index out-of-bounds!");
}
System.out.println("After catch.");
}
}

```

Esse programa exibirá a saída abaixo:

```

Before exception is generated.
Index out-of-bounds!
After catch.

```

Embora curto, esse programa ilustra vários pontos-chave do tratamento de exceções. Em primeiro lugar, o código cujos erros você quer monitorar está dentro de um bloco **try**. Em segundo lugar, quando ocorre uma exceção (nesse caso, pela tentativa de indexar **nums** além de seus limites), ela é lançada fora do bloco **try** e capturada pela instrução **catch**. Nesse ponto, o controle passa para **catch** e o bloco **try** é encerrado. Isto é, **catch** não é chamada. Em vez disso, a execução do programa é transferida para ela. Logo, a instrução **println()** que vem após o índice fora do limite nunca será executada. Após a cláusula **catch** ser executada, o controle do programa continua nas instruções seguintes a **catch**. Portanto, é função do tratador de exceções remediar o problema que causou a exceção (se possível), para que a execução do programa possa continuar normalmente. Se o problema não puder ser remediado, o tratador de exceções deve tomar alguma outra medida apropriada, como informar o usuário e encerrar o programa.

Lembre-se, se nenhuma exceção for lançada por um bloco **try**, nenhuma cláusula **catch** será executada e o controle do programa será retomado após a instrução **catch**. Para confirmar isso, no programa anterior, mude a linha

```
| nums[7] = 10;
```

para

```
| nums[0] = 10;
```

Agora, nenhuma exceção é gerada e o bloco **catch** não é executado.

É importante entender que as exceções do código que fica dentro de um bloco **try** estão sendo monitoradas. Isso inclui exceções que podem ser geradas por um método chamado de dentro do bloco **try**. Uma exceção lançada por um método chamado de dentro de um bloco **try** pode ser capturada pelas cláusulas **catch** associadas a esse bloco **try** – presumindo, claro, que o próprio método não capture a exceção. Por exemplo, este é um programa válido:

```
/* Uma exceção pode ser gerada por um
método e capturada por outro. */
```

```

class ExcTest {
    // Gera uma exceção.
    static void genException() {
        int[] nums = new int[4];

        System.out.println("Before exception is generated.");

        // gera uma exceção de índice fora do limite
        nums[7] = 10; ← A exceção é gerada aqui.
        System.out.println("this won't be displayed");
    }
}

class ExcDemo2 {
    public static void main(String[] args) {

        try {                                A exceção é capturada aqui.
            ExcTest.genException();
        } catch (ArrayIndexOutOfBoundsException exc) { ←
            // captura a exceção
            System.out.println("Index out-of-bounds!");
        }
        System.out.println("After catch.");
    }
}

```

Esse programa produz a saída a seguir, que é igual à produzida pela primeira versão mostrada anteriormente:

```

Before exception is generated.
Index out-of-bounds!
After catch.

```

Já que **genException()** é chamado de dentro de um bloco **try**, a exceção que ele gera (e não captura) é capturada pela instrução **catch** de **main()**. No entanto, é bom ressaltar que se **genException()** tivesse capturado a exceção, ela nunca teria sido passada para **main()**.

Verificação do progresso

1. O que é uma exceção?
2. O código cujas exceções estão sendo monitoradas deve fazer parte de que instrução?
3. O que **catch** faz? Após um **catch** ser executado, o que acontece com o fluxo de execução?

Respostas:

1. Uma exceção é um erro de tempo de execução.
2. Para que as exceções de um código sejam monitoradas, ele deve fazer parte de um bloco **try**.
3. A cláusula **catch** recebe exceções. Ela não é chamada; assim, a execução não retorna para o ponto em que a exceção foi gerada. Em vez disso, continua após o bloco **catch**.

CONSEQUÊNCIAS DE UMA EXCEÇÃO NÃO CAPTURADA

Capturar uma das exceções padrão Java, como fez o programa anterior, tem um benefício adicional: impede que o programa seja encerrado anormalmente. Quando uma exceção é lançada, ela deve ser capturada por um código em algum local. Em geral, quando o programa não captura uma exceção, ela é capturada pela JVM. O problema é que o tratador de exceções padrão da JVM encerra a execução e exibe uma mensagem de erro seguida por uma lista das chamadas de método que levaram à exceção. (Normalmente essa lista é chamada de *rastreamento de pilha*). Por exemplo, nesta versão do exemplo anterior, a exceção de índice fora do limite não é capturada pelo programa.

```
// Deixa a JVM tratar o erro.
class NotHandled {
    public static void main(String[] args) {
        int[] nums = new int[4];

        System.out.println("Before exception is generated.");

        // gera uma exceção de índice fora do limite
        nums[7] = 10;
    }
}
```

Quando ocorre o erro de indexação do array, a execução é interrompida, e a mensagem de erro a seguir é exibida.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
    at NotHandled.main(NotHandled.java:9)
```

Embora essa mensagem seja útil na depuração, no mínimo não seria algo que você gostaria que outras pessoas vissem! Por isso, é importante seu programa tratar ele próprio as exceções, em vez de depender da JVM.

Como mencionado anteriormente, o tipo da exceção deve coincidir com o tipo especificado em uma instrução **catch**. Se não coincidir, a exceção não será capturada. Por exemplo, o programa abaixo tenta capturar um erro no limite do array com a instrução **catch** de uma **ArithmeticException** (outra das exceções internas Java). Quando o limite do array é excedido, uma **ArrayIndexOutOfBoundsException** é gerada, mas não será capturada pela instrução **catch**. Isso resulta no programa sendo encerrado anormalmente.

```
// Não funcionará!
class ExcTypeMismatch {
    public static void main(String[] args) {
        int[] nums = new int[4];
```

Essa linha lança uma
ArrayIndexOutOfBoundsException.

```
        try {
            System.out.println("Before exception is generated.");

            //gera uma exceção de índice fora do limite
            nums[7] = 10; ←
            System.out.println("this won't be displayed");
        }
```

```

    }

    /* Não pode capturar um erro de limite de array com uma
       ArithmeticException. */
    catch (ArithmetricException exc) { ← Essa linha tenta capturá-la com uma
        // captura a exceção                                ArithmeticException.
        System.out.println("Index out-of-bounds!");
    }
    System.out.println("After catch.");
}
}

```

A saída é mostrada aqui:

```

Before exception is generated.
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
        at ExcTypeMismatch.main(ExcTypeMismatch.java:10)

```

Como a saída mostra, a instrução **catch** de uma **ArithmetricException** não captura uma **ArrayIndexOutOfBoundsException**.

EXCEÇÕES PERMITEM QUE VOCÊ TRATE ERROS NORMALMENTE

Um dos principais benefícios do tratamento de exceções é que ele permite que seu programa responda a um erro de maneira elegante e racional. Em alguns casos, pode ser possível corrigir o problema e permitir que o programa continue a ser executado. Por exemplo, se o usuário tentar abrir um arquivo, mas especificar um nome inválido, você poderia contatá-lo novamente, pedindo um novo nome de arquivo. Em outros casos, o erro não pode ser corrigido, mas a execução do programa pode continuar. Por exemplo, uma conexão de rede poderia esgotar seu tempo-limite, mas o programa que a estava usando continuaria executando outras tarefas que não dependessem dela. Nesse caso, você pode informar o usuário sobre o problema, cancelar a operação que causou a exceção, mas permitir que outras partes do programa continuem. É claro que, às vezes, não há como corrigir ou contornar um problema e a execução do programa deve terminar. No entanto, mesmo assim, você deve executar um encerramento organizado.

Para ter uma ideia de como um tratador de exceções pode impedir o encerramento abrupto do programa, considere o exemplo a seguir. Ele divide os elementos de um array pelos de outro. Se uma divisão por zero ocorrer, uma **ArithmetricException** será gerada. No programa, essa exceção é tratada pelo relato do erro e a execução continua. Logo, tentar dividir por zero não causa um erro abrupto de tempo de execução que resultaria no encerramento do programa. Em vez disso, a situação é tratada, permitindo que a execução do programa continue. Obviamente é muito simples impedir um erro de divisão por zero se assegurarmos que o denominador não seja zero antes de a divisão ocorrer. No entanto, produzir erros de divisão por zero proporciona uma maneira fácil de gerarmos exceções para fins de demonstração.

```
// Trata o erro e continua a execução.
class ExcDemo3 {
    public static void main(String[] args) {
        int[] numer = { 4, 8, 16, 32, 64, 128 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i<numer.length; i++) {
            try {
                System.out.println(numer[i] + " / " +
                                   denom[i] + " is " +
                                   numer[i]/denom[i]);
            }
            catch (ArithmeticException exc) {
                // captura a exceção
                System.out.println("Can't divide by Zero!");
            }
        }
    }
}
```

A saída do programa é mostrada abaixo:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
```

Esse exemplo ilustra outro ponto importante: uma vez que uma exceção foi tratada, ela é removida do sistema. Portanto, no programa, cada vez que o laço é percorrido, entramos novamente no bloco **try**; qualquer exceção anterior terá sido tratada. Isso permite que seu programa trate erros repetidos.

Verificação do progresso

1. O tipo de exceção de uma cláusula **catch** é importante?
2. O que acontece quando uma exceção não é capturada?
3. Quando ocorre uma exceção, o que o programa deve fazer?

Respostas:

1. O tipo de exceção de uma cláusula **catch** deve coincidir com o tipo de exceção que se deseja capturar.
2. Uma exceção não capturada acaba levando ao encerramento anormal do programa.
3. Um programa deve tratar exceções de uma maneira racional e elegante, eliminando sua causa, se possível, e retornando a execução.

USANDO VÁRIAS CLÁUSULAS catch

Como mencionado, você pode associar mais de uma cláusula **catch** a uma instrução **try**. Na verdade, isso é comum. No entanto, cada **catch** deve capturar um tipo de exceção diferente. Por exemplo, o programa mostrado aqui é uma variação do anterior. Ele captura erros tanto de limite de array quanto de divisão por zero. Nessa versão, o tamanho de **numer** é maior que o de **denom**. Logo, em algum momento, um erro de limite de array será produzido. A cláusula **catch** adicional tratará dele.

```
// Usa várias cláusulas catch.
class ExcDemo4 {
    public static void main(String[] args) {
        // Aqui, numer é maior do que denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i<numer.length; i++) {
            try {
                System.out.println(numer[i] + " / " +
                    denom[i] + " is " +
                    numer[i]/denom[i]);
            }
            catch (ArithmaticException exc) { ← Várias cláusulas catch.
                // captura a exceção
                System.out.println("Can't divide by Zero!");
            }
            catch (ArrayIndexOutOfBoundsException exc) { ←
                // captura a exceção
                System.out.println("No matching element found.");
            }
        }
    }
}
```

Esse programa produz a saída a seguir:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
No matching element found.
```

Como a saída confirma, cada instrução **catch** responde apenas ao seu tipo de exceção.

Em geral, as cláusulas **catch** são verificadas na ordem em que ocorrem em um programa. Só uma cláusula que apresente correspondência é executada. Todos os outros blocos **catch** são ignorados.

CAPTURANDO EXCEÇÕES DE SUBCLASSES

Há um ponto importante no uso de várias cláusulas **catch** relativo às subclasses. A cláusula **catch** de uma superclasse também será aplicada a qualquer uma de suas subclasses. Se você quiser capturar exceções tanto do tipo da superclasse quanto do tipo da subclasse, insira primeiro a subclasse na sequência **catch**. Se não o fizer, a instrução **catch** da superclasse também capturará todas as classes derivadas. Essa regra é autoaplicável, porque inserir a superclasse antes faz um código inalcançável ser criado, já que a cláusula **catch** da subclasse não pode ser executada. Em Java, códigos inalcançáveis causam um erro de tempo de compilação.

Por exemplo, considere o programa a seguir. Ele gera tanto uma **ArrayIndexOutOfBoundsException** quanto uma **ArithmaticException**. No entanto, captura **ArrayIndexOutOfBoundsException** e **Exception**. Isso funciona porque **Exception** é a superclasse de todas as exceções relacionadas a programas. Elas incluem **ArrayIndexOutOfBoundsException** e **ArithmaticException**, entre muitas outras. Ou seja, capturar **Exception** também captura **ArithmaticException**.

```
// Subclasses devem preceder as superclasses em cláusulas catch.
class ExcDemo5 {
    public static void main(String[] args) {
        // Aqui, numer é mais longo do que denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i<numer.length; i++) {
            try {
                System.out.println(numer[i] + " / " +
                    denom[i] + " is " +
                    numer[i]/denom[i]);
            }
            catch (ArrayIndexOutOfBoundsException exc) { ←———— Captura a subclasse.
                // captura a exceção
                System.out.println("No matching element found.");
            }
            catch (Exception exc) { ←———— Captura a superclasse.
                System.out.println("Some exception occurred.");
            }
        }
    }
}
```

A saída do programa é mostrada abaixo:

```
4 / 2 is 2
Some exception occurred.
16 / 4 is 4
32 / 4 is 8
```

```

| Some exception occurred.
| 128 / 8 is 16
| No matching element found.
| No matching element found.

```

Nesse caso, a primeira cláusula **catch** trata **ArrayIndexOutOfBoundsException**. A segunda captura todas as outras exceções relativas a programas, inclusive a **Arithmeti-
cException** gerada quando ocorre uma divisão por zero.

A ordem das cláusulas **catch** do exemplo anterior é importante. Como explicado, uma exceção de subclasse deve ser capturada antes da exceção de sua superclasse. Faça um teste e tente inverter as duas cláusulas **catch** desta forma:

```

// Parece certo, mas na verdade está errado!
catch (Exception exc) {
    System.out.println("Some exception occurred.");
}
catch (ArrayIndexOutOfBoundsException exc) {
    // captura a exceção
    System.out.println("No matching element found.");
}

```

Embora “pareça correta”, essa sequência não será compilada. A primeira instrução **catch** capturará todas as exceções e a segunda nunca será alcançada, o que produzirá um erro de tempo de compilação.

Pergunte ao especialista

P Por que eu iria querer capturar exceções da superclasse?

R Há, claro, várias razões. Estas são algumas. Em primeiro lugar, se você adicionar uma cláusula **catch** que capture exceções de tipo **Exception**, na verdade terá adicionado uma cláusula “que captura tudo” ao seu tratador que lida com as exceções relacionadas ao programa. Embora normalmente não seja recomendada, essa cláusula “que captura tudo” pode ser útil em uma situação em que o encerramento anormal do programa tiver que ser evitado não importando o que ocorrer. Em segundo lugar, em algumas situações, uma categoria inteira de exceções pode ser tratada pela mesma cláusula. A captura da superclasse dessas exceções permitirá que você trate todas sem código duplicado.

BLOCOS try PODEM SER ANINHADOS

Um bloco **try** pode ser aninhado dentro de outro. Uma exceção gerada dentro do bloco **try** interno que não seja capturada por um **catch** associado a esse **try** será propagada para o bloco **try** externo. Por exemplo, aqui a **ArrayIndexOutOfBoundsException** não é capturada pelo **catch** interno e sim pelo externo:

```
// Usa um bloco try aninhado.
class NestTrys {
    public static void main(String[] args) {
        // numer é mais longo do que denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        try { // try externo ← Blocos try aninhados.
            for(int i=0; i<numer.length; i++) {
                try { // try aninhado ←
                    System.out.println(numer[i] + " / " +
                        denom[i] + " is " +
                        numer[i]/denom[i]);
                }
                catch (ArithmaticException exc) {
                    // captura a exceção
                    System.out.println("Can't divide by Zero!");
                }
            }
            catch (ArrayIndexOutOfBoundsException exc) {
                // captura a exceção
                System.out.println("No matching element found.");
                System.out.println("Fatal error - program terminated.");
            }
        }
    }
}
```

A saída do programa é mostrada abaixo:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
Fatal error - program terminated.
```

Nesse exemplo, uma exceção que pode ser tratada pelo **try** interno – um erro de divisão por zero – permite que o programa continue. No entanto, um erro de limite de array é capturado pelo **try** externo, o que encerra o programa.

Embora certamente não seja a única razão para usarmos instruções **try** aninhadas, o programa anterior mostra algo importante que pode ser generalizado. Com frequência blocos **try** aninhados são usados para permitir que diferentes categorias de erros sejam tratadas de maneiras distintas. Alguns tipos de erros são catastróficos e não podem ser corrigidos. Outros são menores e podem ser tratados imediatamente. Muitos programadores usam um bloco **try** externo para capturar os erros mais graves, permitindo que blocos **try** internos tratem os menos sérios, se possível.

Verificação do progresso

1. Um bloco **try** pode ser usado para tratar dois ou mais tipos de exceções diferentes?
2. A cláusula **catch** de uma exceção da superclasse também captura subclasses dessa superclasse?
3. Em blocos **try** aninhados, o que acontece a uma exceção que não é capturada pelo bloco interno?

LANÇANDO UMA EXCEÇÃO

Os exemplos anteriores capturaram exceções geradas automaticamente pela JVM. Contudo, é possível lançar manualmente uma exceção usando a instrução **throw**. Sua forma geral é mostrada a seguir:

```
throw obExceç;
```

Aqui, *obExceç* deve ser um objeto de uma classe de exceção derivada de **Throwable**.

Veja um exemplo que ilustra a instrução **throw** lançando manualmente uma **ArithmeticException**:

```
// Lança manualmente uma exceção.
class ThrowDemo {
    public static void main(String[] args) {
        try {
            System.out.println("Before throw.");
            throw new ArithmeticException(); ← Lança uma exceção.
        }
        catch (ArithmetricException exc) {
            // captura a exceção
            System.out.println("Exception caught.");
        }
        System.out.println("After try/catch block.");
    }
}
```

A saída do programa é esta:

```
| Before throw.
| Exception caught.
| After try/catch block.
```

Respostas:

1. Sim.
2. Sim.
3. Uma exceção não capturada por um bloco **try/catch** interno passa para o bloco **try** externo.

Observe como a **ArithmeticException** foi criada com o uso de **new** na instrução **throw**. Lembre-se, **throw** lança um objeto, logo, você deve criar um objeto para ela lançar. Isto é, você não pode apenas lançar um tipo.

Pergunte ao especialista

P Por que eu iria querer lançar uma exceção manualmente?

R Quase sempre, as exceções que lançamos são instâncias de classes de exceção que criamos. Como veremos posteriormente neste capítulo, criar nossas próprias classes de exceções nos permite tratar erros no código como parte da estratégia geral de tratamento de exceções do programa.

Relançando uma exceção

Uma exceção capturada por uma instrução **catch** pode ser relançada para ser capturada por um **catch** externo. A razão mais provável para fazermos um relançamento dessa forma é permitir que vários tratadores acessem a exceção. Por exemplo, pode ocorrer de um tratador gerenciar um aspecto de uma exceção e um segundo tratador lidar com outro aspecto. Lembre-se de que, quando relançarmos uma exceção, ela não será recapturada pela mesma cláusula **catch**, mas sim propagada para uma instrução **catch** externa. Para relançar uma exceção, use uma instrução **throw** dentro de uma cláusula **catch**, lançando a exceção passada como argumento.

O programa a seguir ilustra o relançamento de uma exceção:

```
// Relança uma exceção.
class Rethrow {
    public static void genException() {
        // aqui, numer é mais longo do que denom
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i<numer.length; i++) {
            try {
                System.out.println(numer[i] + " / " +
                    denom[i] + " is " +
                    numer[i]/denom[i]);
            }
            catch (ArithmetricException exc) {
                // captura a exceção
                System.out.println("Can't divide by Zero!");
            }
            catch (ArrayIndexOutOfBoundsException exc) {
                // captura a exceção
                System.out.println("No matching element found.");
                throw exc; // relança a exceção
            }
        }
    }
}
```

↑
Relança a exceção.

```

    }
}

class RethrowDemo {
    public static void main(String[] args) {
        try {
            Rethrow.genException();
        }
        catch(ArrayIndexOutOfBoundsException exc) { ←———— Captura a exceção relançada.
            // recaptura a exceção
            System.out.println("Fatal error - " +
                "program terminated.");
        }
    }
}

```

Nesse programa, erros de divisão por zero são tratados localmente, por `genException()`, mas um erro de limite de array é relançado. Nesse caso, ele é capturado por `main()`.

Verificação do progresso

1. O que `throw` faz?
2. `throw` lança tipos ou objetos?
3. Uma exceção pode ser relançada após ser capturada?

EXAME MAIS DETALHADO DE `Throwable`

Até agora, capturamos exceções, mas nada fizemos com o objeto de exceção. Como todos os exemplos anteriores mostram, uma cláusula `catch` especifica um tipo de exceção e um parâmetro. O parâmetro recebe o objeto de exceção. Já que todas as exceções são subclasses de `Throwable`, todas dão suporte aos métodos definidos por `Throwable`. Alguns dos mais usados são mostrados na Tabela 10-1.

Dos métodos definidos por `Throwable`, `printStackTrace()` e `toString()` estão entre os mais interessantes. Você pode exibir a mensagem de erro padrão mais um registro das chamadas de método que levam ao lançamento da exceção chamando `printStackTrace()`, e pode usar `toString()` para recuperar a mensagem de erro padrão associada à exceção. O método `toString()` também é chamado quando uma exceção é usada como argumento de `println()`. O programa a seguir demonstra esses métodos:

Respostas:

1. `throw` gera uma exceção.
2. `throw` lança objetos. Claro que esses objetos devem ser instâncias de classes de exceção válidas.
3. Sim.

Tabela 10-1 Métodos mais usados definidos por Throwable

Método	Descrição
Throwable fillInStackTrace()	Retorna um objeto Throwable contendo um rastreamento de pilha completo. Esse objeto pode ser relançado.
String getLocalizedMessage()	Retorna uma descrição localizada da exceção.
String getMessage()	Retorna uma descrição da exceção.
void printStackTrace()	Exibe o rastreamento de pilha.
void printStackTrace (PrintStream <i>fluxo</i>)	Envia o rastreamento de pilha para o fluxo especificado.
void printStackTrace (PrintWriter <i>fluxo</i>)	Envia o rastreamento de pilha para o fluxo especificado.
String toString()	Retorna um objeto String contendo uma descrição completa da exceção. Esse método é chamado por println() , por exemplo, na exibição de um objeto Throwable .

```
// Usando dois métodos de Throwable.

class ExcTest {
    static void genException() {
        int[] nums = new int[4];

        System.out.println("Before exception is generated.");

        // gera uma exceção de índice fora do limite
        nums[7] = 10;
        System.out.println("this won't be displayed");
    }
}

class UseThrowableMethods {
    public static void main(String[] args) {

        try {
            ExcTest.genException();
        }
        catch (ArrayIndexOutOfBoundsException exc) {
            // captura a exceção
            System.out.println("Standard message is: ");
            System.out.println(exc);
            System.out.println("\nStack trace: ");
            exc.printStackTrace();
        }
        System.out.println("After catch.");
    }
}
```

A saída desse programa é mostrada aqui:

```
Before exception is generated.
Standard message is:
java.lang.ArrayIndexOutOfBoundsException: 7

Stack trace:
java.lang.ArrayIndexOutOfBoundsException: 7
    at ExcTest.genException (UseThrowableMethods.java:10)
    at UseThrowableMethods.main (UseThrowableMethods.java:19)

After catch.
```

Preste atenção no rastreamento de pilha. Ele exibe a ordem em que os métodos que levam à exceção são chamados, com o último método sendo exibido no topo. Aqui, o rastreamento mostra que **main()** chamou **genException()**. Já que **genException()** está no topo, é o método em que a exceção ocorreu.

USANDO **finally**

Podemos querer definir um bloco de código para ser executado na saída de um bloco **try/catch**. Por exemplo, uma exceção poderia causar um erro que encerrasse o método atual, fazendo-o retornar prematuramente. No entanto, o método pode ter que executar alguma ação antes de ser encerrado. Por exemplo, ele pode ter alocado algum recurso que precise ser liberado. É importante que uma exceção não impeça que o recurso seja liberado. Esses tipos de circunstâncias são comuns em programação e Java fornece uma maneira conveniente de tratá-las: **finally**.

O bloco **finally** será executado sempre que a execução deixar um bloco **try/catch**, não importando as condições causadoras. Isto é, tendo o bloco **try** terminado normalmente, ou devido a uma exceção, o último código executado será o definido por **finally**. O bloco **finally** também é executado quando um código do bloco **try** ou de qualquer de suas instruções **catch** retorna do método.

Para especificar um bloco **finally**, adicione-o ao fim de uma sequência **try/catch**. A forma geral de um bloco **try/catch** que inclui **finally** é mostrada abaixo.

```
try {
    // bloco de código cujos erros estão sendo monitorados
}
catch (TipoExceç1 obEx){
    // tratador de TipoExceç1
}
catch(TipoExceç2 obEx){
    // tratador de TipoExceç2
}
//...
finally {
    // código de finally
}
```

Veja um exemplo de **finally**:

```
// Usa finally.
class UseFinally {
    public static void genException(int what) {
        int t;
        int[] nums = new int[2];

        System.out.println("Receiving " + what);
        try {
            switch(what) {
                case 0:
                    t = 10 / what; // gera erro de divisão por zero
                    break;
                case 1:
                    nums[4] = 4; // gera erro de índice de array.
                    break;
                case 2:
                    return; // retorna do bloco try
            }
        }
        catch (ArithmetricException exc) {
            // captura a exceção
            System.out.println("Can't divide by Zero!");
            return; // retorna de catch
        }
        catch (ArrayIndexOutOfBoundsException exc) {
            // captura a exceção.
            System.out.println("No matching element found.");
        }
        finally { ← Esta instrução é executada quando saímos de blocos try/catch.
            System.out.println("Leaving try.");
        }
    }
}

class FinallyDemo {
    public static void main(String[] args) {

        for(int i=0; i < 3; i++) {
            UseFinally.genException(i);
            System.out.println();
        }
    }
}
```

Esta é a saída produzida pelo programa:

```
Receiving 0
Can't divide by Zero!
Leaving try.
```

```

| Receiving 1
| No matching element found.
| Leaving try.

| Receiving 2
| Leaving try.

```

Como a saída mostra, independentemente de como saímos do bloco **try**, o bloco **finally** é executado.

Verificação do progresso

1. As classes de exceção são subclasses de que classe?
2. Quando o código de um bloco **finally** é executado?
3. Como você pode exibir um rastreamento de pilha dos eventos que levam a uma exceção?

USANDO throws

Em alguns casos, quando um método gera uma exceção que ele não trata, deve declará-la em uma cláusula **throws**. A forma geral de um método que inclui uma cláusula **throws** é a seguinte:

```

tipo-ret nomeMét(lista-parâm) throws lista-exceç {
    // corpo
}

```

Aqui, *lista-exceç* é uma lista separada por vírgulas com as exceções que o método pode lançar para fora dele.

Você deve estar se perguntando por que não precisou especificar uma cláusula **throws** em alguns dos exemplos anteriores, que lançaram exceções para fora de métodos. A resposta é que exceções que são subclasses de **Error** e **RuntimeException** não precisam ser especificadas em uma lista **throws**. Java apenas presume que o método pode lançar uma. Todos os outros tipos de exceções *têm* de ser declaradas e não fazê-lo causa um erro de tempo de compilação.

Na verdade, você viu um exemplo de uma cláusula **throws** anteriormente neste livro. Como deve lembrar, ao usar entradas do teclado, teve de adicionar a cláusula

```
| throws java.io.IOException
```

a **main()**. Já podemos entender o porquê. Uma instrução de entrada pode gerar uma **IOException** e, naquele momento, você não pôde tratar a exceção. Assim, essa exce-

Respostas:

1. **Throwable**
2. Um bloco **finally** é o último elemento executado quando saímos de um bloco **try/catch**.
3. Para exibir um rastreamento de pilha, chame **printStackTrace()**, que é definido por **Throwable**.

ção seria lançada para fora de **main()** e deveria ser especificada como tal. Agora que você conhece as exceções, pode tratar facilmente **IOException**.

Examinemos um exemplo que trata **IOException**. Ele cria um método chamado **prompt()**, que exibe uma mensagem de solicitação e então lê um caractere a partir do teclado. Já que a entrada está sendo fornecida, uma **IOException** pode ocorrer. No entanto, o método **prompt()** não trata ele próprio a **IOException**. Em vez disso, usa uma cláusula **throws**, ou seja, o método chamador deve tratá-la. No exemplo a seguir, o método chamador é **main()** e ele lida com o erro.

```
// Usa throws.
class ThrowsDemo {
    public static char prompt(String str)
        throws java.io.IOException { ← Observe a cláusula throws.

        System.out.print(str + ": ");
        return (char) System.in.read();
    }

    public static void main(String[] args) {
        char ch;

        try {
            ch = prompt("Enter a letter"); ← Como o método prompt() pode lançar
        }                                     uma exceção, uma chamada a ele deve
        catch(java.io.IOException exc) {       ser inserida em um bloco try.
            System.out.println("I/O exception occurred.");
            ch = 'X';
        }

        System.out.println("You pressed " + ch);
    }
}
```

Aproveitando o gancho, observe que **IOException** é totalmente qualificada com o nome de seu pacote, **java.io**. Como você aprenderá no Capítulo 11, o sistema de I/O Java fica no pacote **java.io**. Logo, é aí que encontramos **IOException**. Também seria possível importar **java.io** e então referenciar **IOException** diretamente.

EXCEÇÕES INTERNAS DA LINGUAGEM JAVA

Dentro do pacote padrão **java.lang**, Java define várias classes de exceção. Algumas foram usadas pelos exemplos anteriores. As mais gerais dessas exceções são subclasses do tipo padrão **RuntimeException**. Já que **java.lang** é importado implicitamente para todos os programas Java, a maioria das exceções derivada de **RuntimeException** fica disponível automaticamente. Além disso, não precisam ser incluídas na lista **throws** de nenhum método. No jargão Java, elas são chamadas de *exceções não verificadas*, porque o compilador não verifica se um método trata ou lança essas exceções. As exceções não verificadas definidas em **java.lang**

Tabela 10-2 Exceções não verificadas definidas em java.lang

Exceção	Significado
ArithmaticException	Erro aritmético, como a divisão por zero.
ArrayIndexOutOfBoundsException	O índice do array está fora dos limites.
ArrayStoreException	Atribuição de um tipo incompatível a um elemento do array.
ClassCastException	Coerção inválida.
EnumConstantNotPresentException	É feita a tentativa de usar um valor de enumeração não definido.
IllegalArgumentException	Argumento inválido usado para chamar um método.
IllegalMonitorStateException	Operação de monitor inválida, como esperar em uma thread não bloqueada.
IllegalStateException	O ambiente ou o aplicativo está no estado incorreto.
IllegalThreadStateException	Operação solicitada não compatível com o estado atual da thread.
IndexOutOfBoundsException	Um índice de algum tipo está fora dos limites.
NegativeArraySizeException	Array criado com um tamanho negativo.
NullPointerException	Uso inválido de uma referência nula.
NumberFormatException	Conversão inválida de um string para um formato numérico.
SecurityException	Tentativa de violar a segurança.
StringIndexOutOfBoundsException	Tentativa de indexar fora dos limites de um string.
TypeNotPresentException	Tipo não encontrado.
UnsupportedOperationException	Uma operação sem suporte foi encontrada.

Tabela 10-3 Exceções verificadas definidas em java.lang

Exceção	Significado
ClassNotFoundException	Classe não encontrada.
CloneNotSupportedException	Tentativa de clonar um objeto que não implementa a interface Cloneable .
IllegalAccessException	O acesso a uma classe é negado.
InstantiationException	Tentativa de criar um objeto de uma interface ou classe abstrata.
InterruptedException	Uma thread foi interrompida por outra thread.
NoSuchFieldException	Um campo solicitado não existe.
NoSuchMethodException	Um método solicitado não existe.
ReflectiveOperationException	Superclasse de exceções relacionadas à reflexão (adicionada pelo JDK 7).

estão listadas na Tabela 10-2. A Tabela 10-3 lista as exceções definidas por **java.lang** que devem ser incluídas na lista **throws** de um método se ele puder gerar uma dessas exceções sem tratá-la. Elas se chamam *exceções verificadas*. Java define vários outros tipos de exceções associados às suas diversas bibliotecas de classes, como **IOException**, já mencionada.

Pergunte ao especialista

P Ouvi dizer que Java dá suporte a algo chamado *exceções encadeadas*. O que são elas?

R As exceções encadeadas foram adicionadas a Java pelo JDK 1.4. O recurso de exceções encadeadas permite que você especifique uma exceção como a causa subjacente de outra. Por exemplo, imagine uma situação em que um método lançasse uma **ArithmaticException** devido a uma tentativa de divisão por zero. No entanto, a causa real do problema foi um erro de I/O, que fez o divisor ser configurado inapropriadamente. Embora o método deva mesmo lançar uma **ArithmaticException**, já que foi esse erro que ocorreu, você também pode querer informar ao código chamador que a causa subjacente foi um erro de I/O. As exceções encadeadas permitem que você manipule essa e qualquer outra situação em que existam camadas de exceções.

Para permitir o uso de exceções encadeadas, dois construtores e dois métodos foram adicionados a **Throwable**. Os construtores são mostrados aqui:

Throwable(Throwable *excCaus*)

Throwable(String *msg*, Throwable *excCaus*)

Na primeira forma, *excCaus* é a exceção que causou a exceção atual, isto é, *excCaus* é a razão subjacente que fez uma exceção ocorrer. A segunda forma permite que você especifique uma descrição e uma exceção causadora. Esses dois construtores também foram adicionados às classes **Error**, **Exception** e **RuntimeException**.

Os métodos de exceção encadeada adicionados a **Throwable** são **getCause()** e **initCause()**.

Esses métodos são mostrados abaixo:

Throwable **getCause()**

Throwable **initCause(Throwable *excCaus*)**

O método **getCause()** retorna a exceção causadora da exceção atual. Se não houver exceção subjacente, **null** será retornado. O método **initCause()** associa *excCaus* à exceção chamadora e retorna uma referência à exceção. Logo, você pode associar uma causa a uma exceção após a exceção ter sido criada. Em geral, **initCause()** é usado para definir uma causa para classes de exceção legadas que não deem suporte aos dois construtores adicionais descritos anteriormente.

As exceções encadeadas não são algo de que todo programa precise. No entanto, em caso sem que o conhecimento de uma causa subjacente seja útil, elas oferecem uma solução elegante.

Verificação do progresso

1. Para que **throws** é usada?
2. Qual é a diferença entre exceções verificadas e não verificadas?
3. Se um método gerar uma exceção que ele próprio trata, deve incluir uma cláusula **throws** para a exceção?

NOVOS RECURSOS DE EXCEÇÕES ADICIONADOS PELO JDK7

Com o lançamento do JDK 7, o mecanismo de tratamento de exceções Java foi expandido pela inclusão de três recursos novos. O primeiro automatiza o processo de liberar um recurso, como um arquivo, quando este não é mais necessário. Ele se baseia em uma forma expandida de **try**, chamada instrução *try-with-resources*, e é descrito no Capítulo 11, quando os arquivos serão discutidos. O segundo recurso novo se chama *multi-catch* e o terceiro às vezes é chamado de *relançamento final* ou *relançamento mais preciso*. Esses dois recursos serão descritos aqui.

Multi-catch permite que duas ou mais exceções sejam capturadas pela mesma cláusula **catch**. Como você aprendeu anteriormente, é possível (na verdade, é comum) um **try** ser seguido por duas ou mais cláusulas **catch**. Embora geralmente cada cláusula **catch** forneça sua própria sequência de código, são comuns situações em que duas ou mais cláusulas **catch** executam a mesma sequência de código, ainda que capturem exceções diferentes. Em vez de ter de capturar cada tipo de exceção individualmente, agora você pode usar a mesma cláusula **catch** para tratá-las sem duplicação de código.

Para criar um multi-catch, especifique uma lista de exceções na mesma cláusula **catch**. Faça isso separando cada tipo de exceção da lista com o operador OR. Cada parâmetro multi-catch é implicitamente **final**. (Você pode especificar **final** explicitamente, se quiser, mas não é necessário.) Já que cada parâmetro multi-catch é implicitamente **final**, não pode receber um novo valor.

Veja como você pode usar o recurso multi-catch para capturar **ArithmetricException** e **ArrayIndexOutOfBoundsException** com a mesma cláusula **catch**:

```
| catch(final ArithmetricException | ArrayIndexOutOfBoundsException e) {
```

Aqui está um programa simples que demonstra o uso de multi-catch:

```
// Usa o recurso multi-catch. Nota: Este código requer JDK 7 ou
// posterior para ser compilado.
class MultiCatch {
    public static void main(String[] args) {
        int a=88, b=0;
        int result;
```

Respostas:

1. Quando um método gera uma exceção verificada que ele não trata, deve declarar esse fato usando uma cláusula **throws**.
2. Nenhuma cláusula **throws** é necessária para exceções não verificadas.
3. Não. Uma cláusula **throws** só é necessária quando o método não trata a exceção.

```
char[] chrs = { 'A', 'B', 'C' };

for(int i=0; i < 2; i++) {
    try {
        if(i == 0)
            result = a / b;// gera uma ArithmeticException
        else
            chrs[5] = 'X'; // gera uma ArrayIndexOutOfBoundsException

        // Esta cláusula catch captura as duas exceções.
    }
    catch(ArithmeticException | ArrayIndexOutOfBoundsException e)
        System.out.println("Exception caught: " + e);
    }

    System.out.println("After multi-catch.");
}
}
```

O programa gerará uma **ArithmeticException** quando a divisão por zero for tentada, e gerará uma **ArrayIndexOutOfBoundsException** quando for feita a tentativa de acesso fora dos limites de **chrs**. As duas exceções são capturadas pela mesma instrução **catch**.

O recurso de relançamento mais preciso restringe o tipo de exceção que pode ser relançado apenas às exceções verificadas que o bloco **try** associado lança, às exceções que não sejam tratadas por uma cláusula **catch** anterior e às que sejam um subtipo ou supertipo do parâmetro. Embora esse recurso não seja usado com frequência, agora ele está disponível para uso. Para que o recurso de relançamento final seja válido, o parâmetro de **catch** deve ser final, ou seja, não deve receber um novo valor dentro do bloco **catch**. Ele também pode ser especificado explicitamente como **final**, mas isso não é necessário.

CRIANDO SUBCLASSES DE EXCEÇÕES

Embora as exceções internas de Java tratem os erros mais comuns, o mecanismo Java de tratamento de exceções não se limita a esses erros. Na verdade, parte do poder da abordagem que Java usa para as exceções está no tratamento das exceções que criamos para erros em nosso próprio código. É fácil criar uma exceção, só temos de definir uma subclasse de **Exception** (que, claro, é subclasse de **Throwable**). Nossas subclasses não precisam implementar algo – é sua existência no sistema de tipos que nos permite usá-las como exceções.

A classe **Exception** não define métodos próprios, mas herda os métodos fornecidos por **Throwable**. Logo, todas as exceções, inclusive as criadas por nós, têm os métodos definidos por **Throwable** disponíveis para elas. Claro, podemos sobrepor um ou mais desses métodos nas subclasses de exceções que criarmos.

Os dois construtores mais usados de **Exception** são:

```
Exception()
Exception(String msg)
```

A primeira forma cria uma exceção que não tem descrição. A segunda permite que você especifique uma descrição da exceção.

Embora geralmente seja útil especificarmos uma descrição quando uma exceção é criada, às vezes é melhor sobrepor **toString()**. Vejamos por quê: a versão de **toString()** definida por **Throwable** (e herdada por **Exception**) exibe o nome da exceção seguido de dois pontos e então sua descrição. Sobrepondo **toString()**, podemos impedir que sejam exibidos o nome da exceção e os dois pontos. Isso gera uma saída mais limpa, o que, em alguns casos, é desejável. É claro que podemos especificar uma mensagem e sobrepor **toString()**. Dessa forma, o método **getMessage()** definido por **Throwable** retornará algo diferente de nulo.

Aqui está um exemplo que cria uma exceção chamada **NonIntResultException**, gerada quando a divisão de dois valores inteiros produz um resultado com componente fracionário. **NonIntResultException** tem dois campos que armazenam os valores inteiros; um construtor; e uma sobreposição do método **toString()**, que permite que uma descrição mais amigável da exceção seja exibida com o uso de **println()**. Também passa uma mensagem na forma de string para o construtor de **Exception** como complemento, mas ela não é usada no programa.

```
// Usa uma exceção personalizada.

// Cria uma exceção.
class NonIntResultException extends Exception {
    int n;
    int d;

    NonIntResultException(int i, int j) {
        super("Result is not an integer.");
        n = i;
        d = j;
    }

    public String toString() {
        return "Result of " + n + " / " + d +
            " is non-integer.";
    }
}

class CustomExceptDemo {
    public static void main(String[] args) {
        // Aqui, numer contém alguns valores ímpares.
        int[] numer = { 4, 8, 15, 32, 64, 127, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i<numer.length; i++) {
```

```

try {
    if((numer[i] % denomin[i]) != 0)
        throw new
            NonIntResultException(numer[i], denomin[i]);
    System.out.println(numer[i] + " / " +
                        denomin[i] + " is " +
                        numer[i]/denomin[i]);
}
catch (ArithmaticException exc) {
    // captura a exceção
    System.out.println("Can't divide by Zero!");
}
catch (ArrayIndexOutOfBoundsException exc) {
    // captura a exceção
    System.out.println("No matching element found.");
}
catch (NonIntResultException exc) {
    System.out.println(exc);
}
}
}

```

A saída do programa é mostrada abaixo:

```
| 4 / 2 is 2
| Can't divide by Zero!
| Result of 15 / 4 is non-integer.
| 32 / 4 is 8
| Can't divide by Zero!
| Result of 127 / 8 is non-integer.
| No matching element found.
| No matching element found.
```

Pergunte ao especialista

P Quando devo usar o tratamento de exceções em um programa? Quando devo criar minhas próprias classes de exceção personalizadas?

R Já que a API Java faz uso massivo de exceções para relatar erros, quase todos os programas do mundo real usam o tratamento de exceções. Essa é a parte do tratamento de exceções que a maioria dos programadores novos de Java acha fácil. É mais difícil decidir quando e como usar suas próprias exceções personalizadas. Em geral, os erros podem ser relatados de duas maneiras: com valores de retorno e com exceções. Quando uma abordagem é melhor do que a outra? Uma resposta direta seria: em Java, o tratamento de exceções deve ser a norma. Certamente, retornar um código de erro é uma alternativa válida em alguns casos, mas as exceções fornecem uma maneira mais poderosa e estruturada de tratar erros. Elas são a maneira como os programadores profissionais de Java tratam erros em seu código.

TENTE ISTO 10-1 Adicionando exceções às classes de pilha simples

SimpleStackExc.java
 FixedLengthStack.java
 ISimpleStack.java
 SimpleStackExcDemo.java

Neste projeto, você criará duas classes de exceções para serem usadas pelas classes de pilha desenvolvidas na seção Tente isto 8-1. Elas indicarão as condições de pilha cheia e vazia. Essas exceções serão lançadas pelos métodos **push()** e **pop()**, respectivamente, para relatar erros. Em seguida, o projeto adicionará as exceções à classe **FixedLengthStack**. No Exercício 19, você será solicitado a adicioná-las a **DynamicStack**. Como verá, o uso de exceções para relatar erros é uma melhoria significativa nas classes de pilha.

PASSO A PASSO

- Crie um arquivo chamado **SimpleStackExc.java**. Nesse arquivo, adicione as exceções a seguir.

```
/*
 * Tente isto 10-1
 *
 * Adiciona o tratamento de exceções às classes de pilha.
 */

// Exceção para erros de pilha cheia.
class StackFullException extends Exception {
    int size;

    StackFullException(int s) {
        super("Stack Full");
        size = s;
    }

    public String toString() {
        return "\nStack is full. Maximum size is " + size;
    }
}

// Uma exceção para erros de pilha vazia.
class StackEmptyException extends Exception {

    StackEmptyException() {
        super("Stack Empty");
    }

    public String toString() {
        return "\nStack is empty.";
    }
}
```

Uma **StackFullException** é gerada quando é feita uma tentativa de armazenar um item em uma pilha já cheia. Observe que o tamanho máximo da pilha é passado para o construtor para que essa informação possa ser relatada para o usuário. Uma **StackEmptyException** é gerada quando é feita uma tentativa de remover um elemento de uma pilha vazia. Como complemento, as duas passam uma mensagem na forma de string para o construtor de **Exception**, mas também sobrepõem **toString()**.

2. Modifique a classe **FixedLengthStack** para que ela lance exceções quando um erro ocorrer, como mostrado aqui.

```
// Pilha de tamanho fixo para caracteres que usa exceções.
class FixedLengthStack implements ISimpleStack {
    private char[] data; // esse array contém a pilha
    private int tos; // índice do topo da pilha

    // Constrói uma pilha vazia dado seu tamanho.
    FixedLengthStack(int size) {
        data = new char[size]; // cria o array para armazenar a pilha
        tos = 0;
    }

    // Constrói uma pilha a partir de outra.
    FixedLengthStack(FixedLengthStack otherStack) {
        // o tamanho da nova pilha é igual ao de otherStack
        data = new char[otherStack.data.length];

        // configura tos com a mesma posição
        tos = otherStack.tos;

        // copia o conteúdo
        for(int i = 0; i < tos; i++)
            data[i] = otherStack.data[i];
    }

    // Constrói uma pilha com valores iniciais.
    FixedLengthStack(char[] chrs) throws StackFullException {
        // cria o array para armazenar os valores iniciais
        data = new char[chrs.length];
        tos = 0;

        // inicializa a pilha inserindo nela o conteúdo
        // de chrs
        for(char ch : chrs)
            push(ch);
    }

    // Insere um caractere na pilha.
    public void push(char ch) throws StackFullException {
        if(isFull())
            throw new StackFullException(data.length);
    }
}
```

```

        data[tos] = ch;
        tos++;
    }

    // Extrai um caractere da pilha.
    public char pop() throws StackEmptyException {
        if(isEmpty())
            throw new StackEmptyException();

        tos--;
        return data[tos];
    }

    // Retorna true se a pilha estiver vazia.
    public boolean isEmpty() {
        return tos==0;
    }

    // Retorna true se a pilha estiver cheia.
    public boolean isFull() {
        return tos==data.length;
    }
}

```

Observe que duas etapas são necessárias para a inclusão de exceções em **push()** e **pop()**. Em primeiro lugar, os dois métodos devem ter uma cláusula **throws** adicionada a suas declarações. Em segundo lugar, quando um erro ocorrer, esses métodos lançarão uma exceção apropriada. O uso de exceções permite que o código chamador trate o erro de uma maneira racional. Você deve lembrar que a versão anterior apenas relatava o erro. Lançar uma exceção é uma abordagem muito melhor. Além disso, é a abordagem *adeuada*.

Mais uma coisa: observe que o construtor **FixedLengthStack(char[] chrs)** também tem uma cláusula **throws StackFullException**. Isso ocorre porque **push()** é usado para inicializar a pilha com os caracteres de **chrs**. Já que **push()** pode lançar uma exceção, qualquer método que fizer uso dele deve tratar a exceção ou passá-la para o código chamador. Nesse caso, o construtor apenas a passa adiante. É claro que, nesse construtor, não deve ocorrer tal exceção porque **data** tem tamanho suficiente para acomodar os caracteres de **chrs**, mas o compilador continua exigindo que a possibilidade seja considerada.

3. Uma vez que **FixedLengthStack** implementa a interface **ISimpleStack**, ela terá que ser alterada para refletir a cláusula **throws**. Abaixo podemos ver **ISimpleStack** atualizada. Lembre-se, isso deve permanecer em um arquivo próprio chamado **ISimpleStack.java**.

```
// Uma interface de pilha simples que lança exceções.
public interface ISimpleStack {
```

```

    // Insere um caractere na pilha.
    void push(char ch) throws StackFullException;

    // Extrai um caractere da pilha.
    char pop() throws StackEmptyException;

    // Retorna true se a pilha estiver vazia.
    boolean isEmpty();

    // Retorna true se a pilha estiver cheia.
    boolean isFull();
}

```

4. Para testar a classe **FixedLengthStack** atualizada, crie a classe **SimpleStackExcDemo** mostrada aqui e insira-a em um arquivo chamado **SimpleStackExcDemo.java**:

```

// Demonstra as exceções de pilha.
class SimpleStackExcDemo {
    public static void main(String[] args) {
        FixedLengthStack stack = new FixedLengthStack(5);
        char ch;
        int i;

        try {
            // excede a pilha
            for(i=0; i < 6; i++) {
                System.out.print("Attempting to push : " +
                    (char) ('A' + i));
                stack.push((char) ('A' + i));
                System.out.println(" - OK");
            }
            System.out.println();
        }
        catch (StackFullException exc) {
            System.out.println(exc);
        }

        System.out.println();

        try {
            // tenta acessar elemento em fila vazia
            for(i=0; i < 6; i++) {
                System.out.print("Popping next char: ");
                ch = stack.pop();
                System.out.println(ch);
            }
        }
        catch (StackEmptyException exc) {
            System.out.println(exc);
        }
    }
}

```

5. Agora, compile **SimpleStackExc.java**, **ISimpleStack.java** e **FixedLengthStack.java**. Para concluir, compile e execute **SimpleStackExcDemo.java**. Você verá a saída a seguir:

```

Attempting to push : A - OK
Attempting to push : B - OK
Attempting to push : C - OK
Attempting to push : D - OK
Attempting to push : E - OK
Attempting to push : F
Stack is full. Maximum size is 5

Popping next char: E
Popping next char: D
Popping next char: C
Popping next char: B
Popping next char: A
Popping next char:
Stack is empty.

```

EXERCÍCIOS

1. Que classe fica no topo da hierarquia de exceções?
2. Explique resumidamente como **try** e **catch** são usados.
3. O que está errado neste fragmento?

```

// ...
vals[18] = 10;
catch (ArrayIndexOutOfBoundsException exc) {
    // trata erro
}

```

4. O que acontece quando uma exceção não é capturada?
5. O que está errado no seguinte fragmento?

```

class A extends Exception { ... }

class B extends A { ... }
// ...

try {
    // ...
}
catch (A exc) { ... }
catch (B exc) { ... }

```

6. Um **catch** interno pode relançar uma exceção para um **catch** externo?
7. O bloco **finally** é a última parte do código executada antes de o programa terminar. Isso é verdadeiro ou falso? Explique sua resposta.
8. Que tipo de exceções deve ser declarado explicitamente na cláusula **throws** de um método?
9. O que está errado neste fragmento?

```
class MyClass { // ... }
// ...
throw new MyClass();
```

10. Quais são as três maneiras pelas quais uma exceção pode ser gerada?
11. Quais são as duas subclasses diretas de **Throwable**?
12. O que é o recurso multi-catch?
13. Um código deve normalmente lançar exceções de tipo **Error**?
14. Na seção Tente isto 10-1, é mencionado que é preferível lançar uma exceção a exibir uma mensagem de erro. Por que é assim?
15. Suponhamos que **methodA()** chamassem **methodB()**, que chama **methodC()**, que por sua vez chama **methodD()**. Suponhamos também que **methodA()** capturasse todas as exceções, **methodB()** capturasse **Runtime-Exceptions**, **methodC()** capturasse **ArithmetricExceptions** e **methodD()** não capturasse exceções. Quem capturará uma exceção ou erro lançado por **methodD()** se:
 - A. uma **NullPointerException** for lançada?
 - B. uma **ArithmetricException** for lançada?
 - C. um **Error** for lançado?
16. O que há de errado nessa declaração de método?

```
void methodA() {
    throw new ClassNotFoundException();
}
```

17. O que há de errado nessa declaração de método?
18. É válido um método criar uma nova **ArrayIndexOutOfBoundsException** e lançá-la mesmo sem usar um array em local nenhum?

- 19.** Na seção Tente isto 10-1, duas novas exceções foram criadas para a classe **FixedLengthStack** usar. Modifique a classe **DynamicStack** da seção Tente isto 8-1 para que também use essas novas exceções quando apropriado.
- 20.** Simplifique o método a seguir o máximo possível, mas de uma maneira que, para o usuário, ele continue se comportando da mesma forma. Renomeie também o método com algo mais apropriado.

```
int messy(int[] data) {
    int c = 0;

    for(int x : data)
        try {
            int y = 1/x;
        } catch (ArithmetricException exc) { c++; }
    return c;
}
```

- 21.** Qual será a saída do programa abaixo? Explique por quê.

```
class Prog1 {
    public static void main(String[] args) {
        String[] data = {"Larry", "Moe", null, "Curly"};

        try {
            for(String s : data)
                System.out.println(s.length());
        }
        catch (Exception exc) { }
    }
}
```

- 22.** E a deste programa? Explique.

```
class Prog2 {
    public static void main(String[] args) {
        String[] data = {"Larry", "Moe", null, "Curly"};
        int sum = 0;

        try {
            for(String s : data)
                sum += s.length();
        }
        catch (Exception exc) { }
        System.out.println(sum);
    }
}
```

23. Qual será a saída do programa a seguir? Explique.

```
class Prog3 {
    public static void main(String[] args) {
        String[] data = {"Larry", "Moe", null, "Curly"};
        int sum = 0;

        try {
            for(String s : data)
                sum += s.length();
            System.out.println(sum);
        }
        catch (Exception exc) { }
    }
}
```

24. E deste? Por quê?

```
class Prog4 {
    public static void main(String[] args) {
        String[] data = {"Larry", "Moe", null, "Curly"};
        int sum = 0;

        for(String s : data)
            sum += s.length();
        System.out.println(sum);
    }
}
```

25. Qual será a saída do programa abaixo? Explique por quê.

```
class Prog5 {
    public static void main(String[] args) {
        Object[] data = {"Larry", new Prog5(), "Moe", null, "Curly"};

        try {
            for(Object s : data)
                System.out.println((String) s);
        }
        catch (Exception exc) {}
    }
}
```

26. Simplifique o método a seguir o máximo possível, mas de uma maneira que, para o usuário, ele continue se comportando da mesma forma. Renomeie também o método com algo mais apropriado.

```
int foolish(int x) {
    try {
        throw new RuntimeException();
    } catch (RuntimeException exc) {
    }
    finally { return x+3; }
}
```

27. A cláusula **finally** foi projetada para fazer uma limpeza após o tratamento de uma exceção. Mas o que aconteceria se seu código lançasse uma exceção? Isso é válido? Tente fazê-lo e explique o que ocorreu.

11

Usando I/O

PRINCIPAIS HABILIDADES E CONCEITOS

- Entender o fluxo
- Saber a diferença entre fluxos de bytes e de caracteres
- Conhecer as classes de fluxos de bytes Java
- Conhecer as classes de fluxos de caracteres Java
- Conhecer os fluxos predefinidos
- Usar fluxos de bytes
- Usar fluxos de bytes para I/O de arquivo
- Fechar automaticamente um arquivo usando **try-with-resources**
- Ler e gravar dados binários
- Usar arquivos de acesso aleatório
- Usar fluxos de caracteres
- Usar fluxos de caracteres para I/O de arquivo
- Usar a classe **File**
- Aplicar encapsuladores de tipo Java para converter strings numéricos

Desde o começo deste livro, você vem usando partes do sistema de I/O (input/output ou entrada/saída) Java, como a instrução `println()`. No entanto, fez isso sem muita explicação formal. Como o sistema de I/O Java é baseado na hierarquia de classes, não foi possível apresentar sua teoria e seus detalhes sem antes discutir as classes, a herança e as exceções. Agora é hora de você ver detalhadamente a abordagem usada por Java para I/O.

Prepare-se, porque o sistema de I/O Java é bem grande, contendo muitas classes, interfaces e métodos. Parte da razão de seu tamanho é que Java define dois sistemas de I/O completos: um para I/O de bytes e outro para I/O de caracteres. Não será possível discutir todos os aspectos de I/O Java aqui. (Um livro inteiro poderia ser facilmente dedicado ao sistema de I/O Java!) No entanto, este capítulo apresentará alguns dos recursos mais usados e importantes. Felizmente, o sistema de I/O Java é coeso e coerente; uma vez que você entenda seus aspectos básicos, o resto será fácil de dominar.

Antes de começarmos, é necessário fazer uma observação importante. As classes de I/O descritas neste capítulo dão suporte a I/O de arquivo e a I/O de console com base em texto. Elas não são usadas para criar interfaces gráficas de usuário (GUIs). Logo, você não as usará para criar aplicativos de janelas, por exemplo. No entanto, Java inclui um suporte significativo à construção de interfaces gráficas de usuário. Os aspectos básicos da programação de GUIs são encontrados no Capítulo 15, onde os applets são introduzidos, e na Parte II, que oferece uma introdução ao Swing. (Swing é o kit moderno de ferramentas Java para GUIs.)

I/O JAVA É BASEADO EM FLUXOS

Os programas Java executam I/O por intermédio de fluxos. Um *fluxo* é uma abstração que produz ou consome informações. Ele é vinculado a um dispositivo físico pelo sistema de I/O Java. Todos os fluxos se comportam igualmente, mesmo que os dispositivos físicos aos quais estejam vinculados sejam diferentes. Logo, as mesmas classes e métodos de I/O podem ser aplicados a diferentes tipos de dispositivos. Por exemplo, os mesmos métodos usados para gravação no console também podem ser usados na gravação em um arquivo em disco. Java implementa os fluxos dentro de hierarquias de classes definidas no pacote `java.io`.

FLUXOS DE BYTES E FLUXOS DE CARACTERES

Versões modernas de Java definem dois tipos de fluxos: o de bytes e o de caracteres. (A versão original de Java definia só o fluxo de bytes, mas os fluxos de caracteres foram rapidamente adicionados.) Os fluxos de bytes fornecem um meio conveniente para o tratamento de entrada e saída de bytes. Eles são usados, por exemplo, na leitura ou gravação de dados binários. São especialmente úteis no trabalho com arquivos. Os fluxos de caracteres foram projetados para o tratamento da entrada e saída de caracteres. Eles usam o Unicode e, portanto, podem ser internacionalizados. Além disso, em alguns casos, os fluxos de caracteres são mais eficientes do que os fluxos de bytes.

O fato de Java definir dois tipos de fluxos diferentes aumenta muito o sistema de I/O, porque dois conjuntos de hierarquias de classes separados (um para bytes e outro para caracteres) são necessários. O grande número de classes pode fazer o sistema de I/O parecer mais assustador do que realmente é. Lembre-se apenas de que a funcionalidade dos fluxos de bytes é, em grande parte, equivalente à dos fluxos de caracteres.

Outra coisa: no nível mais baixo, todo I/O continua orientado a bytes. Os fluxos baseados em caracteres apenas fornecem um meio conveniente e eficiente de tratamento de caracteres.

CLASSE DE FLUXOS DE BYTES

Os fluxos de bytes são definidos pelo uso de duas hierarquias de classes. No topo delas estão duas classes abstratas: `InputStream` e `OutputStream`. `InputStream` define

Tabela 11-1 Classes de fluxo de bytes

Classe de fluxo de bytes	Significado
BufferedInputStream	Fluxo de entrada armazenado em buffer.
BufferedOutputStream	Fluxo de saída armazenado em buffer.
ByteArrayInputStream	Fluxo de entrada que lê de um array de bytes.
ByteArrayOutputStream	Fluxo de saída que grava em um array de bytes.
DataInputStream	Fluxo de entrada que contém métodos para a leitura dos tipos de dados primitivos.
DataOutputStream	Fluxo de saída que contém métodos para a gravação dos tipos de dados primitivos.
FileInputStream	Fluxo de entrada que lê em um arquivo.
FileOutputStream	Fluxo de saída que grava em um arquivo.
FilterInputStream	InputStream filtrado.
FilterOutputStream	OutputStream filtrado.
InputStream	Classe abstrata que descreve a entrada em fluxo.
ObjectInputStream	Fluxo de entrada para objetos.
ObjectOutputStream	Fluxo de saída para objetos.
OutputStream	Classe abstrata que descreve a saída em fluxo.
PipedInputStream	<i>Pipe</i> de entrada.
PipedOutputStream	<i>Pipe</i> de saída.
PrintStream	Fluxo de saída que contém print() e println() .
PushbackInputStream	Fluxo de entrada que permite que bytes sejam retornados para o fluxo.
SequenceInputStream	Fluxo de entrada que é uma combinação de dois ou mais fluxos de entrada que serão lidos sequencialmente, um após o outro.

as características comuns a fluxos de entrada de bytes, e **OutputStream** descreve o comportamento dos fluxos de saída de bytes.

A partir de **InputStream** e **OutputStream**, são criadas muitas subclasses concretas que oferecem funcionalidade variada e tratam os detalhes de leitura e gravação em vários dispositivos, como os arquivos em disco. As classes de fluxo de bytes são mostradas na Tabela 11-1.

CLASSES DE FLUXOS DE CARACTERES

Os fluxos de caracteres são definidos pelo uso de duas hierarquias de classes encabeçadas por estas duas classes abstratas: **Reader** e **Writer**. **Reader** é usada para entrada e **Writer** para saída. As classes concretas derivadas de **Reader** e **Writer** operam com fluxos de caracteres Unicode.

De **Reader** e **Writer** são derivadas muitas subclasses concretas que tratam várias situações de I/O. Em geral, as classes baseadas em caracteres são equivalentes às classes baseadas em bytes. As classes de fluxos de caracteres são mostradas na Tabela 11-2.

Tabela 11-2 Classes de fluxo de caracteres

Classe de fluxo de caracteres	Significado
BufferedReader	Fluxo de caractere de entrada armazenado em buffer.
BufferedWriter	Fluxo de caractere de saída armazenado em buffer.
CharArrayReader	Fluxo de entrada que lê de um array de caracteres.
CharArrayWriter	Fluxo de saída que grava em um array de caracteres.
FileReader	Fluxo de entrada que lê de um arquivo.
FileWriter	Fluxo de saída que grava em um arquivo.
FilterReader	Leitor filtrado.
FilterWriter	Gravador filtrado.
InputStreamReader	Fluxo de entrada que converte bytes em caracteres.
LineNumberReader	Fluxo de entrada que conta linhas.
OutputStreamWriter	Fluxo de saída que converte caracteres em bytes.
PipedReader	<i>Pipe</i> de entrada.
PipedWriter	<i>Pipe</i> de saída.
PrintWriter	Fluxo de saída que contém print() e println() .
PushbackReader	Fluxo de entrada que permite que caracteres sejam retornados para o fluxo.
Reader	Classe abstrata que descreve a entrada de caracteres em fluxo.
StringReader	Fluxo de entrada que lê de um string.
StringWriter	Fluxo de saída que grava em um string.
Writer	Classe abstrata que descreve a saída de caracteres em fluxo.

FLUXOS PREDEFINIDOS

Como você sabe, todos os programas Java importam automaticamente o pacote **java.lang**. Esse pacote define uma classe chamada **System**, que encapsula vários aspectos do ambiente de tempo de execução. Entre outras coisas, ela contém três variáveis de fluxo predefinidas, chamadas **in**, **out** e **err**. Esses campos são declarados como **public**, **final** e **static** dentro de **System**, ou seja, podem ser usados por qualquer parte do programa e sem referência a um objeto **System** específico.

System.out é o fluxo de saída básico. Por padrão, ele usa o console. **System.in** é a entrada básica, que por padrão é o teclado. **System.err** é o fluxo de erro básico, que por padrão também usa o console. No entanto, esses fluxos podem ser redirecionados para qualquer dispositivo de I/O compatível.

System.in é um objeto de tipo **InputStream**; **System.out** e **System.err** são objetos de tipo **PrintStream**. Eles são fluxos de bytes, mesmo que normalmente sejam usados na leitura e gravação de caracteres no console. São fluxos de bytes e não de caracteres porque os fluxos predefinidos faziam parte da especificação original de Java, que não incluía os fluxos de caracteres. Como veremos, é possível encapsulá-los em fluxos baseados em caracteres, se desejado.

Tabela 11-3 Métodos definidos por InputStream

Método	Descrição
int available()	Retorna o número de bytes de entrada atualmente disponíveis para leitura.
void close()	Fecha a origem da entrada. Tentativas de leitura adicionais gerarão uma IOException .
void mark(int numBytes)	Insere uma marca no ponto atual do fluxo de entrada que permanecerá válida até <i>numBytes</i> bytes serem lidos.
boolean markSupported()	Retorna true se mark()/reset() tiverem suporte no fluxo chamador.
int read()	Retorna uma representação em inteiros do próximo byte disponível da entrada. –1 é retornado quando o fim do fluxo é alcançado.
int read(byte[] buffer)	Tenta ler até <i>buffer.length</i> bytes em <i>buffer</i> e retorna o número de bytes que foram lidos com sucesso. –1 é retornado quando o fim do fluxo é alcançado na primeira tentativa de leitura.
int read(byte[] buffer, int deslocamento, int numBytes)	Tenta ler até <i>numBytes</i> bytes em <i>buffer</i> começando em <i>buffer[deslocamento]</i> e retornando o número de bytes lidos com sucesso. É retornado –1 quando o fim do fluxo é alcançado na primeira tentativa de leitura.
void reset()	Volta o ponteiro da entrada à marca definida anteriormente.
long skip(long numBytes)	Ignora (isto é, salta) <i>numBytes</i> bytes da entrada, retornando o número de bytes ignorados.

Verificação do progresso

1. O que é um fluxo?
2. Que tipos de fluxos Java define?
3. Quais são os fluxos internos?

USANDO OS FLUXOS DE BYTES

Começaremos nosso estudo de I/O Java com os fluxos de bytes. Como explicado, no topo da hierarquia de fluxos de bytes estão as classes **InputStream** e **OutputStream**. A Tabela 11-3 mostra os métodos de **InputStream** e a Tabela 11-4 mostra os de **OutputStream**. Em geral, os métodos de **InputStream** e **OutputStream** podem lançar uma **IOException** em caso de erro. Os métodos definidos por essas duas classes abstratas estão disponíveis para todas as suas subclasses. Logo, formam um conjunto mínimo de funções de I/O que todos os fluxos de bytes terão.

Respostas:

1. Um fluxo é uma abstração que produz ou consome informações.
2. Java define fluxos tanto de bytes quanto de caracteres.
3. **System.in**, **System.out** e **System.err**.

Tabela 11-4 Métodos definidos por OutputStream

Método	Descrição
void close()	Fecha o fluxo de saída. Tentativas de gravação adicionais gerarão uma IOException .
void flush()	Faz qualquer saída que tiver sido armazenada em buffer ser enviada para seu destino, isto é, esvazia o buffer de saída.
void write(int <i>b</i>)	Grava um único byte em um fluxo de saída. Observe que o parâmetro é um int , o que permite que você chame write() com expressões sem ter que convertê-las novamente para byte .
void write(byte[] <i>buffer</i>)	Grava um array de bytes completo em um fluxo de saída.
void write(byte[] <i>buffer</i> , int <i>deslocamento</i> , int <i>numBytes</i>)	Grava um subconjunto de <i>numBytes</i> bytes a partir do array <i>buffer</i> , começando em <i>buffer[deslocamento]</i> .

Lendo a entrada do console

No início, a única maneira de ler entradas do console era usar um fluxo de bytes, e muitos códigos Java ainda usam somente fluxos de bytes. Atualmente, você pode usar fluxos de bytes ou de caracteres. Para códigos comerciais, o método preferido de leitura de entradas no console é usar um fluxo orientado a caracteres. Isso facilita a internacionalização e a manutenção do programa. Também é mais conveniente operar diretamente com caracteres em vez de fazer conversões repetidas entre caracteres e bytes. No entanto, para exemplos de programas, programas utilitários simples de uso próprio e aplicativos que lidem com entradas brutais do teclado, é aceitável usar os fluxos de bytes. Por isso, I/O de console que faz uso de fluxos de bytes será examinado aqui.

Uma vez que **System.in** é instância de **InputStream**, temos automaticamente acesso aos métodos definidos por **InputStream**. Infelizmente, **InputStream** só define um método de entrada, **read()**, que lê bytes. Há três versões de **read()**, que são mostradas abaixo:

```
int read( ) throws IOException
int read(byte[ ] buffer) throws IOException
int read(byte[ ] buffer, int deslocamento, int numBytes) throws IOException
```

No Capítulo 3, você viu como usar a primeira versão de **read()** para ler um único caractere a partir do teclado (a partir de **System.in**). Ela retorna **-1** quando o fim do fluxo é alcançado. A segunda versão lê bytes no fluxo de entrada e os insere em *buffer* até o array ficar cheio, o fim do fluxo ser alcançado ou um erro ocorrer. Ela retorna o número de bytes lidos ou **-1** quando o fim do fluxo é alcançado. A terceira versão lê a entrada em *buffer* começando no local especificado por *deslocamento*. Até *numBytes* bytes podem ser armazenados. Ela retorna o número de bytes lidos ou **-1** quando o fim do fluxo é alcançado. Todas lançam uma **IOException** quando um erro ocorre. Na leitura a partir de **System.in**, o pressionamento de ENTER gera uma condição de fim de fluxo.

Aqui está um programa que demonstra a leitura de um array de bytes a partir de **System.in**. Observe que qualquer exceção de I/O que possa ser gerada é lançada para fora de **main()**. Essa abordagem é comum na leitura a partir do console, mas você pode tratar esses tipos de erros por conta própria, se quiser.

```
// Lê um array de bytes a partir do teclado.

import java.io.*;

class ReadBytes {
    public static void main(String[] args)
        throws IOException {
        byte[] data = new byte[10];

        System.out.println("Enter some characters.");
        int numRead = System.in.read(data); ← Lê um array de bytes a
        System.out.print("You entered: ");           partir do teclado.
        for(int i=0; i < numRead; i++)
            System.out.print((char) data[i]);
    }
}
```

Veja um exemplo de execução:

```
Enter some characters.
Read Bytes
You entered: Read Bytes
```

Gravando a saída no console

Como no caso da entrada do console, originalmente Java só fornecia fluxos de bytes para a saída no console. Java 1.1 adicionou os fluxos de caracteres. Para obtenção de um código mais portável, fluxos de caracteres são recomendados. No entanto, como **System.out** é um fluxo de bytes, a saída no console baseada em bytes ainda é amplamente usada. Na verdade, todos os programas deste livro vistos até agora a usaram! Por isso, ela será examinada aqui.

A saída no console é obtida mais facilmente com os métodos **print()** e **println()**, que você já conhece. Esses métodos são definidos pela classe **PrintStream** (que é o tipo do objeto referenciado por **System.out**). Mesmo com **System.out** sendo um fluxo de bytes, é aceitável usar esse fluxo para saídas simples no console.

Já que **PrintStream** é um fluxo de saída derivado de **OutputStream**, ele também implementa o método de baixo nível **write()**. Portanto, é possível gravar no console usando **write()**. A forma mais simples de **write()** definida por **PrintStream** é mostrada abaixo:

```
void write(int b)
```

Esse método grava o byte especificado por *b*. Embora *b* seja declarada como um inteiro, só os 8 bits de ordem inferior são gravados. Veja um exemplo curto que usa **write()** para exibir o caractere X seguido por uma nova linha:

```
// Demonstra System.out.write().
class WriteDemo {
    public static void main(String[] args) {
        int b;

        b = 'X';
        System.out.write(b); ← Exibe um byte na tela.
        System.out.write('\n');
    }
}
```

Você não usará **write()** com frequência para gravar a saída no console (embora isso possa ser útil em algumas situações), porque **print()** e **println()** são bem mais fáceis de usar.

PrintStream fornece dois métodos de saída adicionais: **printf()** e **format()**. Os dois proporcionam um controle minucioso sobre o formato dos dados gravados. Por exemplo, você pode especificar o número de casas decimais exibidas, a largura de campo mínima ou o formato de um valor negativo. Esses métodos serão examinados no Capítulo 24, quando a formatação for discutida.

Verificação do progresso

1. Que método é usado na leitura de um byte a partir de **System.in**?
2. Além de **print()** e **println()**, que método pode ser usado na gravação de um byte em **System.out**?

LENDO E GRAVANDO ARQUIVOS USANDO FLUXOS DE BYTES

Java fornece várias classes e métodos que permitem a leitura e gravação de arquivos. É claro que os tipos mais comuns são os arquivos em disco. Em Java, todos os arquivos são orientados a bytes e a linguagem fornece métodos para a leitura e gravação de bytes em um arquivo. Logo, é muito comum ler e gravar arquivos usando fluxos de bytes. No entanto, Java permite o encapsulamento de um fluxo de arquivo orientado a bytes dentro de um objeto baseado em caracteres, o que será mostrado posteriormente neste capítulo.

Para criar um fluxo de bytes vinculado a um arquivo, use **FileInputStream** ou **FileOutputStream**. Para abrir um arquivo, simplesmente crie um objeto de uma dessas classes, especificando o nome do arquivo como argumento do construtor. Uma vez que o arquivo for aberto, você poderá ler e gravar nele.

Respostas:

1. Para ler um byte, chame **read()**.
2. Você pode gravar um byte em **System.out** chamando **write()**.

Obtendo entradas de um arquivo

Um arquivo é aberto para obter entradas com a criação de um objeto **FileInputStream**. O construtor abaixo é muito usado:

```
FileInputStream(String nomeArquivo) throws FileNotFoundException
```

Aqui, *nomeArquivo* especifica o nome do arquivo que você deseja abrir. Se ele não existir, uma **FileNotFoundException** será lançada. **FileNotFoundException** é subclasse de **IOException**.

Para ler em um arquivo, você pode usar **read()**. A versão que usaremos é a seguinte:

```
int read() throws IOException
```

Sempre que é chamado, **read()** lê um único byte no arquivo e o retorna como um valor inteiro. Ele retorna -1 quando o fim do arquivo é alcançado e lança uma **IOException** quando ocorre um erro. Portanto, essa versão de **read()** é igual à usada na leitura a partir do console.

Quando tiver terminado de usar um arquivo, você deve fechá-lo chamando o método **close()**. Sua forma geral é mostrada abaixo:

```
void close() throws IOException
```

O fechamento de um arquivo libera os recursos do sistema alocados para ele, permitindo que sejam usados por outro arquivo. Não fechar um arquivo pode resultar em recursos não usados permanecerem alocados.

O programa a seguir usa **read()** para acessar e exibir o conteúdo de um arquivo, cujo nome é especificado como argumento de linha de comando. Repare como os blocos **try/catch** tratam os erros de I/O que podem ocorrer.

```
/* Exibe um arquivo de texto.

Para usar este programa, especifique o nome
do arquivo que deseja ver.
Por exemplo, para ver um arquivo chamado TEST.TXT,
use a linha de comando abaixo.

java ShowFile TEST.TXT
*/
import java.io.*;

class ShowFile {
    public static void main(String[] args)
    {
        int i;
        FileInputStream fin;

        // Primeiro verifica se um arquivo foi especificado.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile File");
        }
    }
}
```

```

        return;
    }

    try {
        fin = new FileInputStream(args[0]); ← Abre o arquivo.
    } catch(FileNotFoundException exc) {
        System.out.println("File Not Found");
        return;
    }

    try {
        // lê bytes até o EOF ser alcançado
        do {
            i = fin.read(); ← Lê o arquivo.
            if(i != -1) System.out.print((char) i);
        } while(i != -1); ← Quando i for igual a -1, o fim
    } catch(IOException exc) { do arquivo foi alcançado.
        System.out.println("Error reading file.");
    }

    try {
        fin.close(); ← Fecha o arquivo.
    } catch(IOException exc) {
        System.out.println("Error closing file.");
    }
}
}

```

Observe que o exemplo anterior fecha o fluxo após o bloco **try** que lê o arquivo ter terminado. Embora ocasionalmente essa abordagem seja útil, Java dá suporte a uma variação que com frequência é uma opção melhor. A variação chama **close()** dentro de um bloco **finally**. Nessa abordagem, todos os códigos que acessam o arquivo ficam dentro de um bloco **try** e o bloco **finally** é usado para fechar o arquivo. Dessa forma, independentemente de como o bloco **try** termine, o arquivo será fechado. Usando o exemplo anterior, vejamos como o bloco **try** que lê o arquivo pode ser recodificado:

```

try {
    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);
} catch(IOException exc) {
    System.out.println("Error Reading File");
} finally { ← Usa uma cláusula finally para
    // Fecha o arquivo na saída do bloco try.
    try {
        fin.close(); ← fechar o arquivo.
    } catch(IOException exc) {
        System.out.println("Error Closing File");
    }
}

```

Em geral, uma vantagem dessa abordagem é que, se o código que acessa um arquivo for encerrado devido a alguma exceção não relacionada à I/O, mesmo assim o arquivo será fechado pelo bloco **finally**. Embora não seja uma questão importante nesse exemplo (ou na maioria dos outros exemplos de programa) porque o programa simplesmente termina se uma exceção inesperada ocorrer, isso pode ser uma grande fonte de problemas em programas maiores. O uso de **finally** evita esse incômodo.

Às vezes, é mais fácil encapsular as partes de um programa referentes à abertura e ao acesso do arquivo dentro do mesmo bloco **try** (em vez de separar as duas) e então usar um bloco **finally** para fechar o arquivo. Por exemplo, aqui está outra maneira de escrever o programa **ShowFile**:

```
/* Esta variação encapsula o código que abre e
   acessa o arquivo dentro do mesmo bloco try.
   O arquivo é fechado pelo bloco finally.
*/

import java.io.*;

class ShowFile {
    public static void main(String[] args)
    {
        int i;
        FileInputStream fin = null; ←———— Aqui, fin é inicializada com null.

        // Primeiro, confirma se um nome de arquivo foi especificado.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // O código a seguir abre um arquivo, lê caracteres até EOF
        // ser alcançado e então fecha o arquivo via um bloco finally.
        try {
            fin = new FileInputStream(args[0]);

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(FileNotFoundException exc) {
            System.out.println("File Not Found.");
        } catch(IOException exc) {
            System.out.println("An I/O Error Occurred");
        } finally {
            // Fecha o arquivo em todos os casos.
            try {
                if(fin != null) fin.close(); ←———— Só fecha fin se não for null.
            }
        }
    }
}
```

```
        } catch(IOException exc) {
            System.out.println("Error Closing File");
        }
    }
}
```

Nessa abordagem, observe que **fin** é inicializada com **null**. Em seguida, no bloco **finally**, o arquivo só é fechado se **fin** não for **null**. Isso funciona porque **fin** só será diferente de **null** se o arquivo for aberto com sucesso. Logo, **close()** não será chamado se uma exceção ocorrer na abertura do arquivo.

É possível compactar um pouco mais a sequência **try/catch** do exemplo anterior. Já que **FileNotFoundException** é subclasse de **IOException**, ela não precisa ser capturada separadamente. Por exemplo, essa cláusula **catch** poderia ser usada para capturar as duas exceções, eliminando a necessidade de captura de **FileNotFoundException** separadamente. Nesse caso, a mensagem de exceção padrão, que descreve o erro, é exibida.

```
...  
} catch(IOException exc) {  
    System.out.println("I/O Error: " + exc);  
} finally {  
    ...  
}
```

Nessa abordagem, qualquer erro, inclusive de abertura de arquivo, será tratado pela única instrução **catch** existente. Devido à sua concisão, essa é a abordagem usada por muitos dos exemplos de I/O do livro. No entanto, é preciso cuidado, porque ela não será apropriada se quisermos lidar separadamente com uma falha de abertura de arquivo, como pode ocorrer se um usuário digitar errado o nome do arquivo. Em tal situação, poderíamos solicitar o nome correto, por exemplo, antes de entrar em um bloco **try** que acesse o arquivo.

Pergunte ao especialista

P Notei que `read()` retorna -1 quando o fim do arquivo é alcançado, mas que não tem um valor de retorno especial para um erro de arquivo. Por que não?

R Em Java, erros são tratados por exceções. Portanto, se `read()`, ou qualquer outro método de I/O, retornar um valor, nenhum erro ocorreu. Essa é uma maneira muito mais limpa do que tratar erros de I/O usando códigos de erro especiais.

Gravando em um arquivo

Para abrir um arquivo para saída, crie um objeto **FileOutputStream**. Aqui estão dois construtores normalmente utilizados:

```
FileOutputStream(String nomeArquivo) throws FileNotFoundException  
FileOutputStream(String nomeArquivo, boolean incluir)  
    throws FileNotFoundException
```

Se o arquivo não puder ser criado, uma **FileNotFoundException** será lançada. Na primeira forma, quando um arquivo de saída é aberto, qualquer arquivo preexistente com o mesmo nome é destruído. Na segunda forma, se *incluir* for igual a **true**, a saída será acrescida ao fim do arquivo. Caso contrário, o arquivo será sobreposto.

Para gravar em um arquivo, você usará o método **write()**. Sua forma mais simples é mostrada aqui:

```
void write(int b) throws IOException
```

Esse método grava o byte especificado por *b* no arquivo. Embora *b* seja declarada como um inteiro, só os 8 bits de ordem inferior são gravados no arquivo. Se um erro ocorrer durante a gravação, uma **IOException** será lançada.

Uma vez que você tiver terminado de usar um arquivo de saída, deve fechá-lo usando o método **close()**, mostrado abaixo:

```
void close() throws IOException
```

O fechamento de um arquivo libera os recursos do sistema alocados para ele, permitindo que sejam usados por outro arquivo. Também assegura que saídas remanescentes em um buffer sejam realmente gravadas.

O exemplo a seguir copia um arquivo de texto. Os nomes dos arquivos de origem e destino são especificados na linha de comando:

```
/* Copia um arquivo de texto.  
Para usar esse programa, especifique o nome  
do arquivo de origem e do arquivo de destino.  
Por exemplo, para copiar um arquivo chamado FIRST.TXT  
em um arquivo chamado SECOND.TXT, use a linha de comando  
a seguir.  
  
java CopyFile FIRST.TXT SECOND.TXT  
*/  
  
import java.io.*;  
  
class CopyFile {  
    public static void main(String[] args)  
    {  
        int i;  
        FileInputStream fin = null;  
        FileOutputStream fout = null;  
  
        // Primeiro, verifica se os dois arquivos foram especificados.  
        if(args.length != 2) {  
            System.out.println("Usage: CopyFile from to");  
            return;  
        }  
  
        // Copia um arquivo.  
        try {  
            // Tenta abrir os arquivos.  
            fin = new FileInputStream(args[0]);
```

```
fout = new FileOutputStream(args[1]);

do {
    i = fin.read(); ← Lê bytes em um arquivo e
    if(i != -1) fout.write(i); ← grava-os em outro.
} while(i != -1);

} catch(IOException exc) {
    System.out.println("I/O Error: " + exc);
} finally {
    try {
        if(fin != null) fin.close();
    } catch(IOException exc) {
        System.out.println("Error Closing Input File");
    }
    try {
        if(fout != null) fout.close();
    } catch(IOException exc) {
        System.out.println("Error Closing Output File");
    }
}
}
```

Verificação do progresso

1. O que `read()` retorna quando o fim do arquivo é alcançado?
 2. Um arquivo deve ser fechado quando não é mais usado?
 3. Um bloco `finally` pode ser usado para fechar um arquivo?

FECHANDO AUTOMATICAMENTE UM ARQUIVO

Na seção anterior, os exemplos de programas fizeram chamadas explícitas a `close()` para fechar um arquivo quando ele não era mais necessário. É assim que os arquivos têm sido fechados desde que Java foi criada. Como resultado, essa abordagem está disseminada nos códigos existentes. Além disso, ela ainda é válida e útil, porém, o JDK 7 adiciona um novo recurso que oferece uma maneira mais otimizada de gerenciar recursos, como os fluxos de arquivo, automatizando o processo de fechamento. Ela se baseia em uma nova versão da instrução `try` chamada `try-with resources`, e que também é conhecida como *gerenciamento automático de recursos*. A principal vantagem de `try-with-resources` é que ela impede a ocorrência de situações em que um arquivo (ou outro recurso) não é liberado quando não é mais necessário.

Respostas:

1. O método **read()** retorna -1 quando o fim do arquivo é alcançado.
 2. Sim.
 3. Sim.

A instrução *try-with-resources* tem a seguinte forma geral:

```
try(especificação-recurso){
    // usa o recurso
}
```

Aqui, *especificação-recurso* é uma instrução que declara e inicializa um recurso, como um arquivo. Ela é composta por uma declaração de variável em que a variável é inicializada com uma referência ao objeto que está sendo gerenciado. Quando o bloco **try** termina, o recurso é liberado automaticamente. Ou seja, no caso de um arquivo, ele é fechado automaticamente. (Logo, não há necessidade de chamar **close()** explicitamente.) Uma instrução **try-with-resources** também pode incluir cláusulas **catch** e **finally**.

A instrução **try-with-resources** só pode ser usada com os recursos que implementam a interface **AutoCloseable** definida por **java.lang**. Essa interface, que foi adicionada pelo JDK 7, define o método **close()**. **AutoCloseable** é herdada pela interface **Closeable** definida por **java.io**. Ambas as interfaces são implementadas pelas classes de fluxo, inclusive **FileInputStream** e **OutputStream**. Portanto, **try-with-resources** pode ser usada no trabalho com fluxos, o que inclui os fluxos de arquivo.

Como primeiro exemplo do fechamento automático de um arquivo, esta é uma versão retrabalhada do programa **ShowFile** que o usa:

```
/* Esta versão do programa ShowFile usa uma instrução try-with-resources
   para fechar automaticamente um arquivo quando ele não é mais necessário.

   Nota: este código requer o JDK 7 ou posterior.
*/

import java.io.*;

class ShowFile {
    public static void main(String[] args) {
        int i;

        // Primeiro, confirma se um nome de arquivo foi especificado.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // O código a seguir usa try-with-resources para abrir um arquivo
        // e depois fechá-lo automaticamente quando o bloco try é deixado.
        try(FileInputStream fin = new FileInputStream(args[0])) { ←
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);
        }
    }
}
```

Bloco **try-with-resources**.

```
    } catch(IOException exc) {
        System.out.println("I/O Error: " + exc);
    }
}
```

No programa, preste atenção em como o arquivo é aberto dentro da instrução **try-with-resources**:

```
|try(FileInputStream fin = new FileInputStream(args[0])) {
```

Observe como a parte de **try** referente à especificação do recurso declara um **FileInputStream** chamado **fin**, que então recebe uma referência ao arquivo aberto por seu construtor. Logo, nessa versão do programa, a variável **fin** é local do bloco **try**, sendo criada quando entramos nele. Quando saímos de **try**, o arquivo associado a **fin** é fechado automaticamente por uma chamada implícita a **close()**. Não precisamos chamar **close()** explicitamente, ou seja, não vamos esquecer de fechar o arquivo. Essa é uma vantagem-chave da instrução **try-with resources**.

É importante entender que o recurso declarado na instrução **try** é implicitamente **final**, isto é, você não pode redefinir o recurso após ele ter sido criado. Além disso, o escopo do recurso está limitado à instrução **try-with-resources**.

Você pode gerenciar mais de um recurso dentro da mesma instrução **try**. Para isso, simplesmente separe cada especificação de recurso com um ponto e vírgula. O programa a seguir mostra um exemplo. Ele refaz o programa **CopyFile** mostrado anteriormente de modo que use a mesma instrução **try-with-resources** para gerenciar tanto **fin** quanto **fout**.

```
/* Versão de CopyFile que usa try-with-resources.  
 Ela demonstra dois recursos (neste caso, arquivos) sendo  
 gerenciados pela mesma instrução try.  
  
 Nota: este código requer o JDK 7 ou posterior.  
 */  
  
import java.io.*;  
  
class CopyFile {  
    public static void main(String[] args)  
    {  
        int i;  
  
        // Primeiro, confirma se os dois arquivos foram especificados.  
        if(args.length != 2) {  
            System.out.println("Usage: CopyFile from to");  
            return;  
        }  
  
        // Abre e gerencia dois arquivos com a instrução try.  
        try (FileInputStream fin = new FileInputStream(args[0]); ← Gerencia  
             FileOutputStream fout = new FileOutputStream(args[1])) ← dois  
            recursos
```

```

    {
        do {
            i = fin.read();
            if(i != -1) fout.write(i);
        } while(i != -1);

    } catch(IOException exc) {
        System.out.println("I/O Error: " + exc);
    }
}
}

```

Nesse programa, observe como os arquivos de entrada e saída são abertos dentro de **try**:

```

try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
{

```

Quando esse bloco **try** terminar, tanto **fin** quanto terão sido fechados. Se você comparar essa versão do programa com a anterior, verá que ela é muito mais curta. A otimização do código-fonte é um benefício adicional de **try-with-resources**.

Há outro aspecto de **try-with-resources** que precisa ser mencionado. Em geral, quando um bloco **try** é executado, é possível que uma exceção ocorrida nele leve a outra exceção quando o recurso é fechado em uma cláusula **finally**. No caso de uma instrução **try** “comum”, a exceção original é perdida, sendo substituída pela segunda exceção. No entanto, em uma instrução **try-with-resources**, a segunda exceção é *suprimida*, mas não é perdida. Em vez disso, ela é adicionada à lista de exceções suprimidas associadas à primeira exceção. A lista de exceções suprimidas pode ser obtida com o uso do método **getSuppressed()** definido por **Throwable**.

Devido a essas vantagens, espera-se que **try-with-resources** seja usada intensamente em novos códigos. Como tal, ela será usada pelos exemplos restantes deste capítulo. Contudo, também é muito importante que você conheça a abordagem tradicional já mostrada em que **close()** é chamado explicitamente. Há várias razões para isso. Em primeiro lugar, há milhões de linhas de código legado que se baseiam na abordagem tradicional sendo amplamente utilizadas. É importante que todos os programadores Java conheçam bem e saibam usar a abordagem tradicional para fazer a manutenção e atualização desses códigos mais antigos. Em segundo lugar, durante algum tempo, você pode ter de programar em um ambiente anterior ao JDK 7. Nessa situação, a instrução **try-with-resources** não estará disponível e a abordagem tradicional deve ser empregada. Para concluir, podem surgir casos em que o fechamento explícito de um recurso seja mais apropriado do que a abordagem automatizada. Apesar disso, se você estiver usando o JDK 7 ou posterior, provavelmente vai querer usar a nova abordagem automatizada para gerenciar recursos. Ela oferece uma alternativa otimizada e robusta à abordagem tradicional.

LENDÔ E GRAVANDO DADOS BINÁRIOS

Até agora, lemos e gravamos apenas bytes contendo caracteres ASCII, mas é possível – na verdade, comum – ler e gravar outros tipos de dados. Por exemplo, poderíamos querer criar um arquivo contendo **ints**, **doubles** ou **shorts**. Para ler

Tabela 11-5 Um resumo dos métodos de saída definidos por **DataOutputStream**

Método de saída	Finalidade
void writeBoolean(boolean val)	Grava o boolean especificado por <i>val</i> .
void writeByte(int val)	Grava o byte de ordem inferior especificado por <i>val</i> .
void writeChar(int val)	Grava o valor especificado por <i>val</i> como um char .
void writeDouble(double val)	Grava o double especificado por <i>val</i> .
void writeFloat(float val)	Grava o float especificado por <i>val</i> .
void writeInt(int val)	Grava o int especificado por <i>val</i> .
void writeLong(long val)	Grava o long especificado por <i>val</i> .
void writeShort(int val)	Grava o valor especificado por <i>val</i> como um short .

e gravar valores binários de tipos primitivos Java, usaremos **DataInputStream** e **DataOutputStream**.

DataOutputStream implementa a interface **DataOutput**. Essa interface define métodos que gravam todos os tipos primitivos Java em um arquivo. É importante entender que esses dados são gravados usando seu formato binário interno, e não sua forma textual legível por humanos. Vários métodos de saída normalmente usados para tipos primitivos Java são mostrados na Tabela 11-5. Todos lançam uma **IOException** em caso de falha.

Este é o construtor de **DataOutputStream**. Observe que ele se baseia em uma instância de **OutputStream**.

```
DataOutputStream(OutputStream fluxoSaída)
```

Aqui, *fluxoSaída* é o fluxo em que os dados são gravados. Para gravar saídas em um arquivo, você pode usar o objeto criado por **FileOutputStream** para esse parâmetro.

DataInputStream implementa a interface **DataInput**, que fornece métodos para a leitura de todos os tipos primitivos Java. Um resumo desses métodos é mostrado na Tabela 11-6, e todos podem lançar uma **IOException**. **DataInputStream** usa uma instância de **InputStream** como base, sobrepondo-a com métodos que leem os diversos tipos de dados Java. Lembre-se de que **DataInputStream** lê os dados em seu formato binário e não em sua forma legível por humanos. O construtor de **DataInputStream** é mostrado abaixo:

```
DataInputStream(InputStream fluxoEntrada)
```

Tabela 11-6 Um resumo dos métodos de entrada definidos por **DataInputStream**

Método de entrada	Finalidade
boolean readBoolean()	Lê um boolean .
byte readByte()	Lê um byte .
char readChar()	Lê um char .
double readDouble()	Lê um double .
float readFloat()	Lê um float .
int readInt()	Lê um int .
long readLong()	Lê um long .
short readShort()	Lê um short .

Aqui, `fluxoEntrada` é o fluxo vinculado à instância de **DataInputStream** que está sendo criada. Para ler entradas em um arquivo, você pode usar o objeto criado por **FileInputStream** como esse parâmetro. Este é um programa que demonstra **DataOutputStream** e **DataInputStream**. Ele grava e depois lê vários tipos de dados em um arquivo.

```
// Grava e depois lê dados binários.

// Este código requer o JDK 7 ou posterior.

import java.io.*;

class RWData {
    public static void main(String[] args)
    {
        int i = 10;
        double d = 1023.56;
        boolean b = true;

        // Grava alguns valores.
        try (DataOutputStream dataOut =
              new DataOutputStream(new FileOutputStream("testdata")))
        {
            System.out.println("Writing " + i);
            dataOut.writeInt(i); ←

            System.out.println("Writing " + d);
            dataOut.writeDouble(d); ←

            System.out.println("Writing " + b);      Grava dados binários.
            dataOut.writeBoolean(b); ←

            System.out.println("Writing " + 12.2 * 7.4);
            dataOut.writeDouble(12.2 * 7.4); ←
        }
        catch(IOException exc) {
            System.out.println("Write error.");
            return;
        }

        System.out.println();

        // Agora, os lê.
        try (DataInputStream dataIn =
              new DataInputStream(new FileInputStream("testdata")))
        {
            i = dataIn.readInt(); ←
            System.out.println("Reading " + i);      Lê dados binários.

            d = dataIn.readDouble(); ←
            System.out.println("Reading " + d);
        }
    }
}
```

```
b = dataIn.readBoolean(); ←
System.out.println("Reading " + b);
Lê dados binários.

d = dataIn.readDouble(); ←
System.out.println("Reading " + d);
}

catch(IOException exc) {
    System.out.println("Read error.");
}
}
```

A saída do programa é mostrada aqui.

```
Writing 10  
Writing 1023.56  
Writing true  
Writing 90.28  
  
Reading 10  
Reading 1023.56  
Reading true  
Reading 90.28
```

Verificação do progresso

1. Que instrução automatiza o fechamento de um arquivo?
 2. Que fluxos são usados na leitura e gravação de dados binários?

TENTE ISTO 11-1 Utilitário de comparação de arquivos

CompFiles.java

Este projeto desenvolve um utilitário de comparação de arquivos simples, porém útil. Ele funciona abrindo os dois arquivos que serão comparados e então lendo e comparando cada conjunto de bytes correspondente. Se uma discrepância for encontrada, os arquivos são diferentes. Se o fim de cada arquivo for alcançado ao mesmo tempo e se nenhuma discrepancia for encontrada, os arquivos são iguais. Observe que ele usa a nova instrução **try-with-resources** para fechar automaticamente os arquivos.

Respostas:

1. try-with-resources
 2. **DataInputStream e DataOutputStream**

PASSO A PASSO

1. Crie um arquivo chamado **CompFiles.java**.
2. Em **CompFiles.java**, adicione o programa a seguir:

```
/*
Tente isto 11-1

Compara dois arquivos.

Para usar este programa, especifique os nomes
dos arquivos a serem comparados na linha de comando.

java CompFiles FIRST.TXT SECOND.TXT

Este código requer o JDK 7 ou posterior.
*/

import java.io.*;

class CompFiles {
    public static void main(String[] args)
    {
        int i=0, j=0;

        // Primeiro confirma se os dois arquivos foram especificados.
        if(args.length !=2 ) {
            System.out.println("Usage: CompFiles f1 f2");
            return;
        }

        // Compara os arquivos.
        try (FileInputStream f1 = new FileInputStream(args[0]);
             FileInputStream f2 = new FileInputStream(args[1]))
        {
            // Verifica o conteúdo de cada arquivo.
            do {
                i = f1.read();
                j = f2.read();
                if(i != j) break;
            } while(i != -1 && j != -1);

            if(i != j)
                System.out.println("Files differ.");
            else
                System.out.println("Files are the same.");
        } catch(IOException exc) {
            System.out.println("I/O Error: " + exc);
        }
    }
}
```

3. Para testar **CompFiles**, primeiro copie **CompFiles.java** para um arquivo chamado **temp**. Em seguida, use esta linha de comando:

```
| java CompFiles CompFiles.java temp
```

O programa relatará se os arquivos são iguais. Agora, compare **CompFiles.java** com **CopyFile.java** (mostrado anteriormente) usando a seguinte linha de comando:

```
| java CompFiles CompFiles.java CopyFile.java
```

Esses arquivos diferem e **CompFiles** relatará isso.

4. Por conta própria, tente melhorar **CompFiles** com várias opções. Por exemplo, adicione uma opção que ignore a caixa das letras. Outra ideia é fazer **CompFiles** exibir a posição dentro do arquivo que os torna diferentes.

ARQUIVOS DE ACESSO ALEATÓRIO

Até o momento, usamos *arquivos sequenciais*, que são arquivos acessados de maneira estritamente linear, um byte após o outro. No entanto, Java também nos permite acessar o conteúdo de um arquivo em ordem aleatória. Para fazer isso, usaremos **RandomAccessFile**, que encapsula um arquivo de acesso aleatório. **RandomAccessFile** não é derivada de **InputStream** ou **OutputStream**. Em vez disso, ela implementa as interfaces **DataInput** e **DataOutput**, que definem os métodos básicos de I/O. Ela também dá suporte a solicitações de posicionamento – isto é, podemos posicionar o *ponteiro de arquivo* dentro do arquivo. O construtor que usaremos é mostrado abaixo:

```
RandomAccessFile(String nomeArquivo, String acesso)
    throws FileNotFoundException
```

Aqui, o nome do arquivo é passado em *nomeArquivo*, e *acesso* determina que tipo de acesso de arquivo é permitido. Neste capítulo, os valores a seguir são usados para *acesso*: “r” e “rw”. Se for “r”, o arquivo poderá ser lido, mas não gravado. Se for “rw”, o arquivo será aberto no modo de leitura-gravação. (Outros modos são “rwd”, que faz os dados serem gravados imediatamente no dispositivo, e “rws”, que faz os dados e metadados serem gravados imediatamente no dispositivo.)

O método **seek()**, mostrado a seguir, é usado para definir a posição atual do ponteiro dentro do arquivo:

```
void seek(long novaPos) throws IOException
```

Aqui, *novaPos* especifica a nova posição, em bytes, do ponteiro a partir do início do arquivo. Após uma chamada a **seek()**, a próxima operação de leitura ou gravação ocorrerá na nova posição no arquivo.

RandomAccessFile implementa os métodos **read()** e **write()**. Também implementa as interfaces **DataInput** e **DataOutput**, ou seja, métodos de leitura e gravação dos tipos primitivos, como **readInt()** e **writeDouble()**, estão disponíveis.

Este é um exemplo que demonstra I/O de acesso aleatório. Ele grava seis **doubles** em um arquivo e então os lê em ordem não sequencial.

```
// Demonstra arquivos de acesso aleatório.
// Este código requer o JDK 7 ou posterior.

import java.io.*;

class RandomAccessDemo {
    public static void main(String[] args)
    {
        double[] data = { 19.4, 10.1, 123.54, 33.0, 87.9, 74.25 };
        double d;

        // Abre e usa um arquivo de acesso aleatório.
        try (RandomAccessFile raf = new RandomAccessFile("random.dat", "rw")) {
            // Grava valores no arquivo.
            for(int i=0; i < data.length; i++) {
                raf.writeDouble(data[i]);
            }

            // Agora, lê valores específicos
            raf.seek(0); // busca o primeiro double
            d = raf.readDouble();
            System.out.println("First value is " + d);

            raf.seek(8); // busca o segundo double
            d = raf.readDouble();
            System.out.println("Second value is " + d);

            raf.seek(8 * 3); // busca o quarto double
            d = raf.readDouble();
            System.out.println("Fourth value is " + d);

            System.out.println();

            // Agora, lê todos os outros valores.
            System.out.println("Here is every other value: ");
            for(int i=0; i < data.length; i+=2) {
                raf.seek(8 * i); // busca o i-ésimo double
                d = raf.readDouble();
                System.out.print(d + " ");
            }
        }
        catch(IOException exc) {
            System.out.println("I/O Error: " + exc);
        }
    }
}
```

Abre arquivo de acesso aleatório.

Usa `seek()` para configurar o ponteiro do arquivo.

A saída do programa é mostrada aqui:

```
First value is 19.4
Second value is 10.1
Fourth value is 33.0

Here is every other value:
19.4 123.54 87.9
```

Observe como cada valor é localizado. Como os valores double têm 8 bytes, cada valor começa a cada 8 bytes. Logo, o primeiro valor fica localizado em zero, o segundo começa no byte 8, o terceiro no byte 16 e assim por diante. Consequentemente, para ler o quarto valor, o programa busca o local 24.

Verificação do progresso

1. Que classe devemos usar para criar um arquivo de acesso aleatório?
2. Como é posicionado o ponteiro do arquivo?

Pergunte ao especialista

P Ao examinar a documentação fornecida pelo JDK, encontrei uma classe chamada **Console**. É uma classe que eu posso usar para executar I/O baseado no console?

R Uma resposta direta seria sim. A classe **Console** foi adicionada pelo JDK 6 e é usada na leitura e gravação no console. **Console** é basicamente uma classe de conveniência, porque grande parte de sua funcionalidade está disponível em **System.in** e **System.out**. No entanto, seu uso pode simplificar alguns tipos de interações com o console, principalmente na leitura de strings.

Console não fornece construtores. Em vez disso, um objeto **Console** é obtido com uma chamada a **System.console()**, mostrado aqui:

```
static Console console()
```

Se um console estiver disponível, uma referência a ele será retornada. Caso contrário, **null** será retornado. Pode não haver sempre um console disponível, como quando um programa é executado como tarefa de segundo plano. Logo, se **null** for retornado, não será possível executar a I/O de console.

Console define vários métodos que executam I/O, como **readLine()** e **printf()**. Também define um método chamado **readPassword()**, que pode ser usado na obtenção de uma senha. Ele permite que o aplicativo leia uma senha sem ecoar o que é digitado. Você também pode obter uma referência aos objetos **Reader** e **Writer** associados ao console. Em geral, **Console** é uma classe que pode ser útil em alguns tipos de aplicativos.

Respostas

1. Para criar um arquivo de acesso aleatório, use **RandomAccessFile**.
2. Para posicionar o ponteiro do arquivo, use **seek()**.

Tabela 11-7 Métodos definidos por Reader

Método	Descrição
abstract void close()	Fecha a origem da entrada. Tentativas de leitura adicionais gerarão uma IOException .
void mark(int <i>numChars</i>)	Insere uma marca no ponto atual do fluxo de entrada que permanecerá válida até <i>numChars</i> caracteres serem lidos.
boolean markSupported()	Retorna true se mark()/reset() tiverem suporte nesse fluxo.
int read()	Retorna uma representação em inteiros do próximo caractere disponível do fluxo de entrada chamador. É retornado -1 quando o fim do fluxo é alcançado.
int read(char[] <i>buffer</i>)	Tenta ler até <i>buffer.length</i> caracteres em <i>buffer</i> e retorna o número de caracteres que foram lidos com sucesso. É retornado -1 quando o fim do fluxo é alcançado na primeira tentativa de leitura.
abstract int read(char[] <i>buffer</i> , int <i>deslocamento</i> , int <i>numChars</i>)	Tenta ler até <i>numChars</i> caracteres em <i>buffer</i> começando em <i>buffer[deslocamento]</i> e retornando o número de caracteres lidos com sucesso. -1 é retornado quando o fim do fluxo é alcançado na primeira tentativa de leitura.
int read(CharBuffer <i>buffer</i>)	Tenta preencher o buffer especificado por <i>buffer</i> , retornando o número de caracteres lidos com sucesso. É retornado -1 quando o fim do fluxo é alcançado na primeira tentativa de leitura. CharBuffer é uma classe que encapsula uma sequência de caracteres, como um string.
boolean ready()	Retorna true se a próxima solicitação de entrada não tiver de esperar. Caso contrário, retorna false .
void reset()	Volta o ponteiro da entrada à marca definida anteriormente.
long skip(long <i>numChars</i>)	Ignora (isto é, salta) <i>numChars</i> caracteres da entrada, retornando o número de caracteres ignorados.

USANDO OS FLUXOS BASEADOS EM CARACTERES DA LINGUAGEM JAVA

Como as seções anteriores mostraram, os fluxos de bytes Java são ao mesmo tempo poderosos e flexíveis. Porém, não são a maneira ideal de realizar I/O baseado em caracteres. Para esse fim, Java define as classes de fluxos de caracteres. No topo da hierarquia de fluxos de caracteres, estão as classes abstratas **Reader** e **Writer**. A Tabela 11-7 mostra os métodos de **Reader** e a Tabela 11-8 mostra os de **Writer**. A maioria dos métodos pode lançar uma **IOException** em caso de erro. Os métodos definidos por essas duas classes abstratas estão disponíveis para todas as suas subclasses. Logo, formam um conjunto mínimo de funções de I/O que todos os fluxos de caracteres terão.

Entrada do console com o uso de fluxos de caracteres

Para códigos que serão internacionalizados, obter entradas do console com o uso de fluxos Java baseados em caracteres é uma maneira melhor e mais conveniente de ler

Tabela 11-8 Métodos definidos por Writer

Método	Descrição
Writer append(char <i>ch</i>)	Acrescenta <i>ch</i> ao fim do fluxo de saída chamador. Retorna uma referência ao fluxo chamador.
Writer append(CharSequence <i>chars</i>)	Acrescenta <i>chars</i> ao fim do fluxo de saída chamador. Retorna uma referência ao fluxo chamador. CharSequence é uma interface que define operações somente de leitura em uma sequência de caracteres.
Writer append(CharSequence <i>chars</i> , int <i>início</i> , int <i>fim</i>)	Acrescenta a sequência de <i>chars</i> começando em <i>início</i> e terminando em <i>fim</i> ao fim do fluxo de saída chamador. Retorna uma referência ao fluxo chamador. CharSequence é uma interface que define operações somente de leitura em uma sequência de caracteres.
abstract void close()	Fecha o fluxo de saída. Tentativas de gravação adicionais gerarão uma IOException .
abstract void flush()	Faz qualquer saída que tiver sido armazenada em buffer ser enviada para seu destino, isto é, esvazia o buffer de saídas.
void write(int <i>ch</i>)	Grava um único caractere no fluxo de saída chamador. Observe que o parâmetro é um int , o que permite que você chame write() com expressões sem ter de convertê-las novamente para char .
void write(char[] <i>buffer</i>)	Grava um array de caracteres completo no fluxo de saída chamador.
abstract void write(char[] <i>buffer</i> , int <i>deslocamento</i> , int <i>numChars</i>)	Grava um subconjunto de <i>numChars</i> caracteres a partir do array <i>buffer</i> , começando em <i>buffer[deslocamento]</i> , no fluxo de saída chamador.
void write(String <i>str</i>)	Grava <i>str</i> no fluxo de saída chamador.
void write(String <i>str</i> , int <i>deslocamento</i> , int <i>numChars</i>)	Grava um subconjunto de <i>numChars</i> caracteres do array <i>str</i> , começando no <i>deslocamento</i> especificado.

caracteres no teclado do que usar os fluxos de bytes. No entanto, já que **System.in** é um fluxo de bytes, você terá de encapsulá-lo em algum tipo de **Reader**. A melhor classe para a leitura de entradas do console é **BufferedReader**, que dá suporte a um fluxo de entrada armazenado em buffer. Contudo, você não pode construir um **BufferedReader** diretamente a partir de **System.in**. Em vez disso, primeiro deve convertê-lo em um fluxo de caracteres. Para fazê-lo, usará **InputStreamReader**, que converte bytes em caracteres. Para obter um objeto **InputStreamReader** vinculado a **System.in**, use o construtor mostrado a seguir:

```
InputStreamReader(InputStream fluxoEntrada)
```

Como **System.in** referencia um objeto de tipo **InputStream**, pode ser usado em *fluxoEntrada*.

Em seguida, usando o objeto produzido por **InputStreamReader**, construa um **BufferedReader** com o construtor mostrado abaixo:

```
BufferedReader(Reader leitorEntrada)
```

Aqui, `leitorEntrada` é o fluxo vinculado à instância de **BufferedReader** que está sendo criada. Se juntarmos tudo, a linha de código a seguir cria um **BufferedReader** conectado ao teclado.

```
BufferedReader br = new BufferedReader(new  
    InputStreamReader(System.in));
```

Após essa instrução ser executada, `br` será um fluxo baseado em caracteres vinculado ao console por `System.in`.

Lendo caracteres

Caracteres podem ser lidos a partir de `System.in` com o uso do método `read()` definido por **BufferedReader** de maneira semelhante a como são lidos com o uso de fluxos de bytes. Veja três versões de `read()` suportadas por **BufferedReader**:

```
int read() throws IOException  
int read(char[] buffer) throws IOException  
int read(char[] buffer, int deslocamento, int numChars) throws  
    IOException
```

A primeira versão de `read()` lê um único caractere. Ela retorna `-1` quando o fim do fluxo é alcançado. A segunda versão lê caracteres no fluxo de entrada e os insere em `buffer` até o array ficar cheio, o fim do arquivo ser alcançado ou um erro ocorrer. Ela retorna o número de caracteres lidos ou `-1` no fim do fluxo. A terceira versão lê entradas para `buffer` começando no local especificado por `deslocamento`. Podem ser armazenados até `numChars` caracteres. Ela retorna o número de caracteres lidos ou `-1` quando o fim do fluxo é alcançado. Todas lançam uma **IOException** em caso de erro. Na leitura a partir de `System.in`, o pressionamento de ENTER gera uma condição de fim de fluxo.

O programa a seguir demonstra `read()` lendo caracteres do console até o usuário digitar um ponto. Observe que qualquer exceção de I/O que possa ocorrer é simplesmente lançada para fora de `main()`. Como mencionado anteriormente neste capítulo, essa abordagem é comum na leitura a partir do console. É claro que você pode tratar esses tipos de erros deixando-os sob controle do programa, se quiser.

```
// Usa um BufferedReader para ler caracteres do console.  
import java.io.*;  
  
class ReadChars {  
    public static void main(String[] args)  
        throws IOException  
    {  
        char c;                                Cria um BufferedReader  
        BufferedReader br = new ←————— vinculado a System.in.  
            BufferedReader(new  
                InputStreamReader(System.in));  
  
        System.out.println("Enter characters, period to quit.");  
  
        // lê caracteres  
        do {
```

```

        c = (char) br.read();
        System.out.println(c);
    } while(c != '.');
}
}

```

Aqui está um exemplo da execução:

```

Enter characters, period to quit.
One Two.
O
n
e

T
w
o
.

```

Lendo strings

Para ler um string no teclado, use a versão de **readLine()**, que é membro da classe **BufferedReader**. Sua forma geral é mostrada a seguir:

```
String readLine() throws IOException
```

Ela retorna um objeto **String** contendo os caracteres lidos. Quando é feita uma tentativa de leitura no fim do fluxo, ela retorna nulo.

O programa abaixo demonstra **BufferedReader** e o método **readLine()**. Ele lê e exibe linhas de texto até a palavra “stop” ser inserida.

```

// Lê um string no console usando um BufferedReader.
import java.io.*;

class ReadLines {
    public static void main(String[] args)
        throws IOException
    {
        // cria um BufferedReader usando System.in
        BufferedReader br = new BufferedReader(new
                                         InputStreamReader(System.in));
        String str;

        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        do {
            str = br.readLine(); ← Usa o método readLine() de BufferedReader
            System.out.println(str);     para ler uma linha de texto.
        } while(!str.equals("stop"));
    }
}

```

Saída no console com o uso de fluxos de caracteres

Embora ainda seja permitido usar **System.out** em Java para gravações no console, seu uso mais recomendado é para fins de depuração ou para exemplos de programa como os encontrados neste livro. Para programas do mundo real, o melhor método de gravação no console quando se usa Java é com um fluxo **PrintWriter**. **PrintWriter** é uma das classes baseadas em caracteres. Como explicado, usar uma classe baseada em caracteres para a saída no console facilita a internacionalização do programa.

PrintWriter define vários construtores. Usaremos o mostrado abaixo:

```
PrintWriter(OutputStream fluxoSaida, boolean liberaNovaLinha)
```

Aqui, *fluxoSaida* é um objeto de tipo **OutputStream** e *liberaNovaLinha* controla se Java descarregará o fluxo de saída sempre que um método **println()** for chamado. Se *liberaNovaLinha* for **true**, a descarga automática ocorrerá. Se for **false**, a descarga não será automática.

PrintWriter dá suporte aos métodos **print()** e **println()** para todos os tipos, inclusive **Object**. Logo, você pode usar esses métodos da mesma maneira como seriam usados com **System.out**. Se um argumento não for de tipo primitivo, os métodos de **PrintWriter** chamarão o método **toString()** do objeto e então exibirão o resultado.

Para gravar no console usando um **PrintWriter**, especifique **System.out** como fluxo de saída e descarregue o fluxo após cada chamada a **println()**. Por exemplo, esta linha de código cria um **PrintWriter** conectado à saída no console:

```
| PrintWriter pw = new PrintWriter(System.out, true);
```

O aplicativo a seguir ilustra o uso de um **PrintWriter** para o tratamento da saída no console.

```
// Demonstra PrintWriter.
import java.io.*;
public class PrintWriterDemo {
    public static void main(String[] args) {
        PrintWriter pw = new PrintWriter(System.out, true);
        int i = 10;
        double d = 123.65;

        pw.println("Using a PrintWriter.");
        pw.println(i);
        pw.println(d);

        pw.println(i + " + " + d + " is " + (i+d));
    }
}
```

A saída desse programa é:

```
Using a PrintWriter.
10
123.65
10 + 123.65 is 133.65
```

Lembre-se de que não é errado usar **System.out** para gravar saídas de texto simples no console quando estamos aprendendo Java ou depurando programas. No entanto, o uso de **PrintWriter** facilita a internacionalização de aplicativos do mundo real. Já que não há o que se ganhar com o uso de um **PrintWriter** nos exemplos de programas mostrados neste livro, por conveniência, continuaremos a utilizar **System.out** para gravar no console.

Verificação do progresso

1. Quais são as principais classes de fluxo baseado em caracteres?
2. Para ler no console, você deve abrir que tipo de leitor?
3. Para gravar no console, que tipo de gravador deve ser aberto?

I/O DE ARQUIVO COM O USO DE FLUXOS DE CARACTERES

Embora o tratamento de arquivos orientado a bytes seja mais comum, é possível usar fluxos baseados em caracteres para esse fim. A vantagem dos fluxos de caracteres é que eles operam diretamente sobre os caracteres Unicode. Logo, se quisermos armazenar texto Unicode, certamente os fluxos de caracteres serão a melhor opção. Em geral, para executar I/O de arquivo baseado em caracteres, usamos as classes **FileReader** e **FileWriter**.

Usando um **FileWriter**

FileWriter cria um objeto **Writer** que podemos usar para fazer gravações em um arquivo. Os dois construtores mais usados são mostrados abaixo:

```
FileWriter(String nomeArquivo) throws IOException
FileWriter(String nomeArquivo, boolean incluir) throws IOException
```

Aqui, *nomeArquivo* é o nome de um arquivo. Se *incluir* for igual a **true**, a saída será acrescida ao fim do arquivo. Caso contrário, o arquivo será sobreposto. Os dois construtores lançam uma **IOException** em caso de falha. **FileWriter** é derivada de **OutputStreamWriter** e **Writer**, logo, tem acesso aos métodos definidos por essas classes.

A seguir, temos um utilitário ‘teclado para disco’ simples que lê linhas de texto inseridas a partir do teclado e grava-as em um arquivo chamado “test.txt”. O texto é lido até o usuário inserir a palavra “stop”. Um **FileWriter** é usado para as gravações no arquivo.

```
// Utilitário 'teclado para disco' simples que demonstra um FileWriter.
// Este código requer o JDK 7 ou posterior.

import java.io.*;
```

Respostas:

1. **Reader** e **Writer** estão no topo da hierarquia de classes de fluxos baseados em caracteres.
2. Para ler no console, é preciso abrir um **BufferedReader**.
3. Para gravar no console, um **PrintWriter** deve ser aberto.

```

class KtoD {
    public static void main(String[] args)
    {

        String str;
        BufferedReader br =
            new BufferedReader(
                new InputStreamReader(System.in));

        System.out.println("Enter text ('stop' to quit).");

        try (FileWriter fw = new FileWriter("test.txt")) ← Cria um FileWriter.
        {
            do {
                System.out.print(": ");
                str = br.readLine();

                if(str.compareTo("stop") == 0) break;

                str = str + "\r\n"; // adiciona nova linha
                fw.write(str); ← Grava strings no arquivo.
            } while(str.compareTo("stop") != 0);
            catch(IOException exc) {
                System.out.println("I/O Error: " + exc);
            }
        }
    }
}

```

Usando um FileReader

A classe **FileReader** cria um objeto **Reader** que pode ser usado na leitura do conteúdo de um arquivo. O construtor mais usado é mostrado abaixo:

```
FileReader(String nomeArquivo) throws FileNotFoundException
```

Aqui, *nomeArquivo* é o nome de um arquivo. O construtor lança uma **FileNotFoundException** se o arquivo não existir. **FileReader** é derivada de **InputStreamReader** e **Reader**. Logo, tem acesso aos métodos definidos por essas classes.

O programa a seguir cria um utilitário ‘disco para tela’ simples que lê um arquivo de texto chamado “test.txt” e exibe seu conteúdo na tela, portanto, ele complementa o utilitário ‘teclado para disco’ mostrado na seção anterior.

```

// Utilitário 'disco para tela' simples que demonstra um FileReader.
// Este código requer o JDK 7 ou posterior.

import java.io.*;

class DtoS {
    public static void main(String[] args) {
        String s;

```

```

// Cria e usa um FileReader encapsulado em um BufferedReader.
try (BufferedReader br = new BufferedReader(new FileReader("test.txt")))
{
    while((s = br.readLine()) != null) { ← Lê linhas no arquivo
        System.out.println(s);           e as exibe na tela.
    }
} catch(IOException exc) {
    System.out.println("I/O Error: " + exc);
}
}

```

Nesse exemplo, observe que **FileReader** está encapsulado em um **BufferedReader**. Logo, ele tem acesso a **readLine()**. Além disso, o fechamento do **BufferedReader**, nesse caso representado por **br**, fecha automaticamente o arquivo.

Verificação do progresso

1. Que classe é usada na leitura de caracteres em um arquivo?
2. Que classe é usada na gravação de caracteres em um arquivo?

Pergunte ao especialista

P Ouvi falar de outro pacote de I/O chamado NIO. Pode me falar sobre ele?

R Originalmente chamado de *New I/O*, o pacote NIO foi adicionado a Java pelo JDK 1.4. Ele dá suporte à abordagem de operações de I/O baseadas em canais. As classes NIO ficam no pacote **java.nio** e em seus pacotes subordinados, como **java.nio.channels** e **java.nio.charset**. NIO se baseia em dois itens básicos: *buffers* e *canais*. O buffer armazena dados. O canal representa uma conexão aberta com um dispositivo de I/O, como um arquivo ou um soquete. Em geral, para usar o novo sistema de I/O, temos que obter um canal com um dispositivo de I/O e um buffer para armazenar dados. Então operamos com o buffer, inserindo ou exibindo dados quando necessário.

A partir do JDK 7, NIO sofreu melhorias profundas, tanto que o termo *NIO.2* costuma ser usado. As melhorias incluem três pacotes novos (**java.nio.file**, **java.nio.file.attribute** e **java.nio.file.spi**); várias classes, interfaces e métodos novos e o suporte direto a I/O baseada em fluxos. Os acréscimos expandiram as maneiras como NIO pode ser usado, principalmente com arquivos.

É importante entender que NIO não substitui as classes de I/O encontradas em **java.io**, que estão sendo discutidas neste capítulo. Em vez disso, as classes NIO foram projetadas para complementar o sistema de I/O padrão, oferecendo uma abordagem alternativa, que pode ser benéfica em algumas circunstâncias.

Respostas:

1. Para ler caracteres, use um **FileReader**.
2. Para gravar caracteres, use um **FileWriter**.

File

Antes de sairmos do tópico I/O, há mais uma classe que é preciso discutir. Trata-se da classe **File**. Em vez de operar com fluxos, **File** lida diretamente com arquivos e com o sistema de arquivos. Isto é, a classe **File** não especifica como as informações serão recuperadas ou armazenadas em arquivos; ela descreve as propriedades de um arquivo. Um objeto **File** é usado para obter ou tratar as informações associadas a um arquivo em disco, como as permissões, a hora, a data e o caminho do diretório, e navegar nas hierarquias de subdiretórios.

File define vários construtores, inclusive os dois mostrados aqui:

```
File(String caminho)  
File(String caminhoDiretório, String nomearquivo)
```

Na primeira forma, *caminho* especifica o caminho completo do arquivo ou diretório. Na segunda, *caminhoDiretório* é o nome do caminho de um diretório e *nomearquivo* é o nome do arquivo ou subdiretório. Como exemplo, as linhas a seguir criam dois objetos **File** chamados **myFileA** e **myFileB**. O primeiro é construído com um caminho (que inclui um nome de arquivo) como único argumento. O segundo inclui dois argumentos – o caminho e o nome do arquivo.

```
File myFileA = new File("/javafiles/MyClass.java");  
File myFileB = new File("/javafiles", "MyClass.java")
```

Nota: Em geral, Java faz o que é certo com os separadores de caminho usados nas convenções do UNIX e do Windows. Se você usar uma barra (/) em uma versão de Java no Windows, o caminho será resolvido corretamente. Lembre-se, se estiver usando a convenção do Windows, que é uma barra invertida (\), terá que usar sua sequência de escape (\\\) dentro de um string. Por conveniência, os exemplos deste capítulo usam barras comuns.

Obtendo as propriedades de um arquivo

File define muitos métodos que obtêm as propriedades padrão de um objeto **File**. Alguns são mostrados abaixo:

Método	Descrição
boolean canRead()	Retorna true se o arquivo puder ser lido.
boolean canWrite()	Retorna true se o arquivo puder ser gravado.
boolean exists()	Retorna true se o arquivo existir.
String getAbsolutePath()	Retorna o caminho absoluto do arquivo.
String getName()	Retorna o nome do arquivo.
String getParent()	Retorna o nome do diretório pai do arquivo ou nulo se não existir um pai.
boolean isAbsolute()	Retorna true se o caminho for absoluto. Retorna false se o caminho for relativo.
boolean isDirectory()	Retorna true se o arquivo for um diretório.

boolean isFile()	Retorna true se o arquivo for um arquivo “comum”. Retorna false se o arquivo for um diretório ou algum outro objeto que não seja um arquivo.
boolean isHidden()	Retorna true se o arquivo chamador estiver oculto. Caso contrário, retorna false .
long length()	Retorna o tamanho do arquivo, em bytes.

O exemplo a seguir demonstra esses métodos de **File**. Ele presume que um diretório chamado **javafiles** ocorre no diretório raiz e contém um arquivo chamado **MyClass.java**.

```
// Obtém informações sobre um arquivo.
import java.io.*;

class FileDemo {
    public static void main(String[] args) {
        File myFile = new File("/javafiles/MyClass.java");

        System.out.println("File Name: " + myFile.getName());
        System.out.println("Path: " + myFile.getPath());
        System.out.println("Abs Path: " + myFile.getAbsolute());
        System.out.println("Parent: " + myFile.getParent());
        System.out.println(myFile.exists() ? "exists" : "does not exist");
        System.out.println(myFile.isHidden() ? "is hidden" :
                           "is not hidden");
        System.out.println(myFile.canWrite() ? "is writeable" :
                           "is not writeable");
        System.out.println(myFile.canRead() ? "is readable" :
                           "is not readable");
        System.out.println("is " + (myFile.isDirectory() ? "" :
                           "not" + " a directory"));
        System.out.println(myFile.isFile() ? "is normal file" :
                           "might be a named pipe");
        System.out.println(myFile.isAbsolute() ? "is absolute" :
                           "is not absolute");
        System.out.println("File size: " + myFile.length() + " Bytes");
    }
}
```

O programa produzirá uma saída como essa:

```
File Name: MyClass.java
Path: \javafiles\MyClass.java
Abs Path: C:\javafiles\MyClass.java
Parent: \javafiles
exists
is not hidden
is writeable
is readable
is not a directory
is normal file
is not absolute
File size: 369 Bytes
```

Obtendo uma listagem de diretório

Um diretório é um arquivo que contém uma lista de outros arquivos e diretórios. Quando você criar um objeto **File** que seja um diretório, o método **isDirectory()** retornará **true**. Nesse caso, poderá obter uma lista dos arquivos do diretório, e uma maneira de fazê-lo é chamando o método **list()** nesse objeto. Ele tem duas formas. A primeira é mostrada abaixo:

```
String[ ] list( )
```

A lista de arquivos é retornada em um array de objetos **String**.

O programa mostrado aqui ilustra como podemos usar **list()** para examinar o conteúdo de um diretório:

```
// Usando diretórios.  
import java.io.*;  
  
class DirList {  
    public static void main(String[] args) {  
        String dirname = "/javafiles";  
        File myDir = new File(dirname);  
  
        if (myDir.isDirectory()) {  
            System.out.println("Directory of " + dirname);  
            String[] s = myDir.list();  
  
            for (int i=0; i < s.length; i++) {  
                File f = new File(dirname + "/" + s[i]);  
                if (f.isDirectory()) {  
                    System.out.println(s[i] + " is a directory");  
                } else {  
                    System.out.println(s[i] + " is a file");  
                }  
            }  
        } else {  
            System.out.println(dirname + " is not a directory");  
        }  
    }  
}
```

Abaixo temos um exemplo da saída do programa. (É claro que a saída que você verá será diferente, baseada no que houver no diretório.)

```
Directory of /javafiles  
examples is a directory  
MyClass.class is a file  
MyClass.java is a file  
ReadMe.txt is a file  
SampleClass.class is a file  
SampleClass.java is a file  
temp is a directory
```

Usando FilenameFilter

Às vezes podemos querer limitar o número de arquivos retornados pelo método `list()` para incluir somente os arquivos que correspondam a um determinado padrão de nome de arquivo, ou *filtro*. Para fazer isso, podemos usar uma segunda forma de `list()`, mostrada aqui:

```
String[ ] list(FilenameFilter ObjFF)
```

Nessa versão, `ObjFF` é um objeto de uma classe que implementa a interface **FilenameFilter**.

FilenameFilter define só um método, `accept()`, que é chamado uma vez para cada arquivo de uma lista. Sua forma geral é essa:

```
boolean accept(File diretório, String nomearquivo)
```

O método `accept()` retorna `true` se o arquivo especificado por `nomearquivo` do diretório especificado por `diretório` tiver que ser incluído na lista e retorna `false` se o arquivo tiver que ser excluído.

A classe **FilterExt**, mostrada a seguir, implementa **FilenameFilter**. Ela será usada para modificar o programa anterior e restringir a visibilidade dos nomes de arquivo retornados por `list()` a arquivos com nomes que terminem com a extensão especificada na construção do objeto.

```
import java.io.*;

public class FilterExt implements FilenameFilter {
    String ext;

    public FilterExt(String ext) {
        this.ext = "." + ext;
    }

    public boolean accept(File dir, String name) {
        return name.endsWith(ext);
    }
}
```

O programa de listagem de diretório modificado é mostrado abaixo. Agora ele só exibirá arquivos que usem a extensão `.java`.

```
// Diretório de arquivos .java.
import java.io.*;

class DirListFiltered {
    public static void main(String[] args) {
        FilenameFilter only = new FilterExt("java");
        String dirname = "/javafiles";
        File myDir = new File(dirname);

        if (myDir.isDirectory()) {
            System.out.println("Java source files in " + dirname);
```

```

        String[] s = myDir.list(only);

        for (int i=0; i < s.length; i++) {
            System.out.println(s[i]);
        }
    }
}
}

```

Quando executado no mesmo diretório mostrado no exemplo anterior, a saída a seguir é produzida:

```

Java source files in /javafiles
MyClass.java
SampleClass.java

```

A alternativa listFiles()

Há uma variação do método **list()**, chamada **listFiles()**, que você pode achar útil. Suas três formas são essas:

```

File[ ] listFiles()
File[ ] listFiles(FilenameFilter ObjFF)
File[ ] listFiles(FileFilter ObjF)

```

Esses métodos retornam a lista de arquivos como um array de objetos **File** em vez de strings. O primeiro método retorna todos os arquivos e o segundo retorna os arquivos que atendem ao **FilenomeFilter** especificado. Exceto por retornar um array de objetos **File**, essas duas versões de **listFiles()** funcionam como os métodos **list()** que equivalem a elas.

A terceira versão de **listFiles()** retorna os arquivos com nomes de caminho que atendam ao **FileFilter** especificado. **FileFilter** define só um método, **accept()**, que é chamado uma vez para cada arquivo de uma lista. Sua forma geral é a seguinte:

```
boolean accept(File caminho)
```

O método **accept()** retorna **true** se o arquivo especificado por *caminho* tiver que ser incluído na lista, e **false** se o arquivo tiver que ser excluído.

Vários métodos utilitários de File

Além dos métodos **list()** e **listFiles()** que acabamos de descrever, **File** tem muitos outros métodos utilitários que permitem a execução de várias ações em um arquivo ou no sistema de arquivos. Um resumo é mostrado na Tabela 11-9. Um método particularmente interessante é **getFreeSpace()**. Ele retorna o número de bytes de espaço livre restante na partição atual do dispositivo de armazenamento. Aqui está um programa que o mostra em ação:

```

// Mostra o espaço livre na partição atual da unidade.
import java.io.*;

class FreeSpace {
    public static void main(String[] args) {

```

```

    File myFile = new File("\\");
    System.out.println("Free Space: " + myFile.getFreeSpace());
}
}

```

O JDK 7 adicionou um novo método a **File** chamado **toPath()**, que é mostrado abaixo:

Path toPath()

O método **toPath()** retorna um objeto **Path** que representa o arquivo encapsulado pelo objeto **File** chamador. (Em outras palavras, **toPath()** converte um **File** em um **Path**.)

Tabela 11-9 Um resumo dos métodos utilitários fornecidos por **File**

Método	Descrição
boolean delete()	Exclui o arquivo especificado pelo objeto chamador. Retorna true se o arquivo for excluído e false se ele não puder ser removido.
void deleteOnExit()	Remove o arquivo associado ao objeto chamador quando a Máquina Virtual Java é encerrada.
long getFreeSpace()	Retorna o número de bytes de armazenamento livres disponíveis na partição associada ao objeto chamador.
long getTotalSpace()	Retorna a capacidade de armazenamento da partição associada ao objeto chamador.
long getUsableSpace()	Retorna o número de bytes de armazenamento livres e usáveis disponíveis na partição associada ao objeto chamador.
long lastModified()	Obtém o carimbo de hora no arquivo chamador. O valor retornado é o número de milissegundos a partir de primeiro de janeiro de 1970, hora universal coordenada (UTC, Coordinated Universal Time). Se não houver um carimbo de hora disponível, zero será retornado.
boolean mkdir()	Cria o diretório especificado pelo objeto chamador. Retorna true se o diretório for criado e false se ele não puder ser criado. Pode ocorrer uma falha por várias razões, como o caminho especificado no objeto File já existir ou o diretório não poder ser criado porque o caminho inteiro ainda não existe.
boolean mkdirs()	Cria o diretório e todos os diretórios pais necessários especificados pelo objeto chamador. Retorna true se o caminho inteiro for criado; caso contrário, retorna false .
boolean renameTo(File novoNome)	Renomeia com <i>novoNome</i> o arquivo especificado pelo objeto chamador. Retorna true se for bem-sucedido e false se o arquivo não puder ser renomeado.
boolean setLastModified(long milisseg)	Define o carimbo de hora do arquivo chamador com o especificado por <i>milisseg</i> , que contém o número de milissegundos a partir de primeiro de janeiro de 1970, hora universal coordenada (UTC, Coordinated Universal Time).
boolean setReadOnly()	Define o arquivo como somente leitura.
boolean setWritable(boolean como)	Se <i>como</i> for true, o arquivo será definido como gravável. Se for false, ele será definido como somente leitura. Retorna true se o status do arquivo for modificado e false se o status de gravação não puder ser alterado.

Path é uma interface nova adicionada pelo JDK 7. Ela fica no pacote **java.nio.file** e faz parte de NIO. Logo, **toPath()** forma uma ponte entre a classe **File** e a nova interface **Path**.

Verificação do progresso

1. **File** abre um arquivo?
2. Que método de **File** é usado para determinar se um arquivo está oculto?
3. Que método é usado para listar os arquivos de um diretório?

USANDO OS ENCAPSULADORES DE TIPOS DA LINGUAGEM JAVA PARA CONVERTER STRINGS NUMÉRICOS

Antes de concluirmos este capítulo, examinaremos uma técnica útil na leitura de strings numéricos. Como você sabe, o método **println()** de Java fornece uma maneira conveniente de exibirmos vários tipos de dados no console, inclusive valores numéricos de tipos internos, como **int** e **double**. Logo, **println()** converte automaticamente valores numéricos para sua forma legível por humanos. No entanto, métodos como **read()** não fornecem uma funcionalidade paralela que leia e converta um string contendo um valor numérico para seu formato binário interno. Por exemplo, não há uma versão de **read()** que leia um string como “100” e o converta automaticamente para o valor numérico correspondente que possa ser armazenado em uma variável **int**. Em vez disso, Java fornece várias outras maneiras de executar essa tarefa. A que examinaremos aqui usa os *encapsuladores de tipos* Java.

Os encapsuladores de tipos Java são classes que encapsulam, ou *empacotam*, os tipos primitivos. Eles são necessários porque os tipos primitivos não são objetos. Isso limita seu uso. Por exemplo, um tipo primitivo não pode ser passado por referência. Para suprir essa necessidade, Java fornece classes que correspondem a cada um dos tipos primitivos.

Os encapsuladores de tipos são **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character** e **Boolean**. Essas classes oferecem um amplo conjunto de métodos que nos permite integrar totalmente os tipos primitivos à hierarquia de objetos Java. Como benefício adicional, os encapsuladores numéricos também definem métodos que convertem um string numérico no equivalente binário correspondente. Vários desses métodos de conversão são mostrados aqui. Todos retornam um valor binário correspondente ao string.

Encapsulador	Método de conversão
Double	static double parseDouble(String str) throws NumberFormatException
Float	static float parseFloat(String str) throws NumberFormatException
Long	static long parseLong(String str) throws NumberFormatException

Respostas:

1. Não.
2. **isHidden()**
3. **list()** ou **listFiles()**

<code>Integer</code>	<code>static int parseInt(String str) throws NumberFormatException</code>
<code>Short</code>	<code>static short parseShort(String str) throws NumberFormatException</code>
<code>Byte</code>	<code>static byte parseByte(String str) throws NumberFormatException</code>

Os encapsuladores de inteiros também oferecem um segundo método de análise que nos permite especificar a base numérica.

Os métodos de análise fornecem uma maneira fácil de converter um valor numérico, lido como um string a partir do teclado ou de um arquivo de texto, em seu formato interno apropriado. Por exemplo, o programa a seguir demonstra `parseInt()` e `parseDouble()`. Ele calcula a média de uma lista de números inseridos pelo usuário. Primeiro, pergunta ao usuário quantos valores entrarão no cálculo da média. Em seguida, lê esse número usando `readLine()` e usa `parseInt()` para converter o string em um inteiro. Depois, insere os valores, usando `parseDouble()` para converter os strings em seus equivalentes `double`.

```
// Este programa calcula a média de uma lista de números inseridos pelo
usuário.

import java.io.*;

class AvgNums {
    public static void main(String[] args)
        throws IOException
    {
        // cria um BufferedReader usando System.in
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        String str;
        int n;
        double sum = 0.0;
        double avg, t;

        System.out.print("How many numbers will you enter: ");
        str = br.readLine();
        try {
            n = Integer.parseInt(str); ←———— Converte o string em int.
        }
        catch(NumberFormatException exc) {
            System.out.println("Invalid format");
            n = 0;
        }

        System.out.println("Enter " + n + " values.");
        for(int i=0; i < n ; i++) {
            System.out.print(": ");
            str = br.readLine();
            try {
                t = Double.parseDouble(str); ←———— Converte o string em double.
            } catch(NumberFormatException exc) {
                System.out.println("Invalid format");
            }
        }
    }
}
```

```

        t = 0.0;
    }
    sum += t;
}
avg = sum / n;
System.out.println("Average is " + avg);
}
}

```

Aqui está um exemplo da execução:

```

How many numbers will you enter: 5
Enter 5 values.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Average is 3.3

```

Pergunte ao especialista

P O que mais as classes encapsuladoras de tipos primitivos podem fazer?

R Os encapsuladores de tipos primitivos fornecem vários métodos que ajudam a integrar os tipos primitivos à hierarquia de objetos. Por exemplo, muitos mecanismos de armazenamento fornecidos pela biblioteca Java, inclusive mapeamentos, listas e conjuntos, só trabalham com objetos. Logo, para armazenarmos um `int` em uma lista, ele deve ser encapsulado em um objeto. Além disso, todos os encapsuladores de tipos têm o método `compareTo()`, que compara o valor contido dentro do encapsulador, e o método `equals()`, que vê se dois valores são iguais, além de métodos que retornam o valor do objeto de várias formas. O tópico dos encapsuladores de tipos será retomado no Capítulo 13, quando o *autoboxing* for discutido.

TENTE ISTO 11-2 Criando um sistema de ajuda baseado em disco

`FileHelp.java`

Na seção Tente isto 4-1, criamos uma classe `Help` que exibia informações sobre as instruções de controle Java. Naquela implementação, as informações de ajuda estavam armazenadas dentro da própria classe e o usuário selecionava a ajuda em um menu de opções numeradas.

Embora essa abordagem funcione perfeitamente, com certeza não é a maneira ideal de criar um sistema de ajuda. Por exemplo, para que as informações de ajuda possam ser expandidas ou alteradas, o código-fonte do programa deve ser modificado. Além disso, a seleção do tópico por número e não por nome é tediosa e inadequada para listas de tópicos longas. Aqui, corrigiremos essas deficiências criando um sistema de ajuda baseado em disco.

O sistema de ajuda baseado em disco armazena informações em um arquivo de ajuda, o qual é um arquivo de texto padrão que pode ser alterado ou expandido à vontade, sem alteração do programa de ajuda. O usuário obtém ajuda sobre um tópico digitando seu nome. O sistema de ajuda procura o tópico no arquivo. Se ele for encontrado, informações serão exibidas.

PASSO A PASSO

1. Crie o arquivo de ajuda que será usado pelo sistema. O arquivo de ajuda é um arquivo de texto padrão organizado desta forma:

```
#nome-tópico1
info tópico
```

```
#nome-tópico2
info tópico
```

```
.
```

```
.
```

```
#nome-tópicoN
info tópico
```

O nome de cada tópico deve ser precedido por um símbolo # e deve estar em uma linha própria. Anteceder o nome dos tópicos com um símbolo # permite que o programa encontre rapidamente o início de cada tópico. Após o nome do tópico, teremos algum número de linhas de informações sobre ele. No entanto, é preciso que haja uma linha em branco entre o fim das informações de um tópico e o início do próximo tópico.

Aqui está um arquivo de ajuda simples que você pode usar para testar o sistema de ajuda baseado em disco. Ele armazena informações sobre instruções de controle Java.

```
|#if
if(condition) statement;
else statement;

#switch
switch(expression) {
    case constant:
        statement sequence
        break;
    // ...
}

#for
for(init; condition; iteration) statement;

}while
while(condition) statement;
```

```

#do
do {
    statement;
} while (condition);

#break
break; or break label;

#continue
continue; or continue label;

```

Chame esse arquivo de **helpfile.txt**.

2. Crie um arquivo chamado **FileHelp.java**
3. Comece a criar a nova classe **Help** com estas linhas de código:

```

class Help {
    String helpfile; // nome do arquivo de ajuda

    Help(String fname) {
        helpfile = fname;
    }

```

O nome do arquivo de ajuda é passado para o construtor de **Help** e armazenado na variável de instância **helpfile**. Já que cada instância de **Help** terá sua própria cópia de **helpfile**, cada uma pode usar um arquivo diferente. Logo, você pode criar diferentes conjuntos de arquivos de ajuda para conjuntos de tópicos distintos.

4. Adicione o método **helpOn()** mostrado aqui à classe **Help**. Esse método recupera ajuda sobre o tópico especificado.

```

// Exibe ajuda sobre um tópico.
boolean helpOn(String what) {
    int ch;
    String topic, info;

    // Abre o arquivo de ajuda.
    try (BufferedReader helpRdr =
        new BufferedReader(new FileReader(helpfile)))
    {
        do {
            // lê caracteres até um # ser encontrado
            ch = helpRdr.read();

            // agora, vê se os tópicos coincidem
            if(ch == '#') {
                topic = helpRdr.readLine();
                topic = topic.trim(); // remove o espaço em branco
                // inicial e final

                if(what.compareTo(topic) == 0) { // tópico encontrado
                    do {

```

```

        info = helpRdr.readLine();
        if(info != null) System.out.println(info);
    } while((info != null) &&
            (info.trim().compareTo("") != 0));
    return true;
}
}
} while(ch != -1);
}
catch(IOException exc) {
    System.out.println("Error accessing help file.");
    return false;
}
return false; // tópico não encontrado
}

```

A primeira coisa que devemos observar é que **helpOn()** trata ele próprio todas as exceções de I/O possíveis e não inclui uma cláusula **throws**. Ao tratar suas próprias exceções, ele impede que essa carga seja passada para todos os códigos que o usam. Logo, os outros códigos podem simplesmente chamar **helpOn()** sem ter de encapsular essa chamada em um bloco **try/catch**.

O arquivo de ajuda é aberto com o uso de um **FileReader** que está encapsulado em um **BufferedReader**. Já que o arquivo de ajuda contém texto, o uso de um fluxo de caracteres permite que o sistema de ajuda seja internacionalizado com mais eficiência.

O método **helpOn()** funciona assim: um string contendo o nome do tópico é passado no parâmetro **what**, e o arquivo de ajuda é então aberto. Em seguida, o arquivo é pesquisado em busca de uma correspondência entre **what** e um de seus tópicos. Lembre-se, no arquivo, cada tópico é precedido por um símbolo #, logo, o laço de busca procura símbolos #. Quando encontra, verifica se o tópico que vem após o símbolo # coincide com o passado em **what**. Se coincidir, as informações associadas a esse tópico serão exibidas. Se uma ocorrência for encontrada, **helpOn()** retornará **true**. Caso contrário, retornará **false**.

Mais uma coisa: observe que **helpOn()** usa outro método de **String**, chamado **trim()**. Ele remove qualquer lacuna em branco inicial ou final (como espaços e tabulações) do string.

5. A classe **Help** também fornece um método chamado **getSelection()**. Ele solicita um tópico e retorna o string inserido pelo usuário.

```

// Acessa um tópico da Ajuda.
String getSelection() {
    String topic = "";

    BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));

```

```

    System.out.print("Enter topic: ");
    try {
        topic = br.readLine();
    }
    catch(IOException exc) {
        System.out.println("Error reading console.");
    }
    return topic;
}

```

Esse método cria um **BufferedReader** vinculado a **System.in**. Em seguida, solicita o nome de um tópico, lê o tópico e retorna-o para o chamador.

6. O sistema de ajuda baseado em disco completo é mostrado aqui:

```

/*
Tente isto 11-2

Programa de ajuda que usa um arquivo em disco
para armazenar informações de ajuda.

Este código requer o JDK 7 ou posterior.
*/

import java.io.*;

/* A classe Help abre um arquivo de ajuda,
   procura um tópico e exibe as informações
   associadas a esse tópico.
   Observe que ela mesma trata todas as exceções de I/O,
   evitando ser preciso chamar um código
   que faça isso. */
class Help {
    String helpfile; // nome do arquivo de ajuda

    Help(String fname) {
        helpfile = fname;
    }

    // Exibe ajuda sobre um tópico.
    boolean helpOn(String what) {
        int ch;
        String topic, info;

        // Abre o arquivo de ajuda.
        try (BufferedReader helpRdr =
              new BufferedReader(new FileReader(helpfile)))
        {
            do {
                // lê caracteres até um # ser encontrado
                ch = helpRdr.read();

```

```
// agora, vê se os tópicos coincidem
if(ch == '#') {
    topic = helpRdr.readLine();
    topic = topic.trim(); // remove o espaço em branco
                           // inicial e final

    if(what.compareTo(topic) == 0) { // tópico encontrado
        do {
            info = helpRdr.readLine();
            if(info != null) System.out.println(info);
        } while((info != null) &&
                 (info.trim().compareTo("") != 0));
        return true;
    }
}
} while(ch != -1);
}

catch(IOException exc) {
    System.out.println("Error accessing help file.");
    return false;
}
return false; // tópico não encontrado
}

// Acessa um tópico da Ajuda.
String getSelection() {
    String topic = "";

    BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));

    System.out.print("Enter topic: ");
    try {
        topic = br.readLine();
    }
    catch(IOException exc) {
        System.out.println("Error reading console.");
    }
    return topic;
}
}

// Demonstra o sistema de ajuda baseado em arquivo.
class FileHelp {
    public static void main(String[] args) {
        Help hlpobj = new Help("helpfile.txt");
        String topic;
```

```

System.out.println("Try the help system. " +
                    "Enter 'stop' to end.");
do {
    topic = hlpobj.getSelection();

    if (!hlpobj.helpOn(topic))
        System.out.println("Topic not found.\n");

    } while (topic.compareTo("stop") != 0);
}
}

```

Pergunte ao especialista

P Além dos métodos **parse** definidos pelos encapsuladores de tipos primitivos, há outra maneira fácil de converter um string numérico inserido pelo teclado para o formato binário equivalente?

R Sim! Outra maneira de converter um string numérico em seu formato binário interno é usando um dos métodos definidos pela classe **Scanner**, empacotada em **java.util**. Adicionada pelo JDK 5, **Scanner** lê a entrada formatada (isto é, legível por humanos) e a converte para sua forma binária. **Scanner** pode ser usada na leitura de entradas de várias fontes, inclusive do console e de arquivos. Portanto, você pode usá-la para ler um string numérico inserido pelo teclado e atribuir seu valor a uma variável. **Scanner** será descrita com detalhes no Capítulo 24, quando **java.util** for examinado. No entanto, o exemplo a seguir ilustra seu uso básico de leitura de entradas do teclado. Se quiser, teste-a agora.

Para usar **Scanner** na leitura a partir do teclado, primeiro você deve criar um objeto **Scanner** vinculado à entrada do console. Para fazê-lo, use um dos construtores de **Scanner**, como mostrado aqui:

```
| Scanner conin = new Scanner(System.in);
```

Após essa linha ser executada, **conin** poderá ser usada na leitura de entradas do teclado.

Uma vez que você tiver criado um **Scanner**, é só usá-lo para ler entradas numéricas. Veja o procedimento geral:

1. Determine se um tipo específico de entrada está disponível chamando um dos métodos **hasNextX** de **Scanner**, no qual X é o tipo de dado desejado.
2. Se a entrada estiver disponível, leia-a chamando um dos métodos **nextX** de **Scanner**.

Como o exemplo anterior indica, **Scanner** define dois conjuntos de métodos que nos permitem ler entradas. O primeiro é composto pelos métodos **hasNext**. Ele inclui métodos como **hasNextInt()** e **hasNextDouble()**. Todos os métodos **hasNext** retornam **true** quando o tipo de dado desejado é o próximo item disponível no fluxo de dados; caso contrário, retornam **false**. Por exemplo, chamar **hasNextInt()** só retorna **true** se o próximo item do fluxo for um inteiro na forma legível por humanos. Se o dado desejado estiver disponível, podemos lê-lo chamando um dos métodos **next** de **Scanner**, como **nextInt()** ou **nextDouble()**. Esses métodos convertem a forma dos dados legível por humanos em sua representação binária interna e retornam o resultado. Por exemplo, para ler um inteiro, chame **nextInt()**.

A sequência a seguir mostra como ler um inteiro a partir do teclado.

```
Scanner conin = new Scanner(System.in);
int i;

if (conin.hasNextInt()) i = conin.nextInt();
```

Usando esse código, se você inserir o número **123** no teclado, **i** conterá o valor 123.

Tecnicamente, você pode chamar um método **next** sem antes chamar um método **hasNext**. No entanto, pode não ser uma boa ideia. Se um método **next** não puder encontrar o tipo de dado que estiver procurando, lançará uma **InputMismatchException**. Logo, é melhor confirmar primeiro se o tipo de dado desejado está disponível chamando um método **hasNext** antes de chamar o método **next** correspondente.

EXERCÍCIOS

1. Por que Java define fluxos tanto de bytes quanto de caracteres?
2. Já que a entrada e a saída do console são baseadas em texto, por que Java ainda usa fluxos de bytes para esse fim?
3. Mostre como abrir um arquivo para a leitura de bytes.
4. Mostre como abrir um arquivo para a leitura de caracteres.
5. Mostre como abrir um arquivo para I/O de acesso aleatório.
6. Como podemos converter um string numérico como “123,23” em seu equivalente binário?
7. Escreva um programa que copie um arquivo de texto. No processo, faça-o converter todos os espaços em hífens. Use as classes de fluxos de bytes de arquivo. Use a abordagem tradicional para fechar um arquivo chamando **close()** explicitamente.
8. Reescreva o programa descrito no Exercício 7 para que use as classes de fluxos de caracteres. Dessa vez, use a instrução **try-with-resources** para fechar automaticamente o arquivo.
9. Que tipo de fluxo é **System.in**?
10. O que o método **read()** de **InputStream** retorna quando o fim do fluxo é alcançado?
11. Que tipo de fluxo é usado na leitura de dados binários?
12. **Reader** e **Writer** estão no topo das hierarquias de classes _____.
13. A instrução **try-with-resources** é usada para _____ _____.
14. Quando usamos o método tradicional de fechamento de arquivo, geralmente fechar um arquivo dentro de um bloco **finally** é uma boa abordagem. Verdadeiro ou falso?
15. Que classe dá acesso aos atributos de um arquivo?

16. Você pode usar a classe **File** para excluir um arquivo?
17. Crie um método **nameFromPath()** que use como parâmetro um string contendo o nome de caminho completo de um arquivo ou diretório. Faça-o retornar apenas o nome do arquivo ou diretório. Por exemplo, a chamada **nameFromPath("usr/etc/abc.txt")** retorna "abc.txt".
18. Reescreva o programa a seguir para que use **try-with-resources** para eliminar as chamadas a **close()**. Use apenas um bloco **try**:

```
import java.io.*;

class NoTryWithResources {
    public static void main(String[] args) {
        FileInputStream fin = null;
        FileOutputStream fout = null;

        // Primeiro verifica se os dois arquivos foram especificados.
        if(args.length != 2) {
            System.out.println("Usage: NoTryWithResources From To");
            return;
        }

        try {
            fin = new FileInputStream(args[0]);
        }
        catch (IOException exc) {
            System.out.println("IOException: program halted.");
        }
        try {
            fout = new FileOutputStream(args[1]);
        }
        catch (IOException exc) {
            System.out.println("IOException: program halted.");
        }
        try {
            if(fin != null && fout != null) {
                int c = fin.read();
                fout.write(c);
            }
        }
        catch (IOException exc) {
            System.out.println("IOException: program halted.");
        }
        finally {
            try {
                if(fin != null)
                    fin.close();
            }
            catch (IOException exc) {
```

```
        System.out.println("IOException: program halted.");
    }
}

try {
    if(fout != null)
        fout.close();
}
catch (IOException exc) {
    System.out.println("IOException: program halted.");
}
}

}
```

19. Crie um programa que use um nome de arquivo como argumento de linha de comando. Ele presume que o arquivo seja de dados binários. O programa verifica os 4 primeiros bytes para ver se contêm o inteiro -889275714 e exibe “yes”, “no” ou uma mensagem de erro se uma **IOException** tiver sido gerada. (Uma curiosidade: por que estamos procurando esse número específico? *Dica:* teste-o em vários arquivos **.class**.)
 20. Crie um segmento de código que exiba apenas o milésimo byte de um arquivo de dados binários chamado “datafile”. Use um **RandomAccessFile**.
 21. Crie um programa que use o nome de um arquivo de dados binários como argumento de linha de comando. Presuma que o arquivo contenha apenas inteiros. Usando um **RandomAccessFile**, classifique os dados do arquivo do menor para o maior. Use o algoritmo de classificação que quiser. Você não deve carregar os dados do arquivo em um array na memória para então classificá-lo; em vez disso, classifique os dados no próprio arquivo, usando a memória apenas para uma pequena quantidade fixa de variáveis. Exiba os inteiros do arquivo antes e depois da classificação para verificar se ela funcionou corretamente. *Dica:* a classe **RandomAccessFile** tem um método **length()** que retorna o número de bytes do arquivo como um valor **long**.
 22. Um formato de arquivo comum para o armazenamento de dados de uma tabela (como os obtidos em uma planilha) como texto é o CSV, ou “comma-separated values”. Todos os dados da tabela são armazenados no arquivo como linhas de texto e os dados de cada linha são separados uns dos outros por vírgulas. Crie um programa **CSVConverter** que use dois nomes de arquivo como argumentos de linha de comando. O primeiro arquivo é composto por dados binários (não é um arquivo de texto). A primeira linha desse arquivo contém dois inteiros e um caractere ‘\n’ de nova linha. O primeiro inteiro é o número de linhas de dados do arquivo e o segundo é o número de colunas de dados. O resto do arquivo contém linhas de inteiros, separadas por vírgulas, e terminando com caracteres ‘\n’ de nova linha. O programa **CSVConverter** deve extrair todos os dados do arquivo de dados e armazená-los no segundo arquivo como texto em formato **CSV**. Use um **DataInputStream** e um **FileWriter**.

Aqui está um programa que você pode usar para criar um arquivo de dados e usar como entrada de teste em seu programa **CSVConverter**. Se você executar este programa para criar um arquivo de dados e depois executar seu programa **CSVConverter** no arquivo, obterá um novo arquivo de texto com duas linhas, a primeira contendo “1,2,3” e a segunda contendo “4,5,6”.

```
import java.io.*;

public class DataFileCreator {
    public static void main(String[] args) {
        if(args.length != 1)
            System.out.println("Usage: DataFileCreator File");
        return;
    }

    try (DataOutputStream out =
        new DataOutputStream(new FileOutputStream(args[0]))) {
        out.writeInt(2); out.writeInt(3); out.writeChar('\n');
        out.writeInt( 1 ); out.writeChar(',');
        out.writeInt( 2 ); out.writeChar(',');
        out.writeInt( 3 ); out.writeChar('\n');
        out.writeInt( 4 ); out.writeChar(',');
        out.writeInt( 5 ); out.writeChar(',');
        out.writeInt( 6 ); out.writeChar('\n');
    } catch (IOException exc) {
        System.out.println("IOException: file creation cancelled.");
    }
}
```

23. Crie um programa que leia todos os dados de um arquivo de texto formatado com o CSV (consulte o exercício anterior para ver uma definição do formato CSV) e exiba a média de todos os números do arquivo. Por exemplo, se o arquivo tivesse os caracteres a seguir:

1,26,7
444,50,6

a saída relataria que a média dos valores do arquivo é 89. O nome do arquivo CSV deve ser fornecido como argumento de linha de comando. Você pode presumir que o arquivo terá pelo menos uma linha e uma coluna. Implemente seu programa

- A. usando um **BufferedReader** ou
 - B. usando um **Scanner**, como descrito na seção Pergunte ao especialista do fim do capítulo. *Nota: você terá que usar métodos da classe Scanner, não discutidos neste capítulo.*
24. Você pode criar um objeto **File** que não corresponda a um arquivo físico real? Por exemplo, suponhamos que não houvesse um arquivo com o nome “file.txt”

no diretório **/usr/bin**. Nesse caso, o código a seguir é válido? Se for, como saber se o arquivo realmente existe?

```
| File file = new File("/usr/bin/file.txt");
```

25. Crie uma classe chamada **FileUtilities** e adicione a ela os métodos **static** a seguir:

- Um método chamado **moveFile()** que use um objeto **File** e um diretório (um objeto **String**) como parâmetros e mova o arquivo para o diretório. Faça isso usando o método **renameTo()** da classe **File** que foi mencionado neste capítulo. O arquivo deve ser renomeado com o uso do mesmo nome de arquivo mas com um diretório diferente. O método **moveFile()** deve retornar **true** se for bem-sucedido; caso contrário, retornará **false**. Ele deve trabalhar com todos os arquivos e não apenas com arquivos de texto. O string do diretório deve terminar com um caractere de barra ('/').
- Um método chamado **moveFile()** que use dois **Strings** como parâmetros, o primeiro sendo o nome de um arquivo e o segundo o nome de um diretório. O método deve mover o arquivo para o diretório. Faça isso copiando o arquivo para o novo diretório e excluindo-o do diretório antigo. Ele retorna **true** se for bem-sucedido; caso contrário, retorna **false**. Deve trabalhar com todos os arquivos e não apenas com arquivos de texto. O string do diretório deve terminar com um caractere de barra ('/').
- Um método chamado **appendFile()** que use dois **Files** como parâmetros. Ele modifica o primeiro arquivo acrescentando o conteúdo do segundo. O método deve retornar **true** se for bem-sucedido; caso contrário, retornará **false**.

26. As classes encapsuladoras de tipos primitivos **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Boolean** e **Character** fornecem maneiras simples de conversão de seu valor binário em um string numérico equivalente e vice-versa. Especificamente, se **bObj** for um objeto **Byte**, então a instrução

```
| String bString = bObj.toString();
```

atribuirá a **bString** o string numérico equivalente ao valor binário de **bObj**. Por exemplo, se **bObj** tiver o valor 24, **bString** receberá o string “24”. Inversamente, dado um string **bString**, você pode convertê-lo em um valor binário equivalente com

```
| byte b = _____;
```

12

Programação com várias threads

PRINCIPAIS HABILIDADES E CONCEITOS

- Entender os fundamentos da criação de várias threads
- Conhecer a classe **Thread** e a interface **Runnable**
- Criar uma thread
- Criar várias threads
- Determinar quando uma thread termina
- Conhecer as prioridades de threads
- Entender a sincronização de threads
- Usar métodos sincronizados
- Usar blocos sincronizados
- Promover a comunicação entre threads
- Suspender, retomar e interromper threads

Um dos mais empolgantes recursos de Java é o suporte interno à *programação com várias threads**. Um programa com várias threads contém duas ou mais partes que podem ser executadas ao mesmo tempo. Cada parte de um programa assim se chama *thread* e cada thread define um caminho de execução separado. Logo, o uso de várias threads é um tipo de multitarefa especializada que pode ter um grande impacto sobre as características de tempo de execução de um programa.

FUNDAMENTOS DO USO DE VÁRIAS THREADS

Há dois tipos distintos de multitarefa: baseada em processos e baseada em threads. É importante entender a diferença entre os dois. Do modo como os termos são usados neste livro, um processo é, em essência, um programa que está sendo executado. Portanto, a multitarefa *baseada em processos* é o recurso que permite que o computador execute dois ou mais programas ao mesmo tempo. Por exemplo, é a multitarefa baseada em processos que nos permite executar o compilador Java ao mesmo tempo em que estamos usando um editor de texto ou navegando na Internet. Na multitarefa baseada em processos, um programa é a menor unidade de código que pode ser despachada pelo agendador.

* N. de R.T.: Em inglês, *multithreads*.

Em um ambiente multitarefa *baseado em threads*, a thread é a menor unidade de código que pode ser despachada. Ou seja, o mesmo programa pode executar duas ou mais tarefas ao mesmo tempo. Por exemplo, um editor de texto pode formatar texto ao mesmo tempo em que está imprimindo, contanto que essas duas ações estejam sendo executadas por duas threads separadas. Embora os programas Java façam uso de ambientes multitarefa baseados em processos, a multitarefa baseada em processos não é controlada por Java, mas a multitarefa com várias threads sim.

Uma vantagem importante do uso de várias threads é que ele permite a criação de programas muito eficientes, porque podemos utilizar o tempo ocioso que está presente em quase todos os programas. A maioria dos dispositivos de I/O, sejam portas de rede, unidades de disco ou o teclado, é muito mais lenta do que a CPU. Logo, com frequência o programa gasta grande parte de seu tempo de execução esperando receber ou enviar informações de ou para um dispositivo. Usando várias threads, o programa pode executar outra tarefa durante seu tempo ocioso. Por exemplo, enquanto uma parte do programa está enviando um arquivo pela Internet, outra parte pode estar lendo entradas no teclado, e ainda outra pode estar armazenando em buffer o próximo bloco de dados a ser enviado.

Sabemos que, nos últimos anos, os sistemas multiprocessadores e *multicore* se tornaram lugar comum, apesar de os sistemas com um único processador ainda serem muito usados. É importante entender que os recursos *multithread* de Java funcionam nos dois tipos de sistema. Em um sistema *single-core*, a execução concorrente de threads compartilha a CPU, com cada thread recebendo uma fração de tempo. Logo, em um sistema *single-core*, duas ou mais threads não são realmente executadas ao mesmo tempo: o tempo ocioso da CPU é que é utilizado. No entanto, em sistemas multiprocessadores/*multicore*, é possível duas ou mais threads serem executadas simultaneamente. Em muitos casos, isso pode melhorar ainda mais a eficiência do programa e aumentar a velocidade de certas operações.

Uma thread pode estar em um entre vários estados: pode estar *em execução*. Pode estar *pronta para execução* assim que conseguir tempo da CPU. Uma thread em execução pode estar *suspensa*, que é uma interrupção temporária em sua execução. Posteriormente, ela pode ser *retomada*. A thread também pode estar *bloqueada* quando espera um recurso, e pode ser *encerrada*, caso em que sua execução termina e não pode ser retomada.

Junto com a multitarefa baseada em threads vem a necessidade de um tipo especial de recurso chamado *sincronização*, que permite que a execução de threads seja coordenada de certas formas bem definidas. Java tem um subsistema completo dedicado à sincronização, e seus recursos-chave também serão descritos aqui.

Um último ponto: embora o uso de múltiplas threads adicione outra dimensão aos programas, o fato de Java gerenciar threads com elementos da linguagem torna a criação de várias threads conveniente e fácil de usar. Muitos dos detalhes são tratados automaticamente.

A CLASSE Thread E A INTERFACE Runnable

O sistema de várias threads de Java tem como base a classe **Thread** e a interface que a acompanha, **Runnable**. As duas estão empacotadas em **java.lang**. **Thread** encapsula uma thread de execução, além de fornecer métodos que são usados no gerenciamento da execução de threads. Vários serão examinados neste capítulo. Para criar uma nova thread, o programa deve estender **Thread** ou implementar a interface **Runnable**.

Todos os processos têm pelo menos uma thread de execução, que geralmente é chamada de *thread principal*, já que é ela que é executada quando o programa começa. Portanto, foi a thread principal que todos os exemplos de programa anteriores do livro usaram. A partir da thread principal, você pode criar outras threads.

Verificação do progresso

1. Qual é a diferença entre a multitarefa baseada em processos e a baseada em threads?
2. Em geral, que estados uma thread pode assumir?
3. Que classe encapsula uma thread?

CRIANDO UMA THREAD

Você pode criar uma thread instanciando um objeto de tipo **Thread**. A classe **Thread** encapsula um objeto que é executável. Como mencionado, Java define duas maneiras pelas quais você pode criar um objeto executável:

- Você pode implementar a interface **Runnable**.
- Você pode estender a classe **Thread**.

A maioria dos exemplos deste capítulo usará a abordagem que implementa **Runnable**. No entanto, a seção Tente isto 12-1 mostra como implementar uma thread estendendo **Thread**. Lembre-se: as duas abordagens usam a classe **Thread** para instanciar, acessar e controlar a thread. A única diferença é como uma classe para threads é criada.

A interface **Runnable** concebe uma unidade de código executável. Você pode construir uma thread em qualquer objeto que implementar a interface **Runnable**. **Runnable** só define um método, chamado **run()**, que é declarado assim:

```
public void run()
```

Dentro de **run()**, você definirá o código que constitui a nova thread. É importante saber que **run()** pode chamar outros métodos, usar outras classes e declarar variáveis da mesma forma que a thread principal. A única diferença é que **run()** estabelece o ponto de entrada de uma thread de execução concorrente dentro do programa. Essa thread terminará quando **run()** retornar.

Após ter criado uma classe que implementa **Runnable**, você poderá instanciar um objeto de tipo **Thread** em um objeto dessa classe. **Thread** define vários construtores. O que usaremos primeiro é mostrado aqui:

```
Thread(Runnable obThread)
```

Respostas:

1. A multitarefa baseada em processos é usada na execução de dois ou mais programas ao mesmo tempo. A multitarefa baseada em threads, chamada *multithreading*, é usada na execução de partes de um programa ao mesmo tempo.
2. Os estados das threads são em execução, pronta para ser executada, suspensa, bloqueada e encerrada. Quando uma thread suspensa é reiniciada, diz-se que ela foi retomada.
3. **Thread**

Nesse construtor, *obThread* é a instância de uma classe que implementa a interface **Runnable**. Isso define onde a execução da thread começará.

Uma vez criada, a nova thread só começará a ser executada quando você chamar seu método **start()**, que é declarado por **Thread**. O método **start()** faz a JVM chamar **run()**. Ele é mostrado abaixo:

```
void start()
```

Veja um exemplo que cria uma nova thread e começa a executá-la:

```
// Cria uma thread implementando Runnable.

class MyThread implements Runnable { ← Objetos de MyThread
    String thrdName; ← podem ser executados em
                        suas próprias threads, porque
MyThread implementa
Runnable.

    MyThread(String name) {
        thrdName = name;
    }

    // Ponto de entrada da thread.
    public void run() { ← Threads começam a ser
                        executadas aqui.
        System.out.println(thrdName + " starting.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println("In " + thrdName +
                                   ", count is " + count);
            }
        } catch(InterruptedException exc) {
            System.out.println(thrdName + " interrupted.");
        }
        System.out.println(thrdName + " terminating.");
    }
}

class UseThreads {
    public static void main(String[] args) {
        System.out.println("Main thread starting.");

        // Primeiro, constrói um objeto MyThread.
        MyThread mt = new MyThread("Child #1"); ← Cria um objeto executável.

        // Em seguida, constrói uma thread a partir desse objeto.
        Thread newThrd = new Thread(mt); ← Constrói uma thread neste objeto.

        // Para concluir, começa a execução da thread.
        newThrd.start(); ← Começa a executar a thread.

        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {

```

```
        Thread.sleep(100);
    }
    catch(InterruptedException exc) {
        System.out.println("Main thread interrupted.");
    }
}

System.out.println("Main thread ending.");
}
```

Examinemos esse programa com detalhes. Primeiro, **MyThread** implementa **Runnable**, ou seja, um objeto de tipo **MyThread** fica disponível para ser usado como uma thread e pode ser passado para o construtor de **Thread**.

Dentro de **run()**, é estabelecido um laço que conta de 0 a 9. Observe a chamada a **sleep()**. O método **sleep()** faz a thread em que é chamado suspender a execução pelo período de milissegundos especificado. Sua forma geral é mostrada aqui:

```
static void sleep(long milissegundos) throws InterruptedException
```

O período em milissegundos da suspensão é especificado em *milissegundos*. Esse método pode lançar uma **InterruptedException**. Logo, as chamadas feitas a ele devem ser encapsuladas em um bloco **try**. O método **sleep()** também tem uma segunda forma que permite a especificação do período em milissegundos e nanossegundos, se for preciso esse nível de precisão. Em **run()**, **sleep()** pausa a thread por 400 milissegundos a cada passagem pelo laço. Isso permite que a thread seja executada com lentidão suficiente para observarmos sua execução.

Dentro de **main()**, um novo objeto **Thread** é criado pela sequência de instruções a seguir:

```
// Primeiro constrói um objeto MyThread.
MyThread mt = new MyThread("Child #1");

// A seguir, constrói uma thread a partir desse objeto.
Thread newThrd = new Thread(mt);

// Finalmente, inicia a execução da thread.
newThrd.start();
```

Como os comentários sugerem, primeiro um objeto de **MyThread** é criado. Esse objeto é então usado para construir um objeto **Thread**. Isso é possível porque **MyThread** implementa **Runnable**. Para concluir, a execução da nova thread é iniciada com uma chamada a **start()**, o que faz o método **run()** da thread filha ser executado. Após chamar **start()**, a execução retorna para **main()** e entra no laço **for**. Observe que esse laço itera 50 vezes, pausando 100 milissegundos sempre que é percorrido. As duas threads continuam sendo executadas, compartilhando a CPU em sistemas de CPU única, até seus laços terminarem. A saída produzida por esse programa é mostrada abaixo. Devido a diferenças entre os ambientes de computação, a saída que você verá pode diferir um pouco da mostrada aqui:

```
Main thread starting.
.Child #1 starting.
...In Child #1, count is 0
```

```

....In Child #1, count is 1
....In Child #1, count is 2
....In Child #1, count is 3
....In Child #1, count is 4
....In Child #1, count is 5
....In Child #1, count is 6
....In Child #1, count is 7
....In Child #1, count is 8
....In Child #1, count is 9
Child #1 terminating.
.....Main thread ending.

```

Há outro ponto interessante a ser observado nesse primeiro exemplo das threads. Para ilustrar o fato de que a thread principal e a thread **mt** estão sendo executadas ao mesmo tempo, não podemos deixar que **main()** termine antes de **mt** ter terminado. Aqui, isso é feito por intermédio das diferenças de ritmo entre as duas threads. Já que as chamadas a **sleep()** dentro do laço **for** de **main()** causam um retardo total de 5 segundos (50 iterações vezes 100 milissegundos), mas o retardo total dentro do laço de **run()** é de apenas 4 segundos (10 iterações vezes 400 milissegundos), **run()** terminará cerca de 1 segundo antes de **main()**. Como resultado, tanto a thread principal quanto a thread **mt** serão executadas ao mesmo tempo até **mt** terminar. Cerca de 1 segundo depois, **main()** terminará.

Embora esse uso das diferenças de ritmo para assegurar que **main()** termine por último seja suficiente nesse e nos próximos exemplos, não é algo muito usado na prática. Java fornece maneiras muito melhores de esperar uma thread terminar.

É importante ressaltar que fazer a thread principal terminar por último não é necessariamente um requisito do uso de várias threads. Para o tipo de threads criadas neste capítulo, um programa continuará a ser executado até todas as suas threads filhas terem terminado. No entanto, fazer a thread principal terminar por último costuma ser útil quando é a primeira vez que ouvimos falar das threads.

Algumas melhorias simples

Embora o programa anterior seja perfeitamente válido, algumas melhorias simples o tornarão mais eficiente e fácil de usar. Em primeiro lugar, é possível fazer a thread começar a ser executada assim que for criada. No caso de **MyThread**, isso é feito pela instânciação de um objeto **Thread** dentro do construtor de **MyThread**. Em segundo lugar, não há necessidade de **MyThread** armazenar o nome da thread, já que é possível dar um nome a uma thread quando ela é criada. Para fazê-lo, use esta versão do construtor de **Thread**:

```
Thread(Runnable obThread, String nome)
```

Aqui, *nome* passa a ser o nome da thread.

Você pode obter o nome de uma thread chamando o método **getName()** definido por **Thread**. Sua forma geral é mostrada aqui:

```
final String getName()
```

Embora não seja necessário no programa a seguir, você pode definir o nome de uma thread após ela ser criada usando **setName()**, que é mostrado abaixo:

```
final void setName(String nomeThread)
```

Aqui, *nomeThread* especifica o nome da thread.

Esta é a versão melhorada do programa anterior:

```
// MyThread melhorada.

class MyThread implements Runnable {
    Thread thrd; ← Uma referência à thread é armazenada em thrd.

    // Constrói uma nova thread.
    MyThread(String name) {
        thrd = new Thread(this, name); ← A thread é nomeada quando é criada.
        thrd.start(); // inicia a thread ← Começa a executar a thread.
    }

    // Começa a execução da nova thread.
    public void run() {
        System.out.println(thrd.getName() + " starting.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println("In " + thrd.getName() +
                    ", count is " + count);
            }
        }
        catch(InterruptedException exc) {
            System.out.println(thrd.getName() + " interrupted.");
        }
        System.out.println(thrd.getName() + " terminating.");
    }
}

class UseThreadsImproved {
    public static void main(String[] args) {
        System.out.println("Main thread starting.");

        MyThread mt = new MyThread("Child #1"); ← Agora a thread começa
                                                    quando é criada.

        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Main thread interrupted.");
            }
        }

        System.out.println("Main thread ending.");
    }
}
```

Essa versão produz a mesma saída de antes. Observe que a instância da thread é armazenada em **thrd** dentro de **MyThread**.

Verificação do progresso

1. Quais são as duas maneiras pelas quais podemos criar uma classe que seja uma thread?
2. Qual é a finalidade do método **run()** definido por **Runnable**?
3. O que faz o método **start()** definido por **Thread**?

Pergunte ao especialista

P Há diferentes tipos de threads?

R Java define dois tipos básicos de threads: de usuário e de daemon. Os tipos de threads criados pelos programas deste capítulo são os threads de usuário. Uma thread de usuário continua a ser executada até terminar. Uma thread de daemon é encerrada automaticamente quando todas as threads de usuário terminam. Por padrão, uma nova thread é do mesmo tipo da thread que a criou. Já que a thread principal é uma thread de usuário, as threads criadas neste capítulo são threads de usuário. Você pode mudar uma thread para uma thread de daemon chamando o método **setDaemon()**, mostrado aqui:

```
final void setDaemon(boolean éDaemon)
```

Se *éDaemon* for **true**, a thread será uma thread de daemon. Caso contrário, será uma thread de usuário. Esse método deve ser chamado antes do método **start()** do thread.

TENTE ISTO 12-1 Estendendo Thread

`ExtendThread.java`

A implementação de **Runnable** é uma maneira de criar uma classe que possa instanciar objetos de thread. A extensão de **Thread** é outra. Neste projeto, você verá como estender **Thread** criando um programa funcionalmente idêntico ao programa **UseThreadsImproved**.

Se uma classe estender **Thread**, ela deve sobrepor o método **run()**, que é o ponto de entrada da nova thread. Também deve chamar **start()** para começar a execução da nova thread. Podemos sobrepor outros métodos de **Thread**, mas não é obrigatório.

Respostas:

1. Para criar uma thread, implemente **Runnable** ou estenda **Thread**.
2. O método **run()** é o ponto de entrada de uma thread.
3. O método **start()** inicia a execução de uma thread.

PASSO A PASSO

- Crie um arquivo chamado **ExtendThread.java**. Copie nesse arquivo o código do segundo exemplo das threads (**UseThreadsImproved.java**).
- Altere a declaração de **MyThread** para que estenda **Thread** em vez de implementar **Runnable**, como mostrado aqui:

```
| class MyThread extends Thread {
```

- Remova esta linha:

```
| Thread thrd;
```

A variável **thrd** não é mais necessária, já que **MyThread** inclui uma instância de **Thread** e pode referenciar a si mesma.

- Altere o construtor de **MyThread** para que fique com a seguinte aparência:

```
// Constrói uma nova thread.
MyThread(String name) {
    super(name); // nomeia a thread
    start(); // inicia a thread
}
```

Como você pode ver, primeiro **super** é usada para chamar esta versão do construtor de **Thread**:

```
Thread(String nome);
```

Aqui, *nome* é o nome da thread. Portanto, esse construtor define o nome da thread.

- Altere **run()** para que chame **getName()** diretamente, sem qualificá-lo com a variável **thrd**. Sua aparência deve ser a seguinte:

```
// Começa a execução da nova thread.
public void run() {
    System.out.println(getName() + " starting.");
    try {
        for(int count=0; count < 10; count++) {
            Thread.sleep(400);
            System.out.println("In " + getName() +
                               ", count is " + count);
        }
    } catch(InterruptedException exc) {
        System.out.println(getName() + " interrupted.");
    }
    System.out.println(getName() + " terminating.");
}
```

- A seguir, temos o programa completo que agora estende **Thread** em vez de implementar **Runnable**. A saída é a mesma de antes.

```
/*
 Tente isto 12-1

 Estende Thread.
*/
class MyThread extends Thread {

    // Constrói uma nova thread.
    MyThread(String name) {
        super(name); // nomeia a thread
        start(); // inicia a thread
    }

    // Começa a execução da nova thread.
    public void run() {
        System.out.println(getName() + " starting.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println("In " + getName() +
                    ", count is " + count);
            }
        }
        catch(InterruptedException exc) {
            System.out.println(getName() + " interrupted.");
        }

        System.out.println(getName() + " terminating.");
    }
}

class ExtendThread {
    public static void main(String[] args) {
        System.out.println("Main thread starting.");

        MyThread mt = new MyThread("Child #1");

        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Main thread interrupted.");
            }
        }

        System.out.println("Main thread ending.");
    }
}
```

CRIANDO VÁRIAS THREADS

Os exemplos anteriores criaram apenas uma thread filha. No entanto, seu programa pode gerar quantas threads precisar. Por exemplo, o programa a seguir cria três threads filhas:

```
// Cria várias threads.

class MyThread implements Runnable {
    Thread thrd;

    // Constrói uma nova thread.
    MyThread(String name) {
        thrd = new Thread(this, name);

        thrd.start(); // inicia a thread
    }

    // Começa a execução da nova thread.
    public void run() {
        System.out.println(thrd.getName() + " starting.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println("In " + thrd.getName() +
                    ", count is " + count);
            }
        }
        catch(InterruptedException exc) {
            System.out.println(thrd.getName() + " interrupted.");
        }
        System.out.println(thrd.getName() + " terminating.");
    }
}

class MoreThreads {
    public static void main(String[] args) {
        System.out.println("Main thread starting.");

        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3"); Cria e começa a executar  
três threads.

        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Main thread interrupted.");
            }
        }
    }
}
```

```

    }

    System.out.println("Main thread ending.");
}
}

```

Um exemplo da saída desse programa é mostrado abaixo:

```

Main thread starting.
Child #1 starting.
.Child #2 starting.
Child #3 starting.
...In Child #3, count is 0
In Child #2, count is 0
In Child #1, count is 0
....In Child #1, count is 1
In Child #2, count is 1
In Child #3, count is 1
....In Child #2, count is 2
In Child #3, count is 2
In Child #1, count is 2
...In Child #1, count is 3
In Child #2, count is 3
In Child #3, count is 3
....In Child #3, count is 4
In Child #1, count is 4
In Child #2, count is 4
....In Child #1, count is 5
In Child #3, count is 5
In Child #2, count is 5
....In Child #3, count is 6
.In Child #2, count is 6
In Child #1, count is 6
...In Child #3, count is 7
In Child #1, count is 7
In Child #2, count is 7
....In Child #2, count is 8
In Child #1, count is 8
In Child #3, count is 8
....In Child #1, count is 9
Child #1 terminating.
In Child #2, count is 9
Child #2 terminating.
In Child #3, count is 9
Child #3 terminating.
.....Main thread ending.

```

Como você pode ver, uma vez iniciadas, todas as três threads filhas são executadas ao mesmo tempo. Observe que as threads são iniciadas na ordem em que são criadas. No entanto, nem sempre isso ocorre. Java pode agendar a execução de threads como quiser. É claro que, devido a diferenças no ‘timing’ ou no ambiente, a saída exata exibida

pelo programa pode diferir, por isso, não se surpreenda ao ver resultados diferentes quando testar o programa.

Pergunte ao especialista

P Por que Java tem duas maneiras de criar threads filhas (estendendo **Thread** ou implementando **Runnable**)? Qual abordagem é melhor?

R A classe **Thread** define vários métodos que podem ser sobrepostos por uma classe derivada. Entre eles, o único que *deve* ser sobreposto é **run()**. É claro que esse é o mesmo método requerido quando implementamos **Runnable**. Alguns programadores de Java acham que as classes só devem ser estendidas quando estão sendo melhoradas ou modificadas de alguma forma. Logo, se não sobrepujermos outros métodos de **Thread**, pode ser melhor simplesmente implementar **Runnable**. Além disso, ao implementar **Runnable**, estamos permitindo que a thread herde uma classe que não seja **Thread**.

DETERMINANDO QUANDO UMA THREAD TERMINA

Costuma ser útil saber quando uma thread terminou. Por exemplo, nos exemplos anteriores, a título de ilustração foi útil manter a thread principal ativa até as outras threads terminarem. Nesses exemplos, conseguimos isso fazendo a thread principal entrar em suspensão por mais tempo do que as threads filhas que ela gerou. É claro que dificilmente essa seria uma solução satisfatória ou que pudesse ser generalizada!

Felizmente, **Thread** fornece dois meios pelos quais você pode determinar se uma thread terminou. O primeiro é chamar o método **isAlive()** na thread. Sua forma geral é mostrada aqui:

```
final boolean isAlive()
```

O método **isAlive()** retorna **true** se a thread em que foi chamado ainda estiver sendo executada. Caso contrário, retorna **false**. Para testar **isAlive()**, substitua a versão de **MoreThreads** mostrada no programa anterior por esta:

```
// Usa isAlive().
class MoreThreads {
    public static void main(String[] args) {
        System.out.println("Main thread starting.");

        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");

        do {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Main thread interrupted.");
            }
        }
    }
}
```

```

    } while (mt1.thrd.isAlive() || _____)
            mt2.thrd.isAlive() || _____)
            mt3.thrd.isAlive()); _____} Espera até todas as
                                         threads terminarem.

    System.out.println("Main thread ending.");
}
}

```

Essa versão usa **isAlive()** para esperar as threads filhas terminarem. Ela produz uma saída semelhante à do exemplo anterior, porém **main()** termina assim que as threads filhas terminam.

Outra maneira de esperar uma thread terminar é chamar o método **join()**, mostrado aqui:

```
final void join() throws InterruptedException
```

Esse método espera até que a thread em que foi chamado termine. Seu nome vem do fato de a thread que fez a chamada ter de esperar até a thread especificada se *juntar* a ela. Formas adicionais de **join()** nos permitem indicar o período de tempo máximo que queremos esperar que a thread especificada termine.

Veja um programa que usa **join()** para assegurar que a thread principal seja a última a terminar:

```

// Usa join().

class MyThread implements Runnable {
    Thread thrd;

    // Constrói uma nova thread.
    MyThread(String name) {
        thrd = new Thread(this, name);
        thrd.start(); // inicia a thread
    }

    // Começa a execução da nova thread.
    public void run() {
        System.out.println(thrd.getName() + " starting.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println("In " + thrd.getName() +
                                   ", count is " + count);
            }
        }
        catch(InterruptedException exc) {
            System.out.println(thrd.getName() + " interrupted.");
        }
        System.out.println(thrd.getName() + " terminating.");
    }
}

```

```
class JoinThreads {  
    public static void main(String[] args) {  
        System.out.println("Main thread starting.");  
  
        MyThread mt1 = new MyThread("Child #1");  
        MyThread mt2 = new MyThread("Child #2");  
        MyThread mt3 = new MyThread("Child #3");  
  
        try {  
            mt1.thrd.join(); ← Espera até a thread  
            System.out.println("Child #1 joined."); especificada terminar.  
            mt2.thrd.join(); ←  
            System.out.println("Child #2 joined.");  
            mt3.thrd.join(); ←  
            System.out.println("Child #3 joined.");  
        }  
        catch(InterruptedException exc) {  
            System.out.println("Main thread interrupted. ");  
        }  
        System.out.println("Main thread ending.");  
    }  
}
```

Um exemplo da saída desse programa é mostrado abaixo. Lembre-se de que, quando você testar o programa, sua saída exata pode variar um pouco.

```
Main thread starting.  
Child #1 starting.  
Child #2 starting.  
Child #3 starting.  
In Child #2, count is 0  
In Child #3, count is 0  
In Child #1, count is 0  
In Child #1, count is 1  
In Child #2, count is 1  
In Child #3, count is 1  
In Child #1, count is 2  
In Child #2, count is 2  
In Child #3, count is 2  
In Child #1, count is 3  
In Child #2, count is 3  
In Child #3, count is 3  
In Child #1, count is 4  
In Child #2, count is 4  
In Child #3, count is 4  
In Child #1, count is 5  
In Child #2, count is 5  
In Child #3, count is 5  
In Child #1, count is 5  
In Child #3, count is 6  
In Child #2, count is 6  
In Child #1, count is 6
```

```
In Child #1, count is 7
In Child #3, count is 7
In Child #2, count is 7
In Child #1, count is 8
In Child #3, count is 8
In Child #2, count is 8
In Child #2, count is 9
In Child #3, count is 9
In Child #3, count is 9
Child #3 terminating.
In Child #1, count is 9
Child #1 terminating.
Child #2 terminating.
Child #1 joined.
Child #2 joined.
Child #3 joined.
Main thread ending.
```

Como você pode ver, quando as chamadas a **join()** retornam, as threads não estão mais sendo executadas. Então, a thread principal termina.

Verificação do progresso

1. Quais são as duas maneiras pelas quais podemos determinar se uma thread terminou?
2. Explique **join()**.

PRIORIDADES DAS THREADS

Cada thread tem associada a ela uma configuração de prioridade. A prioridade de uma thread determina, em parte, quanto tempo de CPU ela receberá em relação a outras threads ativas. Em geral, threads de baixa prioridade recebem pouco tempo. Threads de alta prioridade recebem muito tempo. Como era de se esperar, o tempo de CPU que uma thread recebe tem impacto profundo sobre suas características de execução e sua interação com outras threads sendo executadas atualmente no sistema.

É importante entender que outros fatores além da prioridade afetam quanto tempo de CPU a thread receberá. Por exemplo, se uma thread de alta prioridade estiver esperando algum recurso, talvez uma entrada do teclado, ela será bloqueada, e uma thread de prioridade mais baixa será executada. No entanto, quando a thread de alta prioridade ganhar acesso ao recurso, poderá interceptar a thread de baixa prioridade e retomar a execução. Outro fator que afeta o agendamento de threads é a maneira como o sistema operacional implementa a multitarefa. (Consulte a seção “Pergunte ao Especialista” no fim desta seção.) Logo, não é porque você deu prioridade alta a uma thread e prioridade baixa a outra que uma thread será necessariamente executada

Respostas:

1. Para determinar se uma thread terminou, você pode chamar **isAlive()** ou usar **join()** para esperar que ela se junte à thread chamadora.
2. O método **join()** suspende a execução da thread chamadora até a thread em que foi chamado terminar.

com mais rapidez ou frequência do que a outra. Simplesmente, a thread de alta prioridade tem mais chances de acessar a CPU.

Quando uma thread filha é iniciada, sua configuração de prioridade é igual à da thread mãe. Você pode alterar a prioridade de uma thread chamando o método `setPriority()`, que é membro de **Thread**. Esta é sua forma geral:

```
final void setPriority(int nível)
```

Aqui, *nível* especifica a nova configuração de prioridade da thread que fez a chamada. O valor de *nível* deve estar dentro do intervalo **MIN_PRIORITY** e **MAX_PRIORITY**. Atualmente, esses valores são 1 e 10, respectivamente. Para retornar uma thread para a prioridade padrão, especifique **NORM_PRIORITY**, que atualmente é 5. Essas prioridades estão definidas como variáveis estáticas e finais dentro de **Thread**.

Você pode obter a configuração de prioridade atual chamando o método `getPriority()` de **Thread**, mostrado abaixo:

```
final int getPriority()
```

Embora possa haver situações em que é adequado configurar a prioridade de uma thread, o uso da configuração padrão costuma ser uma opção melhor – principalmente quando estamos apenas começando a usar o multithreading. Uma razão é que nos ambientes *multicore* atuais, aumentar ou diminuir a prioridade de uma thread pode ter pouco impacto sobre suas características de tempo de execução. Além disso, nunca devemos usar a configuração de prioridade de uma thread como meio de gerenciar a interação entre threads. Para tratar a interação das threads, devemos usar um dos recursos Java de sincronização, que serão descritos na próxima seção.

Pergunte ao especialista

P A implementação da multitarefa por parte do sistema operacional afeta o tempo de CPU que uma thread vai receber?

R Um dos fatores mais importantes que afetam a execução de threads é a maneira como o sistema operacional implementa a multitarefa e o agendamento. Por exemplo, é comum um sistema operacional usar a multitarefa com preempção em que cada thread recebe uma fração de tempo, pelo menos ocasionalmente. No entanto, também é possível um sistema operacional usar o agendamento sem preempção em que uma thread deve abandonar a execução para outra ser executada. Quando um programa *multithread* está sendo executado em um ambiente sem preempção, é fácil uma thread assumir o controle, impedindo que outras sejam executadas.

SÍNCRONIZAÇÃO

Quando várias threads são usadas, às vezes é necessário coordenar as atividades de duas ou mais. O processo que faz isso se chama *sincronização*. A razão mais comum para o uso da sincronização é quando duas ou mais threads precisam de acesso a um recurso compartilhado que só pode ser usado por uma thread de cada vez. Por exemplo, quando uma thread está gravando em um arquivo, uma segunda thread deve ser

impedida de gravar ao mesmo tempo. Outra razão para usarmos a sincronização é quando uma thread está esperando um evento causado por outra thread. Nesse caso, é preciso que haja um meio de a primeira thread ser mantida em estado suspenso até o evento ocorrer. Então, a thread em espera deve retomar a execução.

Essencial para a sincronização em Java é o conceito de *monitor*, que controla o acesso a um objeto. Um monitor funciona implementando o conceito de *bloqueio*. Quando um objeto é bloqueado por uma thread, nenhuma outra thread pode acessá-lo. Quando a thread termina, o objeto é desbloqueado e fica disponível para ser usado por outra thread.

Todos os objetos em Java têm um monitor. Esse recurso existe dentro da própria linguagem Java. Logo, todos os objetos podem ser sincronizados. A sincronização é suportada pela palavra-chave **synchronized** e alguns métodos bem definidos que todos os objetos têm. Como a sincronização foi projetada em Java desde o início, é muito mais fácil de usar do que parece. Na verdade, para muitos programas, a sincronização é quase transparente.

Você pode sincronizar seu código de duas maneiras. Ambas envolvem o uso da palavra-chave **synchronized** e serão examinadas aqui.

USANDO MÉTODOS SINCRONIZADOS

Você pode sincronizar o acesso a um método modificando-o com a palavra-chave **synchronized**. Quando esse método for chamado, a thread chamadora entrará no monitor do objeto, que então será bloqueado. Enquanto ele estiver bloqueado, nenhuma outra thread poderá entrar no método (ou em qualquer outro método sincronizado definido pela classe do objeto). Quando a thread retornar do método, o monitor desbloqueará o objeto, permitindo que ele seja usado pela próxima thread. Logo, a sincronização é obtida sem que você faça praticamente nenhum esforço de programação.

O programa a seguir demonstra a sincronização controlando o acesso a um método chamado **sumArray()**, que soma os elementos de um array de inteiros.

```
// Usa a sincronização para controlar o acesso.

class SumArray {
    private int sum;

    synchronized int sumArray(int[] nums) { ←———— sumArray() é sincronizado.
        sum = 0; // zera sum

        for(int i=0; i<nums.length; i++) {
            sum += nums[i];
            System.out.println("Running total for " +
                Thread.currentThread().getName() +
                " is " + sum);
        }
        try {
            Thread.sleep(10); // permite a alternância de tarefas
        }
        catch(InterruptedException exc) {
            System.out.println("Thread interrupted.");
        }
    }
}
```

```
        }
    }
    return sum;
}
}

class MyThread implements Runnable {
    Thread thrd;
    static SumArray sa = new SumArray();
    int[] a;
    int answer;

    // Constrói uma nova thread.
    MyThread(String name, int[] nums) {
        thrd = new Thread(this, name);
        a = nums;
        thrd.start(); // inicia a thread
    }

    // Começa a execução da nova thread.
    public void run() {
        int sum;

        System.out.println(thrd.getName() + " starting.");

        answer = sa.sumArray(a);
        System.out.println("Sum for " + thrd.getName() +
                           " is " + answer);

        System.out.println(thrd.getName() + " terminating.");
    }
}

class Sync {
    public static void main(String[] args) {
        int[] a = {1, 2, 3, 4, 5};

        MyThread mt1 = new MyThread("Child #1", a);
        MyThread mt2 = new MyThread("Child #2", a);

        try {
            mt1.thrd.join();
            mt2.thrd.join();
        }
        catch(InterruptedException exc) {
            System.out.println("Main thread interrupted.");
        }
    }
}
```

A saída do programa é mostrada aqui. (A saída exata pode ser diferente em seu computador.)

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 1
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Sum for Child #2 is 15
Child #2 terminating.
```

Examinemos esse programa em detalhes. Ele cria três classes: a primeira é **SumArray**, a qual contém o método **sumArray()**, que soma um array de inteiros. A segunda classe é **MyThread**, que usa um objeto **static** de tipo **SumArray** para obter a soma de um array de inteiros. Esse objeto se chama **sa** e, por ser **static**, há apenas uma cópia dele compartilhada por todas as instâncias de **MyThread**. Para concluir, a classe **Sync** cria duas threads e as faz calcular a soma de um array de inteiros.

Dentro de **sumArray()**, **sleep()** é chamado para permitir que ocorra uma alternância intencional de tarefas, se puder ocorrer uma – mas não pode. Como **sumArray()** é sincronizado, só pode ser usado por uma thread de cada vez, seja qual for o objeto. Logo, quando a segunda thread filha começa a ser executada, ela não entra em **sumArray()** até que a primeira thread filha tenha acabado de usá-lo. Isso assegura que o resultado correto seja produzido.

Para entender melhor os efeitos de **synchronized**, tente removê-la da declaração de **sumArray()**. Após fazê-lo, **sumArray** não será mais sincronizado e um número ilimitado de threads poderá executá-lo ao mesmo tempo. O problema é que o total atual é armazenado em **sum**, que será alterada por cada thread que chamar **sumArray()** por intermédio do objeto estático **sa**. Logo, quando duas threads chamam **sa.sumArray()** ao mesmo tempo, resultados incorretos são produzidos porque **sum** reflete a soma feita pelas duas threads juntas. Por exemplo, aqui está uma amostra da saída do programa após **synchronized** ser removida da declaração de **sumArray()**. (A saída exata pode ser diferente em seu computador.)

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #2 is 1
Running total for Child #1 is 3
Running total for Child #2 is 5
Running total for Child #2 is 8
Running total for Child #1 is 11
```

```
Running total for Child #2 is 15
Running total for Child #1 is 19
Running total for Child #2 is 24
Sum for Child #2 is 24
Child #2 terminating.
Running total for Child #1 is 29
Sum for Child #1 is 29
Child #1 terminating.
```

Como a saída mostra, as duas threads filhas estão chamando `sa.sumArray()` ao mesmo tempo e o valor de `sum` foi corrompido. Antes de prosseguir, examinemos os pontos-chave de um método sincronizado:

- Um método sincronizado é criado quando precedemos sua declaração com **synchronized**.
- Para qualquer objeto dado, uma vez que um método sincronizado tiver sido chamado, o objeto será bloqueado e nenhum método sincronizado no mesmo objeto poderá ser usado por outra thread de execução.
- Outras threads que tentarem chamar um método sincronizado em um objeto bloqueado entrarão em estado de espera até o objeto ser desbloqueado.
- Quando uma thread deixa o método sincronizado, o objeto é desbloqueado.

A INSTRUÇÃO synchronized

Embora a criação de métodos **synchronized** dentro das classes que criamos seja um meio fácil e eficaz de obter sincronização, ele não funciona em todos os casos. Por exemplo, podemos querer sincronizar o acesso a algum método que não seja modificado por **synchronized**. Isso pode ocorrer por querermos usar uma classe que não foi criada por nós, e sim por terceiros, e não termos acesso ao código-fonte. Logo, não é possível adicionar **synchronized** aos métodos apropriados dentro da classe. Como o acesso a um objeto dessa classe pode ser sincronizado? Felizmente, é muito fácil resolver esse problema: só temos de inserir as chamadas aos métodos definidos por essa classe dentro de um bloco **synchronized**.

Esta é a forma geral de um bloco **synchronized**:

```
synchronized(refobj){
    // instruções a serem sincronizadas
}
```

Aqui, *refobj* é uma referência ao objeto para o qual a sincronização é necessária. Uma vez que entrarmos em um bloco sincronizado, nenhuma outra thread poderá chamar um método sincronizado ou entrar em um bloco sincronizado no objeto referenciado por *refobj* até sairmos do bloco.

Por exemplo, outra maneira de sincronizar as chamadas a `sumArray()` é chamá-lo de dentro de um bloco sincronizado, como mostrado nesta versão do programa:

```
// Usa um bloco sincronizado para controlar o acesso a sumArray.
class SumArray {
    private int sum;
```

```

int sumArray(int[] nums) { ← Aqui, sumArray() não é sincronizado.
    sum = 0; // zera sum

    for(int i=0; i<nums.length; i++) {
        sum += nums[i];
        System.out.println("Running total for " +
            Thread.currentThread().getName() +
            " is " + sum);
        try {
            Thread.sleep(10); // permite a alternância de tarefas
        }
        catch(InterruptedException exc) {
            System.out.println("Thread interrupted.");
        }
    }
    return sum;
}

class MyThread implements Runnable {
    Thread thrd;
    static SumArray sa = new SumArray();
    int[] a;
    int answer;

    // Constrói uma nova thread.
    MyThread(String name, int[] nums) {
        thrd = new Thread(this, name);
        a = nums;
        thrd.start(); // inicia a thread
    }

    // Começa a execução da nova thread.
    public void run() {
        System.out.println(thrd.getName() + " starting.");

        // sincroniza as chamadas a sumArray()
        synchronized(sa) { ← Aqui, as chamadas a sumArray()
            answer = sa.sumArray(a);
            em sa são sincronizadas.
        }
        System.out.println("Sum for " + thrd.getName() +
            " is " + answer);

        System.out.println(thrd.getName() + " terminating.");
    }
}

class Sync {
    public static void main(String[] args) {
        int[] a = {1, 2, 3, 4, 5};

```

```

MyThread mt1 = new MyThread("Child #1", a);
MyThread mt2 = new MyThread("Child #2", a);

try {
    mt1.thrd.join();
    mt2.thrd.join();
} catch(InterruptedException exc) {
    System.out.println("Main Thread interrupted.");
}
}
}
}

```

Essa versão produz uma saída correta e igual à mostrada anteriormente que usa um método sincronizado.

Verificação do progresso

1. Como podemos configurar a prioridade de uma thread?
2. Como podemos restringir o acesso a um objeto a uma thread de cada vez?
3. A palavra-chave **synchronized** pode ser usada para modificar um método ou criar um bloco _____.

Pergunte ao especialista

P Um amigo me falou sobre algo chamado “utilitário de concorrência”. O que é? Além disso, o que é o Framework Fork/Join?

R Os utilitários de concorrência, que estão empacotados em **java.util.concurrent** (e seus subpacotes), dão suporte à programação concorrente. Entre vários outros itens, eles oferecem sincronizadores, pools de threads, gerenciadores de execução e bloqueios que expandem o controle sobre a execução de threads. Um dos recursos mais interessantes da API de concorrência é o Framework Fork/Join, que foi adicionado pelo JDK 7.

O Framework Fork/Join dá suporte ao que costuma ser chamado de *programação paralela*. Esse é o nome normalmente dado às técnicas que se beneficiam de computadores que contêm dois ou mais processadores (inclusive sistemas *multicore*) e subdividem uma tarefa em subtarefas, com cada subtarefa sendo executada em seu próprio processador. Como era de se esperar, essa abordagem pode levar a uma taxa de transferência e a um desempenho significativamente melhores. A principal vantagem do Framework Fork/Join é ser fácil de usar; ele otimiza o desenvolvimento de códigos com várias threads que se adaptam automaticamente para utilizar os vários processadores de um sistema. Após você aprender os fundamentos do multithreading, estará pronto para passar para os utilitários de concorrência. Eles são descritos com detalhes no Capítulo 27.

Respostas:

1. Para configurar a prioridade de uma thread, chame **setPriority()**.
2. Para restringir o acesso a um objeto a uma thread de cada vez, use a palavra-chave **synchronized**.
3. sincronizado

COMUNICAÇÃO ENTRE THREADS COM O USO DE `notify()`, `wait()` E `notifyAll()`

Considere a situação a seguir: uma thread chamada T está sendo executada dentro de um método sincronizado e precisa de acesso a um recurso chamado R que, por enquanto, está indisponível. O que T deve fazer? Se entrar em algum tipo de laço de sondagem à espera de R, T bloqueará o objeto, impedindo que outras threads o acessem. Essa não é uma solução ótima, porque invalida parcialmente as vantagens de programar em um ambiente com várias threads. Uma solução melhor é fazer T abandonar temporariamente o controle do objeto, permitindo que outra thread seja executada. Quando R estiver disponível, T pode ser notificada e retomar a execução. Essa abordagem se baseia em alguma forma de comunicação entre threads em que uma thread pode notificar outra que está bloqueada e ser notificada posteriormente para retomar a execução. Java dá suporte à comunicação entre threads com os métodos `wait()`, `notify()` e `notifyAll()`.

Os métodos `wait()`, `notify()` e `notifyAll()` fazem parte de todos os objetos porque são implementados pela classe **Object**. Esses métodos só podem ser chamados de dentro de um contexto **synchronized**. É assim que são usados: quando a execução de uma thread é bloqueada temporariamente, ela chama `wait()`. Isso faz a thread entrar em suspensão e o monitor desse objeto ser liberado, permitindo que outra thread use o objeto. A thread em suspensão poderá ser ativada posteriormente quando outra thread entrar no mesmo monitor e chamar `notify()` ou `notifyAll()`.

A seguir, temos as diversas formas de `wait()` definidas por **Object**:

```
final void wait() throws InterruptedException
final void wait(long millis) throws InterruptedException
final void wait(long millis, int nanos) throws InterruptedException
```

A primeira forma espera até haver uma notificação. A segunda espera até haver uma notificação ou o período especificado em milissegundos expirar. A terceira forma permite a especificação do período de espera em milissegundos e nanossegundos.

Estas são as formas gerais de `notify()` e `notifyAll()`:

```
final void notify()
final void notifyAll()
```

Uma chamada a `notify()` retoma a execução de uma thread que estava esperando. Uma chamada a `notifyAll()` notifica todas as threads, com a de prioridade mais alta ganhando acesso ao objeto.

Antes de examinarmos um exemplo que use `wait()`, é preciso fazer uma observação importante. Embora normalmente `wait()` espere até `notify()` ou `notifyAll()` ser chamado, em casos muito raros há a possibilidade de a thread que está esperando ser ativada devido a uma *ativação falsa*. As condições que levam a uma ativação falsa são complexas e não fazem parte do escopo deste livro. No entanto, a Oracle recomenda que, devido à possibilidade remota de ativação falsa, chamadas a `wait()` ocorram dentro de um laço que verifique a condição que a thread está esperando. O próximo exemplo mostra essa técnica.

Exemplo que usa `wait()` e `notify()`

Para entender a necessidade e a aplicação de `wait()` e `notify()`, criaremos um programa que simula o tique-taque de um relógio exibindo as palavras “Tick” e “Tock” na

tela. Para fazê-lo, criaremos uma classe chamada **TickTock** contendo dois métodos: **tick()** e **tock()**. O método **tick()** exibe a palavra “Tick” e **tock()** exibe “Tock”. Para o relógio ser executado, duas threads são criadas, uma que chama **tick()** e outra que chama **tock()**. O objetivo é fazer as duas threads serem executadas de maneira que a saída do programa exiba um “tique-taque” coerente – isto é, um padrão repetido de um tique seguido por um taque.

```
// Usa wait() e notify() para simular um relógio funcionando.

class TickTock {

    String state; // contém o estado do relógio

    synchronized void tick(boolean running) {
        if(!running) { // interrompe o relógio
            state = "ticked";
            notify(); // notifica qualquer thread que estiver esperando
            return;
        }

        System.out.print("Tick ");

        state = "ticked"; // define o estado atual com ticked

        notify(); // permite que tock() seja executado ←— tick() notifica tock().
        try {
            while(!state.equals("tocked"))
                wait(); // espera tock() terminar ←— tick() espera tock().
        }
        catch(InterruptedException exc) {
            System.out.println("Thread interrupted.");
        }
    }

    synchronized void tock(boolean running) {
        if(!running) { // interrompe o relógio
            state = "tocked";
            notify(); // notifica qualquer thread que estiver esperando
            return;
        }

        System.out.println("Tock");

        state = "tocked"; // define o estado atual com tocked

        notify(); // permite que tick() seja executado ←— tock() notifica tick().
        try {
            while(!state.equals("ticked"))
                wait(); // espera tick() terminar ←— tock() espera tick().
        }
        catch(InterruptedException exc) {
```

```

        System.out.println("Thread interrupted.");
    }
}
}

class MyThread implements Runnable {
    Thread thrd;
    TickTock ttOb;

    // Constrói uma nova thread.
    MyThread(String name, TickTock tt) {
        thrd = new Thread(this, name);
        ttOb = tt;
        thrd.start(); // inicia a thread
    }

    // Começa a execução da nova thread.
    public void run() {

        if(thrd.getName().compareTo("Tick") == 0) {
            for(int i=0; i<5; i++) ttOb.tick(true);
            ttOb.tick(false);
        }
        else {
            for(int i=0; i<5; i++) ttOb.tock(true);
            ttOb.tock(false);
        }
    }
}

class ThreadCom {
    public static void main(String[] args) {
        TickTock tt = new TickTock();
        MyThread mt1 = new MyThread("Tick", tt);
        MyThread mt2 = new MyThread("Tock", tt);

        try {
            mt1.thrd.join();
            mt2.thrd.join();
        } catch(InterruptedException exc) {
            System.out.println("Main thread interrupted.");
        }
    }
}

```

Aqui está a saída produzida pelo programa:

```

Tick Tock
Tick Tock
Tick Tock
Tick Tock
Tick Tock

```

Examinemos detalhadamente esse programa. A parte central do relógio é a classe **TickTock**. Ela contém dois métodos, **tick()** e **tock()**, que se comunicam para assegurar que um tique seja sempre seguido de um taque, que é sempre seguido por um tique e assim por diante. Observe o campo **state**. Quando o programa está sendo executado, **state** contém o string “ticked” ou “tocked”, que indica o estado atual do relógio. Em **main()**, um objeto **TickTock** chamado **tt** é criado e então usado para iniciar duas threads de execução.

As threads são baseadas em objetos de tipo **MyThread**. O construtor de **MyThread** recebe dois argumentos. O primeiro passa a ser o nome da thread. Ele será “Tick” ou “Tock”. O segundo é uma referência ao objeto **TickTock**, que é **tt** nesse caso. Dentro do método **run()** de **MyThread**, se o nome da thread for “Tick”, chamadas a **tick()** serão feitas. Se o nome da thread for “Tock”, o método **tock()** será chamado. Cinco chamadas que passam **true** como argumento são feitas a cada método. O relógio será executado enquanto **true** for passado. Uma chamada final que passa **false** a cada método interrompe o relógio.

A parte mais importante do programa se encontra nos métodos **tick()** e **tock()** de **TickTock**. Começaremos com o método **tick()**, que, por conveniência, é mostrado novamente aqui.

```
synchronized void tick(boolean running) {  
    if(!running) { // interrompe o relógio  
        state = "ticked";  
        notify(); // notifica qualquer thread que estiver esperando  
        return;  
    }  
  
    System.out.print("Tick ");  
  
    state = "ticked"; // define o estado atual com ticked  
  
    notify(); // permite que tock( ) seja executado  
    try {  
        while(!state.equals("tocked"))  
            wait(); // espera tock( ) terminar  
    }  
    catch(InterruptedException exc) {  
        System.out.println("Thread interrupted.");  
    }  
}
```

Primeiro, observe que **tick()** é modificado por **synchronized**. Lembre-se, **wait()** e **notify()** só são aplicáveis a métodos e blocos sincronizados. O método começa verificando o valor do parâmetro **running**. Esse parâmetro é usado para fornecer o desligamento normal do relógio. Se for **false**, o relógio foi desligado. Nesse caso, **state** será configurada com “ticked” e uma chamada a **notify()** será feita para permitir que threads em espera sejam executadas. Voltaremos a esse ponto em breve.

Supondo que o relógio esteja funcionando quando **tick()** for executado, a palavra “Tick” será exibida, **state** será configurada com “ticked” e uma chamada a **notify()** ocorrerá. A chamada a **notify()** permite que uma thread esperando no mes-

mo objeto seja executada. Em seguida, **wait()** é chamado dentro de um laço **while**. A chamada a **wait()** faz **tick()** ser suspenso até outra thread chamar **notify()**. Logo, o laço não iterará até que outra thread chame **notify()** no mesmo objeto. Como resultado, quando **tick()** é chamado, ele exibe um “Tick”, permite que outra thread seja executada e então entra em suspensão.

O laço **while** que chama **wait()** verifica o valor de **state**, esperando que seja “ticked”, o que só ocorrerá após o método **tock()** ser executado. Como explicado, o uso de um laço **while** para verificar essa condição impede que uma ativação falsa reinicie a thread incorretamente. Se **state** não contiver “ticked” quando **wait()** voltar, uma ativação falsa ocorreu e **wait()** será chamado novamente.

O método **tock()** é uma cópia exata de **tick()**, exceto por exibir “Tock” e configurar **state** com “ticked”. Logo, quando alcançado, ele exibe “Tock”, chama **notify()** e espera. Se vistos como um par, uma chamada a **tick()** só pode ser seguida por uma chamada a **tock()**, que só pode ser seguida por uma chamada a **tick()** e assim por diante. Portanto, os dois métodos são mutuamente sincronizados.

A razão da chamada a **notify()** quando o relógio é interrompido é permitir que uma chamada final a **wait()** seja bem-sucedida. Lembre-se, tanto **tick()** quanto **tock()** executam uma chamada a **wait()** após exibir sua mensagem. O problema é que, quando o relógio for interrompido, um dos métodos ainda estará esperando. Logo, uma chamada final a **notify()** é necessária para o método em espera ser executado. Como teste, tente remover a chamada a **notify()** e veja o que acontece. Como você verá, o programa “travará” e será preciso pressionar CTRL-C para sair. Isso ocorre porque, quando a chamada final a **tock()** chama **wait()**, não há uma chamada correspondente a **notify()** que permita que **tock()** seja concluído. Portanto, **tock()** fica apenas ali, esperando para sempre.

Antes de prosseguir, se tiver alguma dúvida sobre se as chamadas a **wait()** e **notify()** são realmente necessárias para fazer o “relógio” funcionar direito, insira a seguinte versão de **TickTock** no programa anterior. Nela, todas as chamadas a **wait()** e **notify()** foram removidas.

```
// Nenhuma chamada a wait() ou notify().
class TickTock {

    String state; // contém o estado do relógio

    synchronized void tick(boolean running) {
        if(!running) { // interrompe o relógio
            state = "ticked";
            return;
        }

        System.out.print("Tick ");
        state = "ticked"; // define o estado atual com ticked
    }

    synchronized void tock(boolean running) {
        if(!running) { // interrompe o relógio
    
```

```

        state = "locked";
        return;
    }

    System.out.println("Tock");

    state = "locked"; // define o estado atual com locked
}
}

```

Após a substituição, a saída produzida pelo programa será esta:

```

Tick Tick Tick Tick Tick Tock
Tock
Tock
Tock
Tock

```

Fica claro que os métodos **tick()** e **tock()** não estão mais trabalhando em conjunto!

Verificação do progresso

1. Que métodos dão suporte à comunicação entre threads?
2. Todos os objetos dão suporte à comunicação entre threads?
3. O que acontece quando **wait()** é chamado?

Pergunte ao especialista

P Vi o termo *deadlock* ser aplicado a programas *multithread* com comportamento incorreto. O que é e como posso evitar? Além disso, o que é uma *condição de corrida* e como também posso evitá-la?

R Deadlock é, como o nome sugere, uma situação em que uma thread está esperando outra thread fazer algo, mas essa outra thread está esperando a primeira. Logo, as duas threads estão suspensas, esperando uma pela outra, e nenhuma é executada. Essa situação é análoga à de duas pessoas muito polidas, ambas insistindo que a outra passe primeiro pela porta!

Parece fácil evitar deadlocks, mas não é. Por exemplo, pode ocorrer um deadlock de maneira indireta. Com frequência, não conhecemos a causa do deadlock apenas olhando o código-fonte do programa, porque threads sendo executadas ao mesmo tempo podem interagir de maneiras complexas no tempo de execução. Para evitar deadlock, precisamos de uma programação cuidadosa e testes abrangentes. Lembre-se, se um programa com várias threads “travar” ocasionalmente, a causa provável é um deadlock.

Respostas:

1. Os métodos de comunicação entre threads são **wait()**, **notify()** e **notifyAll()**.
2. Sim, todos os objetos dão suporte à comunicação entre threads porque esse suporte faz parte de **Object**.
3. Quando **wait()** é chamado, a thread chamadora abandona o controle do objeto e fica suspensa até receber uma notificação.

Uma condição de corrida ocorre quando duas (ou mais) threads tentam acessar um recurso compartilhado ao mesmo tempo, sem sincronização apropriada. Por exemplo, uma thread poderia estar gravando um novo valor em uma variável ao mesmo tempo em que outra estaria aumentando seu valor atual. Sem sincronização, o novo valor da variável dependerá da ordem em que as threads forem executadas. (A segunda thread incrementará o valor original ou o novo valor gravado pela primeira thread?) Em situações como essa, diz-se que as duas threads estão “disputando corrida”, com o resultado final sendo determinado pela thread que terminar primeiro. Como no deadlock, uma condição de corrida pode ocorrer de maneiras difíceis de descobrir. A solução é a prevenção: uma programação cuidadosa que sincronize apropriadamente o acesso a recursos compartilhados.

SUSPENDENDO, RETOMANDO E ENCERRANDO THREADS

Às vezes é útil suspender a execução de uma thread. Por exemplo, uma thread separada pode ser usada para exibir a hora do dia. Se o usuário não quiser um relógio, sua thread pode ser suspensa. Seja qual for o caso, é uma simples questão de suspender uma thread. Uma vez suspensa, também só temos de reiniciá-la.

O mecanismo de suspensão, encerramento e retomada de threads difere entre as versões antigas de Java e as versões mais modernas, a partir de Java 2. Antes de Java 2, os programas usavam **suspend()**, **resume()** e **stop()**, que são métodos definidos por **Thread**, para pausar, reiniciar e encerrar a execução de uma thread. Embora esses métodos pareçam uma abordagem perfeitamente sensata e conveniente para o gerenciamento da execução de threads, eles não devem mais ser usados. Vejamos o porquê. O método **suspend()** da classe **Thread** foi substituído em Java 2. Isso foi feito porque, às vezes, **suspend()** pode causar problemas sérios que envolvem deadlock. O método **resume()** também foi substituído; ele não pode ser usado sem o método **suspend()** como complemento. O método **stop()** da classe **Thread** também foi substituído em Java 2. A razão é que esse método às vezes também pode causar problemas sérios.

Já que agora não é possível usar os métodos **suspend()**, **resume()** ou **stop()** para controlar uma thread, à primeira vista você pode achar que não há uma maneira de pausar, reiniciar ou encerrar threads. No entanto, felizmente, esse não é o caso. A thread deve ser projetada de modo que o método **run()** verifique periodicamente se ela deve suspender, retomar ou encerrar sua própria execução. Normalmente, isso pode ser feito com o estabelecimento de duas variáveis *flag*: uma para suspender e retomar e outra para encerrar. Para a suspensão e retomada, se a *flag* estiver configurada com “em execução”, o método **run()** deve continuar permitindo que a thread seja executada. Se essa variável for configurada com “suspender”, a thread deve pausar. Quanto à flag de encerramento, se ela for configurada com “encerrar”, a thread deve terminar.

O exemplo a seguir mostra uma maneira de implementar suas próprias versões de **suspend()**, **resume()** e **stop()**:

```
// Suspendendo, retomando e encerrando uma thread.

class MyThread implements Runnable {
    Thread thrd;
```

```

boolean suspended; ← Suspende a thread quando igual a true.
boolean stopped; ← Encerra a thread quando igual a true.

MyThread(String name) {
    thrd = new Thread(this, name);
    suspended = false;
    stopped = false;
    thrd.start();
}

// Este é o ponto de entrada da thread.
public void run() {
    System.out.println(thrd.getName() + " starting.");
    try {
        for(int i = 1; i < 1000; i++) {
            System.out.print(i + " ");
            if((i%10)==0)
                System.out.println();
            Thread.sleep(250);
        }

        // Usa um bloco sincronizado para verificar suspended e stopped.
        synchronized(this) {
            while(suspended) { ←
                wait(); }
                if(stopped) break;
            }
        }
    } catch (InterruptedException exc) {
        System.out.println(thrd.getName() + " interrupted.");
    }
    System.out.println(thrd.getName() + " exiting.");
}

// Encerra a thread.
synchronized void myStop() {
    stopped = true;

    // O código a seguir assegura que uma thread suspensa possa ser
    // encerrada.
    suspended = false;
    notify();
}

// Suspende a thread.
synchronized void mySuspend() {
    suspended = true;
}

```

Este bloco sincronizado verifica
suspended e **stopped**.

```
// Retoma a thread.
synchronized void myResume() {
    suspended = false;
    notify();
}

class Suspend {
    public static void main(String[] args) {
        MyThread ob1 = new MyThread("My Thread");

        try {
            Thread.sleep(1000); // permite que a thread ob1 comece a ser executada

            ob1.mySuspend();
            System.out.println("Suspending thread.");
            Thread.sleep(1000);

            ob1.myResume();
            System.out.println("Resuming thread.");
            Thread.sleep(1000);

            ob1.mySuspend();
            System.out.println("Suspending thread.");
            Thread.sleep(1000);

            ob1.myResume();
            System.out.println("Resuming thread.");
            Thread.sleep(1000);

            ob1.mySuspend();
            System.out.println("Stopping thread.");
            ob1.myStop();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        // espera a thread terminar
        try {
            ob1.thrd.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}
```

Um exemplo da saída desse programa é mostrado abaixo. (Você pode obter uma saída um pouco diferente.)

```

My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Suspending thread.
Resuming thread.
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Suspending thread.
Resuming thread.
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Stopping thread.
My Thread exiting.
Main thread exiting.

```

É assim que o programa funciona. A classe de threads **MyThread** define duas variáveis booleanas, **suspended** e **stopped**, que controlam a suspensão e o encerramento de uma thread. Ambas são inicializadas com **false** pelo construtor. O método **run()** contém um bloco de instruções **synchronized** que verifica **suspended**. Se essa variável for **true**, o método **wait()** será chamado para suspender a execução da thread. Para suspender a execução da thread, chame **mySuspend()**, que configura **suspended** com **true**. Para retomar a execução, chame **myResume()**, que configura **suspended** com **false** e chama **notify()** para reiniciar a thread.

Para encerrar a thread, chame **myStop()**, que configura **stopped** com **true**. O método **myStop()** também configura **suspended** com **false** e então chama **notify()**. Essas etapas são necessárias para o encerramento de uma thread suspensa.

Pergunte ao especialista

P O uso de várias threads parece uma ótima maneira de melhorar a eficiência de meus programas. Pode me dar algumas dicas de como usá-las de maneira eficaz?

R O segredo para o uso eficiente de várias threads é pensar concorrentemente em vez de sequencialmente. Por exemplo, se você tiver dois subsistemas totalmente independentes dentro de um programa, considere transformá-los em threads individuais. No entanto, é preciso tomar cuidado. Se você criar threads demais, pode piorar o desempenho de seu programa em vez de melhorá-lo. Lembre-se, a sobrecarga está associada à mudança de contexto. Se você criar threads demais, mais tempo da CPU será gasto com mudanças de contexto do que na execução de seu programa!

TENTE ISTO 12-2 Usando a thread principal

UseMain.java

Todos os programas Java têm pelo menos uma thread de execução, chamada *thread principal*, que é fornecida ao programa automaticamente quando ele começa a ser executado. Até agora, usamos a thread principal sem lhe dar destaque. Neste projeto, veremos que ela pode ser tratada como todas as outras threads.

PASSO A PASSO

1. Crie um arquivo chamado **UseMain.java**.
2. Para acessar a thread principal, você deve obter um objeto **Thread** que a referencie. Isso é feito com uma chamada ao método **currentThread()**, que é membro **static** de **Thread**. Sua forma geral é mostrada aqui:

```
static Thread currentThread()
```

Esse método retorna uma referência à thread em que é chamado. Logo, se você chamar **currentThread()** enquanto a execução estiver dentro da thread principal, obterá uma referência a essa thread. Uma vez que tiver essa referência, poderá controlar a thread principal como qualquer outra thread.

3. Insira o programa a seguir no arquivo. Ele obtém uma referência à thread principal e então acessa e define seu nome e exibe sua prioridade.

```
/*
 * Tente isto 12-2
 *
 * Controlando a thread principal.
 */

class UseMain {
    public static void main(String[] args) {
        Thread thrd;

        // Acessa a thread principal.
        thrd = Thread.currentThread();

        // Exibe o nome da thread principal.
        System.out.println("Main thread is called: " +
                           thrd.getName());

        // Exibe a prioridade da thread principal.
        System.out.println("Priority: " +
                           thrd.getPriority());

        System.out.println();

        // Define o nome.
    }
}
```

```

        System.out.println("Setting name.\n");
        thrd.setName("Thread #1");

        System.out.println("Main thread is now called: " +
                           thrd.getName());
    }
}

```

4. A saída do programa é mostrada abaixo:

```

Main thread is called: main
Priority: 5

Setting name.

Main thread is now called: Thread #1

```

5. Você deve tomar cuidado com as operações executadas na thread principal. Por exemplo, se adicionar o código a seguir ao fim de **main()**, o programa nunca terminará, porque a thread principal ficará esperando seu próprio encerramento!

```

try {
    thrd.join();
} catch(InterruptedException exc) {
    System.out.println("Interrupted");
}

```

EXERCÍCIOS

- Como o uso de várias threads Java nos permite escrever programas mais eficientes?
- O uso de várias threads é suportado pela classe _____ e pela interface _____.
- Na criação de um objeto executável, por que pode ser melhor estender **Thread** em vez de implementar **Runnable**?
- Mostre como podemos usar **join()** para esperar um objeto de thread chamado **MyThrd** terminar.
- Mostre como configurar uma thread chamada **MyThrd** com três níveis acima da prioridade normal.
- Qual é o efeito da inclusão da palavra-chave **synchronized** em um método?
- Os métodos **wait()** e **notify()** são usados na execução da _____.
- Altere a classe **TickTock** para que ela marque realmente o tempo. Isto é, faça cada tique levar meio segundo e cada taque levar mais meio segundo. Logo, cada tique-taque levará um segundo. (Não se preocupe com o tempo necessário para alternar tarefas, etc.)
- Por que você não pode usar **suspend**, **resume()** e **stop()** em programas novos?

10. Que método definido por **Thread** obtém o nome de uma thread?
11. O que **isAlive()** retorna?
12. Se sincronizando métodos podemos evitar que ocorram problemas de acesso concorrente, por que eles não são sincronizados automaticamente?
13. No fim deste capítulo, todos os métodos **main()** dos exemplos terminavam com um bloco **try** contendo uma ou mais chamadas a **join()** em threads. Explique o porquê.
14. Em aplicativos que usam GUIs (interfaces gráficas de usuário com janelas, botões, menus, etc.), é importante que a interface responda sempre a cliques no mouse e pressionamentos de teclas. Mas às vezes um clique em um botão causa a execução de um método que precisa de muito tempo de processamento, como o de geração de uma imagem complexa ou de carga de um arquivo grande. Enquanto esse método está sendo executado, como podemos fazer a GUI continuar respondendo?
15. Se **t** fosse um **Thread**, **t.start()** chamaria **t.run()**? Isto é, o método **run()** de **t** começaria necessariamente a ser executado antes de **t.start()** retornar?
16. Suponhamos que você criasse uma subclasse **MyThread** de **Thread** em que o método **run()** apenas exibisse o nome da thread 50 vezes usando um laço e então retornasse. Suponhamos também que você criasse 20 objetos **MyThread** com nomes diferentes e começasse a executá-los.
 - A. Descreva qual seria a saída com a maior precisão possível.
 - B. Explique a diferença que veríamos, caso haja alguma, se o método **run()** fosse declarado como **synchronized** na classe **MyThread**.
 - C. Explique a diferença que veríamos, caso haja alguma, se o método **run()** fosse declarado como **synchronized** na classe **MyThread** e sua instrução de saída ficasse dentro do bloco sincronizado **synchronized(this) {...}**.
17. Nos exemplos de bloco sincronizado deste capítulo, sempre usamos o monitor associado a **this**. Podemos usar o monitor associado a algum objeto que não seja aquele cujos métodos estamos utilizando? Por exemplo, podemos criar um objeto sem métodos ou variáveis de instância e usar o monitor desse objeto?
18. Na seção deste capítulo chamada “A instrução **synchronized**”, demos um exemplo que usava as classes **SumArray**, **MyThread** e **Sync**. Em cada uma das variações a seguir, explique o que ocorreria se a alteração proposta fosse feita.
 - A. Adicionar um bloco **synchronized(this)** delimitando o laço **for** do método **sumArray()** de **SumArray**.
 - B. Remover o bloco sincronizado que delimita a instrução

```
| answer = sa.sumArray(a)
```

do corpo do método **run()** de **MyThread** e adicionar um bloco **synchronized(this)** delimitando o laço **for** do método **sumArray** de **SumArray**.

- C. Remover a palavra-chave **static** da frente da variável de instância **sa** de **MyThread**.

- D. Remover o bloco sincronizado que delimita a instrução

```
| answer = sa.sumArray(a)
```

do corpo do método **run()** de **MyThread** e adicionar o modificador **synchronized** à declaração desse método.

- 19.** Este exercício demonstra como dois métodos sincronizados têm que se comunicar para manter um contador entre 0 e 3 quando uma thread está tentando repetidamente aumentá-lo e a outra thread tenta repetidamente diminuí-lo. Ele usa o método **Math.random()** que não emprega parâmetros e retorna um valor **double** aleatório entre 0 e 1. Execute todas as etapas a seguir:

- A. Crie uma classe **Counter** com uma variável de instância privada **count** e dois métodos. O primeiro método

```
synchronized void increment( )
```

tenta incrementar **count** em 1 unidade. Se **count** já tiver atingido seu máximo, que é 3, ele esperará até **count** ser menor do que 3 antes de incrementá-la. O outro método

```
synchronized void decrement( )
```

tenta diminuir **count** em 1 unidade. Se **count** já tiver atingido seu mínimo, que é 0, ele esperará até **count** ser maior do que 0 antes de decrementá-la. Sempre que um dos métodos tem que esperar, ele exibe uma declaração dizendo por que está esperando. Além disso, sempre que ocorre um incremento ou decremeno, o contador exibe uma declaração informando o que ocorreu e mostra o novo valor de **count**.

- B. Crie uma classe de thread cujo método **run()** chame o método **increment()** de **Counter** 20 vezes. Entre cada chamada, ele entra em suspensão por um período de tempo aleatório entre 0 e 500 milissegundos.
- C. Crie uma classe de thread cujo método **run()** chame o método **decrement()** de **Counter** 20 vezes. Entre cada chamada, ele entra em suspensão por um período de tempo aleatório entre 0 e 500 milissegundos.
- D. Crie uma classe **CounterUser** com um método **main()** que gere um **Counter** e as duas threads e comece a executá-las.

Nota: Em vez de criar duas classes de threads, você pode criar apenas uma classe de thread que incremente ou decremente o contador de acordo com um parâmetro passado para seu construtor.

- 20.** O programa a seguir testa a eficiência das threads. Ele deve ser executado com dois argumentos de linha de comando. O primeiro argumento é o tamanho do array que será classificado. O segundo é quantas vezes (**m**) ele será classificado. Primeiro o programa cria um array com o tamanho especificado e então o copia e classifica **m** vezes consecutivamente usando apenas uma thread. Em seguida, inicia **m** threads em paralelo, cada uma fazendo uma cópia e classificação.

Insira o programa em seu computador e execute-o com vários argumentos, cronometrando-o para ver o aumento que as **m** threads podem dar à velocidade ao classificar em paralelo. Será útil selecionar um tamanho suficiente para o array e para **m** de modo que as classificações paralelas demorem 10 ou mais segundos. Escreva um resumo de seus resultados, incluindo uma explicação de qualquer diferença que observar entre as **m** classificações consecutivas e as classificações paralelas com o uso de **m** threads.

```
// Testa a eficiência das threads.

class ThreadSpeedUp {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("Usage: ThreadSpeedUp size numReps");
            return;
        }

        // cria todos os dados e as threads
        int[] data = new int[Integer.parseInt(args[0])];
        for (int i = 0; i < data.length; i++)
            data[i] = data.length - i;

        int numReps = Integer.parseInt(args[1]);
        Thread[] threads = new Thread[numReps];
        for(int i = 0; i < numReps; i++)
            threads[i] = new CopyAndSortThread(data);

        // agora classifica consecutivamente em uma thread
        System.out.println("Starting sorting array of length " + args[0] +
                           " in one thread " + numReps + " times.");

        for (int i = 0; i < numReps; i++) {
            copyAndSort(data);
        }

        System.out.println("Done sorting in one thread. ");
        System.out.println("Starting sorting arrays of length " +
                           args[0] + " using " + numReps + " threads.");

        // inicia todas as threads classificando em paralelo
        for(Thread thd : threads)
            thd.start();

        // espera todas as threads terminarem
        try {
            for(Thread thd: threads)
                thd.join();
        }
    }
}
```

```
        catch (InterruptedException e) {
            System.out.println("InterruptedException occurred.");
        }
        System.out.println("Done sorting using " + numReps + " threads.");
    }

    static void copyAndSort(int[] data) {
        // copia os dados em um novo array
        int[] nums = new int[data.length];

        for (int j = 0; j < data.length; j++)
            nums[j] = data[j];

        // agora classifica o novo array usando a classificação de bolha
        for (int a = 1; a < nums.length; a++) {
            for (int b = nums.length - 1; b >= a; b--) {
                if (nums[b - 1] > nums[b]) { // troca se estiver fora de ordem
                    int t = nums[b - 1];
                    nums[b - 1] = nums[b];
                    nums[b] = t;
                }
            }
        }
    }

    class CopyAndSortThread extends Thread {
        int[] data;

        CopyAndSortThread(int[] d) {
            data = d;
        }

        public void run() {
            ThreadSpeedUp.copyAndSort(data);
        }
    }
}
```

Enumerações, autoboxing e anotações

PRINCIPAIS HABILIDADES E CONCEITOS

- Entender os fundamentos da enumeração
- Usar os recursos de enumeração baseados em classes
- Aplicar os métodos **values()** e **valueof()** a enumerações
- Criar enumerações que tenham construtores, variáveis de instância e métodos
- Empregar os métodos **ordinal()** e **compareTo()** que as enumerações herdam de **Enum**
- Usar os encapsuladores de tipos Java
- Saber os aspectos básicos do *autoboxing* e *autounboxing*
- Usar *autoboxing* com métodos
- Entender como *autoboxing* funciona com expressões
- Ter uma visão geral das anotações

Este capítulo discutirá três recursos que são relativamente novos em Java: as enumerações, o *autoboxing* e as anotações. Embora não façam parte da especificação de original Java, todos expandem o poder e a usabilidade da linguagem. No caso das enumerações e do *autoboxing*, ambos otimizam a linguagem, simplificando certas estruturas comuns. As anotações expandem os tipos de informações que podem ser embutidos dentro de um arquivo-fonte. Coletivamente, todos oferecem uma maneira melhor de resolver problemas comuns de programação. Também serão discutidos neste capítulo os encapsuladores de tipos Java por terem relação com o *autoboxing*.

ENUMERAÇÕES

Embora a enumeração seja um recurso de programação comum que é encontrado em muitas outras linguagens de computador, ela não fazia parte da especificação Java original. Isso ocorreu porque a enumeração é tecnicamente uma conveniência, e não uma necessidade. No entanto, com o passar dos anos, muitos programadores quiseram que Java desse suporte às enumerações, porque elas oferecem uma solução ele-

gante e estruturada para várias tarefas de programação. Essa solicitação foi atendida com o lançamento do JDK 5, que adicionou as enumerações à Java.

Em sua forma mais simples, uma *enumeração* é uma lista de constantes nomeadas que define um novo tipo de dado. Um objeto de um tipo de enumeração só pode conter os valores definidos pela lista. Logo, uma enumeração fornece uma maneira de definirmos precisamente um novo tipo de dado que tem um número fixo de valores válidos.

As enumerações são comuns no dia a dia. Por exemplo, uma enumeração das moedas usadas nos Estados Unidos teria *penny*, *nickel*, *dime*, *half-dollar* e *dollar*. Uma enumeração dos meses do ano seria composta pelos nomes que vão de janeiro a dezembro. Uma enumeração dos dias da semana conteria domingo, segunda, terça, quarta, quinta, sexta e sábado.

Do ponto de vista da programação, as enumerações são úteis sempre que precisamos definir um conjunto de valores que representam um grupo de itens. Por exemplo, podemos usar uma enumeração para representar um conjunto de códigos de status, como sucesso, em espera, falha e nova tentativa, que indiquem o progresso de uma transferência de arquivo. No passado, esses valores eram definidos com variáveis de tipo **final**, mas as enumerações oferecem uma abordagem mais estruturada.

Fundamentos da enumeração

Uma enumeração é criada com o uso da palavra-chave **enum**. Por exemplo, aqui está uma enumeração simples que lista vários meios de transporte:

```
// Enumeração de meios de transporte.  
enum Transport {  
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT  
}
```

Os identificadores **CAR**, **TRUCK**, etc., são chamados de *constantes de enumeração*, ou *constantes enum*, na abreviação. Cada identificador é declarado implicitamente como membro público estático de **Transport**. Além disso, o tipo das constantes de enumeração é o mesmo da enumeração em que elas são declaradas, que nesse caso é **Transport**. Logo, essas constantes também são chamadas de *autotipadas*, em que “auto” representa a enumeração que as contêm.

Uma vez que você tiver declarado uma enumeração, poderá criar uma variável desse tipo. No entanto, ainda que as enumerações definam um tipo de classe, você não pode instanciar uma **enum** usando **new**. Em vez disso, deve declarar e usar uma variável de enumeração de maneira semelhante ao que faria com os tipos primitivos. Por exemplo, esta linha declara **tp** como uma variável de tipo de enumeração **Transport**:

```
| Transport tp;
```

Como **tp** é de tipo **Transport**, os únicos valores que ela pode receber são os definidos pela enumeração ou **null**. Por exemplo, esta linha atribui a **tp** o valor **AIRPLANE**:

```
| tp = Transport.AIRPLANE;
```

Observe que o símbolo **AIRPLANE** é qualificado por **Transport**.

Duas constantes de enumeração podem ser comparadas em busca de igualdade com o uso do operador relacional `==`. Por exemplo, esta instrução compara o valor de `tp` com a constante `TRAIN`:

```
| if(tp == Transport.TRAIN) // ...
```

O valor de uma enumeração também pode ser usado no controle de uma instrução `switch`. Obviamente, todas as instruções `case` devem usar constantes da mesma `enum` usada pela expressão `switch`. Por exemplo, este `switch` é perfeitamente válido:

```
// Usa uma enum para controlar uma instrução switch.
switch(tp) {
    case CAR:
        // ...
    case TRUCK:
        // ...
```

Observe que, nas instruções `case`, os nomes das constantes de enumeração são usados sem qualificação pelo nome do tipo de enumeração, isto é, `TRUCK`, e não `Transport.TRUCK`, é usado. Isso ocorre porque o tipo da enumeração na expressão `switch` já especificou implicitamente o tipo `enum` das constantes `case`. Não há necessidade de qualificar as constantes nas instruções `case` com o nome de seu tipo `enum`. Na verdade, tentar fazê-lo causará um erro de compilação.

Quando uma constante de enumeração é exibida, como em uma instrução `println()`, seu nome compõe a saída. Por exemplo, dada a seguinte instrução:

```
| System.out.println(Transport.BOAT);
```

o nome **BOAT** é exibido.

O programa a seguir reúne todas as peças e demonstra a enumeração `Transport`.

```
// Enumeração de meios de transporte.
enum Transport {
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT ← Declara uma enumeração.
}

class EnumDemo {
    public static void main(String[] args)
    {
        Transport tp; ← Declara uma referência Transport.

        tp = Transport.AIRPLANE; ← Atribui a tp a constante AIRPLANE.

        // Exibe um valor da enum.
        System.out.println("Value of tp: " + tp);
        System.out.println();

        tp = Transport.TRAIN;

        // Compara dois valores da enum.
        if(tp == Transport.TRAIN) ← Compares dois objetos Transport
            System.out.println("tp contains TRAIN.\n"); em busca de igualdade.
```

```
// Usa uma enum para controlar uma instrução switch.
switch(tp) { ← Usa uma enumeração para controlar
    case CAR:           uma instrução switch.
        System.out.println("A car carries people.");
        break;
    case TRUCK:
        System.out.println("A truck carries freight.");
        break;
    case AIRPLANE:
        System.out.println("An airplane flies.");
        break;
    case TRAIN:
        System.out.println("A train runs on rails.");
        break;
    case BOAT:
        System.out.println("A boat sails on water.");
        break;
}
}
}
```

A saída do programa é mostrada aqui:

```
Value of tp: AIRPLANE
```

```
tp contains TRAIN.
```

```
A train runs on rails.
```

Antes de prosseguirmos, é preciso ressaltar uma questão estilística. As constantes de **Transport** usam maiúsculas. (Logo, usamos **CAR** e não **car**.) No entanto, o uso de maiúsculas não é obrigatório, ou seja, não há uma regra que exija que as constantes de enumeração estejam em maiúsculas. Porém, esse é o estilo normalmente usado por programadores de Java. (É claro que há outros pontos de vista e estilos.) Os exemplos deste livro usarão maiúsculas nas constantes de enumeração, a título de consistência.

Verificação do progresso

1. Uma enumeração define uma lista de constantes _____.
2. Que palavra-chave declara uma enumeração?
3. Dado o código

```
enum Directions {
    LEFT, RIGHT, UP, DOWN
}
```

qual é o tipo de dado de **UP**?

Respostas:

1. nomeadas
2. **enum**
3. O tipo de dado de **UP** é **Directions** porque as constantes enumeradas são autotipadas.

AS ENUMERAÇÕES JAVA SÃO TIPOS DE CLASSE

Embora os exemplos anteriores mostrem o mecanismo de criação e uso de uma enumeração, eles não mostram todos os seus recursos. Isso ocorre porque Java implementa enumerações como tipos de classe. Mesmo que não instanciemos uma **enum** usando **new**, seu comportamento é semelhante ao de outras classes. O fato de **enum** definir uma classe permite que a enumeração Java tenha poderes que de outra forma as enumerações não teriam. Por exemplo, podemos lhe dar construtores, adicionar métodos e variáveis de instância e até implementar interfaces.

MÉTODOS values() E valueOf()

Todas as enumerações têm automaticamente dois métodos predefinidos: **values()** e **valueOf()**. Suas formas gerais são mostradas aqui:

```
public static tipo-enum[ ] values()
public static tipo-enum valueOf(String str)
```

O método **values()** retorna um array contendo uma lista com as constantes de enumeração. O método **valueOf()** retorna a constante de enumeração cujo valor corresponde ao string passado em *str*. Nos dois casos, *tipo-enum* é o tipo da enumeração. Por exemplo, no caso da enumeração **Transport** mostrada anteriormente, o tipo de retorno de **Transport.valueOf("TRAIN")** é **Transport**. O valor retornado é **TRAIN**. O programa a seguir demonstra os métodos **values()** e **valueOf()**.

```
// Usa os métodos de enumeração internos.

// Enumeração de meios de transporte.
enum Transport {
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}

class EnumDemo2 {
    public static void main(String[] args)
    {
        Transport tp;

        System.out.println("Here are all Transport constants");

        // usa values()
        Transport[] allTransports = Transport.values(); ←
        for(Transport t : allTransports)
            System.out.println(t);
    }

    System.out.println();

    // usa valueOf()
    tp = Transport.valueOf("AIRPLANE"); ←
    System.out.println("tp contains " + tp);      Obtém a constante de
                                                nome AIRPLANE.
}
}
```

Obtém um array de constantes **Transport**.

A saída do programa é dada a seguir:

```
Here are all Transport constants
CAR
TRUCK
AIRPLANE
TRAIN
BOAT

tp contains AIRPLANE
```

Observe que esse programa usa um laço **for** de estilo for-each para percorrer o array de constantes obtido pela chamada a **values()**. A título de ilustração, a variável **allTransports** foi criada e recebeu uma referência ao array da enumeração. No entanto, essa etapa não é necessária porque **for** poderia ter sido escrito como mostrado aqui, eliminando a necessidade da variável **allTransports**:

```
for(Transport t : Transport.values())
    System.out.println(t);
```

Agora, observe como o valor correspondente ao nome **AIRPLANE** foi obtido pela chamada a **valueOf()**:

```
| tp = Transport.valueOf("AIRPLANE");
```

Como explicado, **valueOf()** retorna o valor da enumeração associado ao nome da constante representada como um string.

CONSTRUTORES, MÉTODOS, VARIÁVEIS DE INSTÂNCIA E ENUMERAÇÕES

É importante entender que cada constante de enumeração é um objeto de seu tipo de enumeração. Logo, uma enumeração pode definir construtores, adicionar métodos e ter variáveis de instância. Quando definimos um construtor para uma **enum**, ele é chamado quando cada constante de enumeração é criada. Cada constante pode chamar qualquer método definido pela enumeração e terá sua própria cópia de qualquer variável de instância também definida pela enumeração. A versão a seguir de **Transport** ilustra o uso de um construtor, uma variável de instância e um método. Ela atribui a cada meio de transporte uma velocidade típica.

```
// Usa um construtor, uma variável de instância e um método com a enumeração.
enum Transport {
    CAR(65), TRUCK(55), AIRPLANE(600), TRAIN(70), BOAT(22); // Observe os
                                                               valores de
                                                               inicialização.

    private int speed; // velocidade típica de cada meio de transporte // Adiciona uma variável
                                                               de instância.

    // Construtor
    Transport(int s) { speed = s; } // Adiciona um construtor.

    int getSpeed() { return speed; } // Adiciona um método.
}
```

```

class EnumDemo3 {
    public static void main(String[] args)
    {
        Transport tp;

        // Exibe a velocidade de um avião.
        System.out.println("Typical speed for an airplane is " +
                           Transport.AIRPLANE.getSpeed() + ←
                           " miles per hour.\n");           Obtém a velocidade
                                                chamando getSpeed().
        // Exibe todos os meios de transporte e velocidades.
        System.out.println("All Transport speeds: ");
        for(Transport t : Transport.values())
            System.out.println(t + " typical speed is " +
                               t.getSpeed() +
                               " miles per hour.");
    }
}

```

A saída é mostrada aqui:

```

Typical speed for an airplane is 600 miles per hour.

All Transport speeds:
CAR typical speed is 65 miles per hour.
TRUCK typical speed is 55 miles per hour.
AIRPLANE typical speed is 600 miles per hour.
TRAIN typical speed is 70 miles per hour.
BOAT typical speed is 22 miles per hour.

```

Essa versão de **Transport** adiciona três coisas. A primeira é a variável de instância **speed**, que é usada para conter a velocidade de cada meio de transporte. A segunda é o construtor de **Transport**, que recebe a velocidade de um meio de transporte. A terceira é o método **getSpeed()**, que retorna o valor de **speed**.

Quando a variável **tp** é declarada em **main()**, o construtor de **Transport** é chamado uma vez para cada constante especificada. Observe como os argumentos do construtor são especificados, sendo colocados em parênteses após cada constante, como mostrado abaixo:

```
| CAR(65), TRUCK(55), AIRPLANE(600), TRAIN(70), BOAT(22);
```

Esses valores são passados para o parâmetro **s** de **Transport()**, que então atribui o valor a **speed**. Há algo mais que devemos observar sobre a lista de constantes de enumeração: ela termina com um ponto e vírgula, isto é, a última constante, **BOAT**, é seguida por um ponto e vírgula. Quando uma enumeração contém outros membros, a lista deve terminar em um ponto e vírgula.

Uma vez que cada constante de enumeração tem sua própria cópia de **speed**, você pode obter a velocidade de um meio de transporte especificado chamando **getSpeed()**. Por exemplo, em **main()** a velocidade de um avião é obtida pela chamada a seguir:

```
| Transport.AIRPLANE.getSpeed()
```

A velocidade de cada meio de transporte é obtida quando percorremos a enumeração usando um laço **for**. Já que há uma cópia de **speed** para cada constante de enumeração, o valor associado a uma constante fica separado e é diferente do valor associado a outra constante. Esse é um conceito poderoso, que só está disponível quando enumerações são implementadas como classes, como ocorre em Java.

Embora o exemplo anterior só tenha um construtor, uma **enum** pode oferecer duas ou mais formas sobrecarregadas, como qualquer outra classe.

Pergunte ao especialista

P Já que as enumerações foram adicionadas a Java, devo evitar usar variáveis de tipo **final**? Em outras palavras, as enumerações tornaram as variáveis **final** obsoletas?

R Não. As enumerações são apropriadas quando trabalhamos com listas de itens que devem ser representados por identificadores. Uma variável **final** é apropriada quando temos um valor constante, como o tamanho de um array, que será usado em muitos locais. Logo, cada uma tem seu uso próprio. A vantagem das enumerações é que as variáveis de tipo **final** não são obrigadas a fazer um trabalho para o qual não são a opção ideal.

Duas restrições importantes

Há duas restrições aplicadas às enumerações. Em primeiro lugar, uma enumeração não pode herdar explicitamente outra classe. Em segundo lugar, uma **enum** não pode ser uma superclasse. Ou seja, uma **enum** não pode ser estendida, mas, por outro lado, terá um comportamento semelhante ao de qualquer outro tipo de classe. O segredo é lembrar que cada constante de enumeração é um objeto da enumeração em que é definida.

ENUMERAÇÕES HERDAM Enum

Apesar de não podermos herdar explicitamente uma superclasse ao declarar uma **enum**, todas as enumerações herdam uma implicitamente: **java.lang.Enum**. Essa classe define vários métodos que estão disponíveis para uso de todas as enumerações. Quase nunca precisamos usar esses métodos, mas há dois que podem ser interessantes: **ordinal()** e **compareTo()**.

O método **ordinal()** obtém um valor que indica a posição de uma constante de enumeração na lista de constantes. Ele é chamado de *valor ordinal*. O método **ordinal()** é mostrado aqui:

```
final int ordinal()
```

Ele retorna o valor ordinal da constante chamadora. Os valores ordinais começam em zero. Logo, na enumeração **Transport**, **CAR** tem valor ordinal zero, **TRUCK** tem valor ordinal 1, **AIRPLANE** tem valor ordinal 2 e assim por diante.

Você pode comparar o valor ordinal de duas constantes da mesma enumeração usando o método **compareTo()**. Ele tem a seguinte forma geral:

```
final int compareTo(tipo-enum e)
```

Aqui, *tipo-enum* é o tipo da enumeração e *e* é a constante que está sendo comparada à constante chamadora. Lembre-se, tanto a constante chamadora quanto *e* devem ser da mesma enumeração. Se a constante chamadora tiver valor ordinal menor do que o de *e*, **compareTo()** retornará um valor negativo. Se os dois valores ordinais forem iguais, zero será retornado. Se a constante chamadora tiver valor ordinal maior do que o de *e*, um valor positivo será retornado.

O programa a seguir demonstra **ordinal()** e **compareTo()**:

```
// Demonstra ordinal() e compareTo().

// Enumeração de meios de transporte.
enum Transport {
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}

class EnumDemo4 {
    public static void main(String[] args)
    {
        Transport tp, tp2, tp3;

        // Obtém todos os valores ordinais usando ordinal().
        System.out.println("Here are all Transport constants" +
            " and their ordinal values: ");
        for(Transport t : Transport.values())
            System.out.println(t + " " + t.ordinal()); ← Obtém os valores ordinais.

        tp = Transport.AIRPLANE;
        tp2 = Transport.TRAIN;
        tp3 = Transport.AIRPLANE;

        System.out.println();
        // Demonstra compareTo()
        if(tp.compareTo(tp2) < 0) ← Compara valores ordinais.
            System.out.println(tp + " comes before " + tp2);

        if(tp.compareTo(tp2) > 0)
            System.out.println(tp2 + " comes before " + tp);

        if(tp.compareTo(tp3) == 0)
            System.out.println(tp + " equals " + tp3);
    }
}
```

A saída do programa é mostrada abaixo:

```
Here are all Transport constants and their ordinal values:
CAR 0
TRUCK 1
AIRPLANE 2
TRAIN 3
```

```
BOAT 4

AIRPLANE comes before TRAIN
AIRPLANE equals AIRPLANE
```

Verificação do progresso

1. O que **values()** retorna?
2. Uma enumeração pode ter um construtor?
3. O que é o valor ordinal de uma constante de enumeração?

TENTE ISTO 13-1 Semáforo controlado por computador

`TrafficLightDemo.java`

As enumerações são particularmente úteis quando o programa precisa de um conjunto de constantes com valores reais e arbitrários, contanto que sejam diferentes. Esse tipo de situação surge com frequência quando programamos. Um caso comum envolve o tratamento dos estados em que algum dispositivo pode se encontrar. Por exemplo, suponhamos que estivéssemos escrevendo um programa para controlar um semáforo. O código do semáforo deve percorrer automaticamente os três estados do sinal: verde, amarelo e vermelho. Além disso, deve permitir que outro código saiba a cor atual do sinal e deixe a cor ser configurada com um valor inicial conhecido. Ou seja, os três estados devem ser representados de alguma forma. Embora possamos representar esses três estados com valores inteiros (por exemplo, os valores 1, 2 e 3) ou com strings (como “vermelho”, “verde” e “amarelo”), uma enumeração oferece uma abordagem muito melhor. O uso de uma enumeração resulta em código mais eficiente do que se strings representassem os estados e mais estruturado do que se inteiros os representassem.

Neste projeto, você criará a simulação de um semáforo automático, como o que acabou de ser descrito. O projeto demonstrará não só uma enumeração em ação, mas também outro exemplo do uso de várias threads e de sincronização.

Respostas:

1. O método **values()** retorna um array contendo uma lista com todas as constantes definidas pela enumeração chamadora.
2. Sim.
3. O valor ordinal de uma constante de enumeração descreve sua posição na lista de constantes, com a primeira constante tendo o valor ordinal zero.

PASSO A PASSO

1. Crie um arquivo chamado **TrafficLightDemo.java**.
2. Comece definindo uma enumeração chamada **TrafficLightColor** que represente os três estados do sinal, como mostrado aqui:

```
// Enumeração com as cores de um semáforo.
enum TrafficLightColor {
    RED, GREEN, YELLOW
}
```

Sempre que a cor do sinal for necessária, seu valor na enumeração será usado.

3. Em seguida, defina **TrafficLightSimulator**, como mostrado abaixo. **TrafficLightSimulator** é a classe que encapsula a simulação do semáforo.

```
// Semáforo computadorizado.
class TrafficLightSimulator implements Runnable {
    private Thread thrd; // contém a thread que executa a simulação
    private TrafficLightColor tlc; // contém a cor do sinal
    boolean stop = false; // configura com true para interromper a
                          // simulação
    boolean changed = false; // true quando o sinal mudou

    TrafficLightSimulator(TrafficLightColor init) {
        tlc = init;

        thrd = new Thread(this);
        thrd.start();
    }

    TrafficLightSimulator() {
        tlc = TrafficLightColor.RED;

        thrd = new Thread(this);
        thrd.start();
    }
}
```

Observe que **TrafficLightSimulator** implementa **Runnable**. Isso é necessário porque uma thread separada é usada na execução de cada sinal. Essa thread percorrerá as cores. Dois construtores são criados. O primeiro permite a especificação da cor inicial do semáforo, e o segundo tem como padrão o vermelho. Os dois iniciam uma nova thread para executar o semáforo. Agora, examine as variáveis de instância. Uma referência à thread do semáforo é armazenada em **thrd**. A cor atual do semáforo é armazenada em **tlc**. A variável **stop** é usada para interromper a simulação. Inicialmente, ela é configurada com **false**. O semáforo será executado até essa variável ser configurada com **true**. A variável **changed** é igual a **true** quando o sinal muda.

4. Em seguida, adicione o método **run()**, mostrado a seguir, que começa a execução do semáforo:

```
// Inicia o semáforo.
public void run() {
    while(!stop) {
        try {
            switch(tlc) {
                case GREEN:
                    Thread.sleep(10000); // verde por 10 segundos
                    break;
                case YELLOW:
                    Thread.sleep(2000); // amarelo por 2 segundos
                    break;
                case RED:
                    Thread.sleep(12000); // vermelho por 12 segundos
                    break;
            }
        } catch(InterruptedException exc) {
            System.out.println(exc);
        }
        changeColor();
    }
}
```

Esse método percorre o semáforo pelas cores. Primeiro, ele entra em suspensão durante um período apropriado, baseado na cor atual. Depois, chama **changeColor()** para mudar para a próxima cor da sequência.

5. Agora, adicione o método **changeColor()**, como mostrado aqui:

```
// Muda a cor.
synchronized void changeColor() {
    switch(tlc) {
        case RED:
            tlc = TrafficLightColor.GREEN;
            break;
        case YELLOW:
            tlc = TrafficLightColor.RED;
            break;
        case GREEN:
            tlc = TrafficLightColor.YELLOW;
    }

    changed = true;
    notify(); // sinaliza que a cor mudou
}
```

A instrução **switch** examina a cor armazenada atualmente em **tlc** e então atribui a próxima cor da sequência. Observe que esse método é sincronizado. Isso é necessário porque ele chama **notify()** para sinalizar que ocorreu uma mudança de cor. (Lembre-se de que **notify()** só pode ser chamado a partir de um contexto sincronizado.)

6. O próximo método é **waitForChange()**, que espera até a cor do sinal ser mudada.

```
// Espera até uma mudança de sinal ocorrer.
synchronized void waitForChange() {
    try {
        while(!changed)
            wait(); // espera o sinal mudar
        changed = false;
    } catch(InterruptedException exc) {
        System.out.println(exc);
    }
}
```

Esse método apenas chama **wait()**. A chamada não retornará até **changeColor()** executar uma chamada a **notify()**. Logo, **waitForChange()** não retornará até o sinal mudar.

7. Para concluir, adicione os métodos **getColor()**, que retorna a cor atual do sinal, e **cancel()**, que interrompe a thread do semáforo configurando **stop** com **true**. Esses métodos são mostrados abaixo:

```
// Retorna a cor atual.
synchronized TrafficLightColor getColor() {
    return tlc;
}

// Interrompe o semáforo.
synchronized void cancel() {
    stop = true;
}
```

8. Aqui está o código reunido em um programa completo que demonstra o semáforo:

```
// Tente isto 13-1

// Uma simulação de um semáforo que usa
// uma enumeração para descrever as cores das luzes.

// Enumeração com as cores de um semáforo.
enum TrafficLightColor {
    RED, GREEN, YELLOW
}

// Semáforo computadorizado.
class TrafficLightSimulator implements Runnable {
    private Thread thrd; // contém a thread que executa a simulação
    private TrafficLightColor tlc; // contém a cor atual
    boolean stop = false; // configura com true para interromper a
                          // simulação
    boolean changed = false; // true quando o sinal mudou

    TrafficLightSimulator(TrafficLightColor init) {
        tlc = init;
```

```
    thrd = new Thread(this);
    thrd.start();
}

TrafficLightSimulator() {
    tlc = TrafficLightColor.RED;

    thrd = new Thread(this);
    thrd.start();
}

// Inicia o semáforo.
public void run() {
    while(!stop) {
        try {
            switch(tlc) {
                case GREEN:
                    Thread.sleep(10000); // verde por 10 segundos
                    break;
                case YELLOW:
                    Thread.sleep(2000); // amarelo por 2 segundos
                    break;
                case RED:
                    Thread.sleep(12000); // vermelho por 12 segundos
                    break;
            }
        } catch(InterruptedException exc) {
            System.out.println(exc);
        }
        changeColor();
    }
}

// Muda a cor.
synchronized void changeColor() {
    switch(tlc) {
        case RED:
            tlc = TrafficLightColor.GREEN;
            break;
        case YELLOW:
            tlc = TrafficLightColor.RED;
            break;
        case GREEN:
            tlc = TrafficLightColor.YELLOW;
    }

    changed = true;
    notify(); // sinaliza que a cor mudou
}

// Espera até uma mudança de sinal ocorrer.
synchronized void waitForChange() {
    try {
        while(!changed)
            wait(); // espera o sinal mudar
    }
}
```

```

        changed = false;
    } catch(InterruptedException exc) {
        System.out.println(exc);
    }
}

// Retorna a cor atual.
synchronized TrafficLightColor getColor() {
    return tlc;
}

// Interrompe o semáforo.
synchronized void cancel() {
    stop = true;
}
}

class TrafficLightDemo {
    public static void main(String[] args) {
        TrafficLightSimulator tl =
            new TrafficLightSimulator(TrafficLightColor.GREEN);

        for(int i=0; i < 9; i++) {
            System.out.println(tl.getColor());
            tl.waitForChange();
        }

        tl.cancel();
    }
}

```

A saída a seguir é produzida. Como você pode ver, o semáforo percorre as cores na ordem verde, amarelo e vermelho:

```

GREEN
YELLOW
RED
GREEN
YELLOW
RED
GREEN
YELLOW
RED

```

Observe como o uso da enumeração no programa simplifica e adiciona estrutura ao código que precisa saber o estado do semáforo. Como o sinal só pode ter três estados (vermelho, verde ou amarelo), o uso de uma enumeração assegura que só esses valores sejam válidos, impedindo assim um mau uso acidental.

9. Podemos melhorar o programa anterior beneficiando-nos dos recursos de classe de uma enumeração. Por exemplo, adicionando um construtor, uma variável de instância e um método a **TrafficLightSimulator**, podemos melhorar significativamente o programa. Essa melhoria será deixada como exercício. Consulte o Exercício 4.

AUTOBOXING

Java tem dois recursos muito úteis: *autoboxing* e *autounboxing*. Eles não faziam parte da especificação original, mas foram adicionados pelo JDK 5. O *autoboxing/unboxing* simplifica e otimiza bastante códigos que têm de converter tipos primitivos em objetos e vice-versa. Já que essas situações são encontradas com frequência em código Java, os benefícios do *autoboxing/unboxing* afetam quase todos os programadores de Java. Como você verá no Capítulo 14, esses recursos também trazem grandes contribuições à usabilidade dos genéricos.

O *autoboxing/unboxing* está diretamente relacionado aos encapsuladores de tipos Java e à maneira como os valores são movidos para dentro e para fora da instância de um encapsulador. Portanto, começaremos com uma visão geral dos encapsuladores de tipos e do processo de empacotar e desempacotar valores manualmente.

Encapsuladores de tipos

Como você sabe, Java usa tipos primitivos, como **int** ou **double**, para armazenar os tipos de dados básicos suportados pela linguagem. Tipos primitivos, em vez de objetos, são usados para representar esses valores por questões de desempenho. O uso de objetos para esses tipos básicos adicionaria uma sobrecarga inaceitável até mesmo ao cálculo mais simples. Logo, os tipos primitivos não fazem parte da hierarquia de objetos e não herdam **Object**.

Apesar dos benefícios oferecidos ao desempenho pelos tipos primitivos, podemos precisar de uma representação na forma de objeto. Por exemplo, não podemos passar um tipo primitivo por referência para um método. Além disso, muitas das estruturas de dados padrão implementadas por Java operam em objetos, ou seja, não podemos usar essas estruturas de dados para armazenar tipos primitivos. Para tratar essas (e outras) situações, Java fornece *encapsuladores de tipos*, que são classes que encapsulam um tipo primitivo dentro de um objeto. As classes encapsuladoras de tipos foram introduzidas brevemente no Capítulo 11. Aqui, serão examinadas com mais detalhes.

Os encapsuladores de tipos são **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character** e **Boolean**, que ficam no pacote **java.lang**. Essas classes oferecem um amplo conjunto de métodos que nos permite integrar totalmente os tipos primitivos à hierarquia de objetos Java.

Provavelmente os encapsuladores de tipos mais usados sejam os que representam valores numéricos. Eles são **Byte**, **Short**, **Integer**, **Long**, **Float** e **Double**. Todos os encapsuladores de tipos numéricos herdam a classe abstrata **Number**. **Number** declara métodos que retornam o valor de um objeto em cada um dos tipos numéricos diferentes. Esses métodos são mostrados aqui:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

Por exemplo, **doubleValue()** retorna o valor de um objeto na forma de um **double**, **floatValue()** retorna o valor como um **float**, e assim por diante. Esses métodos são implementados por todos os encapsuladores de tipos numéricos.

Cada um dos encapsuladores de tipos numéricos define construtores que permitem que um objeto seja construído a partir de um valor dado, ou a partir da representação desse valor na forma de string. Por exemplo, estes são os construtores definidos para **Integer** e **Double**:

```
Integer(int num)
Integer(String str) throws NumberFormatException
Double(double num)
Double(String str) throws NumberFormatException
```

Se *str* não tiver um valor numérico válido, uma **NumberFormatException** será lançada.

Todos os encapsuladores de tipos sobreponem o método **toString()**. Ele retorna a forma legível por humanos do valor contido dentro do encapsulador. Isso nos permite exibir o valor passando um objeto encapsulador de tipo para **println()**, por exemplo, sem precisar convertê-lo em seu tipo primitivo.

O processo de encapsular um valor dentro de um objeto se chama *boxing*. Antes do JDK 5, o boxing era feito manualmente, com o programador construindo de maneira explícita a instância de um encapsulador com o valor desejado. Por exemplo, a linha seguinte encapsula manualmente o valor 100 em um **Integer**:

```
| Integer iOb = new Integer(100);
```

Nesse exemplo, um novo objeto **Integer** com o valor 100 é criado explicitamente e uma referência a ele é atribuída a **iOb**.

O processo de extrair o valor de um encapsulador de tipo se chama *unboxing*. Novamente, antes do JDK 5, o unboxing também ocorria manualmente, com o programador chamando de maneira explícita um método no encapsulador para obter seu valor. Por exemplo, a linha seguinte extraí manualmente o valor de **iOb** para um **int**:

```
| int i = iOb.intValue();
```

Aqui, **intValue()** retorna o valor encapsulado dentro de **iOb** como um **int**.

O programa a seguir demonstra os conceitos anteriores:

```
// Demonstra o boxing e o unboxing manualmente com um encapsulador de tipo.
class Wrap {
    public static void main(String[] args) {
        Integer iOb = new Integer(100); ← Encapsula manualmente o valor 100.

        int i = iOb.intValue(); ← Extrai manualmente o valor de iOb.

        System.out.println(i + " " + iOb); // exibe 100 100
    }
}
```

Esse programa encapsula o valor inteiro 100 dentro de um objeto **Integer** chamado **iOb**. Em seguida, obtém esse valor chamando **intValue()** e armazena o resultado em **i**. Para concluir, exibe os valores de **i** e **iOb**, ambos iguais a 100.

O mesmo procedimento geral usado pelo exemplo anterior no boxing e unboxing manual de valores era requerido por todas as versões de Java anteriores ao JDK 5 e ainda é amplamente usado em código legado. O problema é que ele é tedioso e propenso a erros, porque exige que o programador crie manualmente o objeto apropriado ao encapsulamento de um valor e obtenha explicitamente o tipo primitivo apropriado quando seu valor é necessário. Felizmente, o *autoboxing/unboxing* melhora muito esses procedimentos essenciais.

Fundamentos do autoboxing

Autoboxing é o processo pelo qual um tipo primitivo é encapsulado (embalado) automaticamente no encapsulador de tipo equivalente sempre que um objeto desse tipo é necessário. Não há necessidade de construir explicitamente um objeto. Autounboxing é o processo pelo qual o valor de um objeto embalado é extraído (desembalado) automaticamente de um encapsulador de tipo quando seu valor é necessário. Não há necessidade de chamar um método como **intValue()** ou **doubleValue()**.

A inclusão do *autoboxing* e do *autounboxing* otimiza bastante a codificação de vários algoritmos, removendo o tédio de encapsular e extrair valores manualmente. Também ajuda a evitar erros. Com o *autoboxing*, não é necessário construir manualmente um objeto para encapsular um tipo primitivo. Você só tem de atribuir esse valor a uma referência do encapsulador de tipo. Java constrói automaticamente o objeto. Por exemplo, esta é a maneira moderna de construir um objeto **Integer** com o valor 100:

```
| Integer i0b = 100; // faz o autobox de um int
```

Observe que o objeto não é criado explicitamente com o uso de **new**. Java trata isso para você, automaticamente.

Para fazer o unbox de um objeto, apenas atribua a referência desse objeto a uma variável de tipo primitivo. Por exemplo, para fazer o unbox de **iOb**, você pode usar a linha seguinte:

```
| int i = iOb; // autounbox
```

Java trata os detalhes.

O programa a seguir demonstra as instruções anteriores:

```
// Demonstra o autoboxing/unboxing.
class AutoBox {
    public static void main(String[] args) {
        Integer iOb = 100; // faz o autobox de um int ←
        int i = iOb; // autounbox ←
        System.out.println(i + " " + iOb); // exibe 100 100
    }
}
```

Faz o autobox e depois o autounbox do valor 100.

Verificação do progresso

1. Qual é o encapsulador do tipo **double**?
2. O que acontece quando encapsulamos um valor primitivo?
3. Autoboxing é o recurso que encapsula automaticamente um valor primitivo em um objeto do encapsulador de tipo correspondente. Verdadeiro ou falso?

Autoboxing e os métodos

Além do simples caso de atribuições, o *autoboxing* ocorre automaticamente sempre que um tipo primitivo deve ser convertido em um objeto, e o *autounboxing* ocorre sempre que um objeto deve ser convertido em um tipo primitivo. Logo, o *autoboxing/unboxing* pode ocorrer quando um argumento é passado para um método ou quando um valor é retornado por um método. Por exemplo, considere o seguinte:

```
// O autoboxing/unboxing ocorre com parâmetros
// e valores de retorno de métodos.

class AutoBox2 {
    // Esse método tem um parâmetro Integer.
    static void m(Integer v) { ← Recebe um Integer.
        System.out.println("m() received " + v);
    }

    // Esse método retorna um int.
    static int m2() { ← Retorna um int.
        return 10;
    }

    // Esse método retorna um Integer.
    static Integer m3() { ← Retorna um Integer.
        return 99; // faz o autoboxing de 99 para um Integer.
    }

    public static void main(String[] args) {
        // Passa um int para m(). Já que m() tem um parâmetro Integer,
        // o valor int passado é encapsulado automaticamente.
        m(199);
    }
}
```

Respostas:

1. **Double**
2. Quando um tipo primitivo é encapsulado, seu valor é inserido dentro de um objeto do encapsulador de tipo correspondente.
3. Verdadeiro.

```
// Aqui, iOb recebe o valor int retornado por m2().  
// Esse valor é encapsulado automaticamente para  
// poder ser atribuído a iOb.  
Integer iOb = m2();  
System.out.println("Return value from m2() is " + iOb);  
  
// Em seguida, m3() é chamado. Ele retorna um valor Integer  
// que é encapsulado automaticamente em um int.  
int i = m3();  
System.out.println("Return value from m3() is " + i);  
  
// Agora, Math.sqrt() é chamado com iOb como argumento.  
// Nesse caso, iOb sofre autounboxing e seu valor é promovido  
// a double, que é o tipo que sqrt() precisa.  
iOb = 100;  
System.out.println("Square root of iOb is " + Math.sqrt(iOb));  
}  
}
```

Esse programa exibe o resultado a seguir:

```
m() received 199  
Return value from m2() is 10  
Return value from m3() is 99  
Square root of iOb is 10.0
```

No programa, observe que **m()** especifica um parâmetro **Integer**. Dentro de **main()**, **m()** recebe o valor **int** 199. Como **m()** está esperando um **Integer**, esse valor sofre boxing automático. Em seguida, **m2()** é chamado. Ele retorna o valor **int** 10. Esse valor **int** é atribuído a **iOb** em **main()**. Como **iOb** é um **Integer**, o valor retornado por **m2()** sofre *autoboxing*. Agora, **m3()** é chamado. Ele retorna um **Integer** que é extraído automaticamente para um **int**. Para concluir, **Math.sqrt()** é chamado com **iOb** como argumento. Nesse caso, **iOb** sofre *autounboxing* e seu valor é promovido a **double**, já que esse é o tipo esperado por **Math.sqrt()**.

Autoboxing/unboxing ocorre em expressões

Em geral, o *autoboxing* e o unboxing ocorrem sempre que uma conversão para um objeto ou a partir de um objeto é necessária; isso se aplica a expressões. Dentro de uma expressão, um objeto numérico sofre unboxing automático. O resultado da expressão é encapsulado novamente, se preciso. Por exemplo, considere o programa a seguir:

```
// Autoboxing/unboxing ocorre dentro de expressões.  
  
class AutoBox3 {  
    public static void main(String[] args) {  
        Integer iOb, iOb2;  
        int i;  
  
        iOb = 99;  
        System.out.println("Original value of iOb: " + iOb);
```

```

// O trecho a seguir faz o unboxing automático de iOb,
// executa o incremento e encapsula
// o resultado novamente em iOb.
++iOb; ←
System.out.println("After ++iOb: " + iOb);

// Aqui, iOb sofre unboxing, seu valor é aumentado em 10
// e o resultado é encapsulado e armazenado novamente em iOb.
iOb += 10; ←
System.out.println("After iOb += 10: " + iOb);

// Agora, iOb sofre unboxing, a expressão é avaliada
// e o resultado é encapsulado novamente
// e atribuído a iOb2.
iOb2 = iOb + (iOb / 3); ←
System.out.println("iOb2 after expression: " + iOb2);

// A mesma expressão é avaliada,
// mas o resultado não é encapsulado.
i = iOb + (iOb / 3); ←
System.out.println("i after expression: " + i);
}
}

```

Autoboxing/
unboxing
ocorre em
expressões.

A saída é mostrada abaixo:

```

Original value of iOb: 99
After ++iOb: 100
After iOb += 10: 110
iOb2 after expression: 146
i after expression: 146

```

Preste atenção nesta linha:

```
| ++iOb;
```

Ela faz o valor de **iOb** ser incrementado. Funciona do seguinte modo: **iOb** sofre unboxing, o valor é incrementado e o resultado é encapsulado novamente.

Graças ao *autounboxing*, você pode usar objetos numéricos inteiros, como um **Integer**, para controlar uma instrução **switch**. Por exemplo, considere o fragmento a seguir:

```

Integer iOb = 2;

switch(iOb) {
    case 1: System.out.println("one");
    break;
    case 2: System.out.println("two");
    break;
    default: System.out.println("error");
}

```

Quando a expressão **switch** é avaliada, **iOb** sofre unboxing e seu valor **int** é obtido.

Como os exemplos do programa mostram, graças ao *autoboxing/unboxing*, é intuitivo e fácil usar objetos numéricos em uma expressão. No passado, um código assim teria envolvido coerções e chamadas a métodos como **intValue()**.

Advertência

Uma vez que temos o *autoboxing* e o *autounboxing*, alguém poderia ficar tentado a usar apenas objetos como **Integer** ou **Double**, abandonando totalmente os tipos primitivos. Por exemplo, com o *autoboxing/unboxing* podemos escrever um código como este:

```
// Uso inadequado do autoboxing/unboxing!
Double a, b, c;

a = 10.2;
b = 11.4;
c = 9.8;

Double avg = (a + b + c) / 3;
```

Nesse exemplo, objetos de tipo **Double** contêm valores cuja média é calculada e o resultado atribuído a outro objeto **Double**. Embora esse código esteja tecnicamente correto e, na verdade, funcione de maneira apropriada, é uma aplicação bastante inadequada do *autoboxing/unboxing*. É muito menos eficiente do que um código equivalente escrito com o uso do tipo primitivo **double**. Isso ocorre porque cada operação de *autoboxing* e *autounboxing* adiciona uma sobrecarga que não existe quando o tipo primitivo é usado.

Em geral, devemos restringir o uso de encapsuladores de tipos apenas aos casos em que a representação de um tipo primitivo na forma de objeto seja requerida. O *autoboxing/unboxing* não foi adicionado a Java como uma maneira “sorrateira” de eliminar os tipos primitivos.

Verificação do progresso

1. Um valor primitivo é encapsulado automaticamente quando é passado como argumento para um método que está esperando um objeto encapsulador de tipo?
2. Devido aos limites impostos pelo sistema de tempo de execução Java, o *autoboxing/unboxing* não ocorre em objetos usados em expressões. Verdadeiro ou falso?
3. Graças ao *autoboxing/unboxing*, devemos usar objetos em vez de tipos primitivos para executar a maioria das operações aritméticas. Verdadeiro ou falso?

Respostas:

1. Sim.
2. Falso.
3. Falso.

ANOTAÇÕES (METADADOS)

Outro recurso adicionado a Java pelo JDK 5 é a *anotação*. Ele nos permite embutir informações complementares (uma anotação) em um arquivo-fonte. Por exemplo, podemos anotar um método usando informações sobre o status de sua versão. Essas informações não alteram as ações de um programa. No entanto, podem ser usadas por várias ferramentas durante o desenvolvimento e a implantação. Por exemplo, uma anotação pode ser processada por um gerador de código-fonte, pelo compilador ou por uma ferramenta de implantação. O termo *metadados* também é usado para fazer referência a esse recurso, mas o termo *anotação* é o mais descritivo e normalmente mais usado.

Embora quase sempre usemos anotações predefinidas em vez de definir anotações personalizadas, é útil conhecer sua sintaxe e conceitos básicos, portanto, começaremos com uma visão geral da criação e uso de uma anotação.

Criando e usando uma anotação

Uma anotação é criada com um mecanismo baseado na **interface**. Aqui está um exemplo simples:

```
// Um tipo de anotação simples.
@interface MyAnno {
    String str();
    int val();
}
```

Esse exemplo declara uma anotação chamada **MyAnno**. Observe o símbolo **@** que precede a palavra-chave **interface**. Ele informa ao compilador que um tipo de anotação está sendo declarado. Em seguida, observe os dois membros **str()** e **val()**. Todas as anotações são compostas somente por declarações de métodos. No entanto, não fornecemos corpos para esses métodos. Em vez disso, Java implementa os métodos. Além do mais, os métodos agem como campos.

Todos os tipos de anotações estendem automaticamente a interface **Annotation**. Logo, **Annotation** é uma superinterface de todas as anotações; ela é declarada dentro do pacote **java.lang.annotation**.

Uma vez que você tiver declarado uma anotação, poderá usá-la para comentar uma declaração. Qualquer tipo de declaração pode ter uma anotação associada. Por exemplo, classes, métodos, campos, parâmetros e constantes **enum** podem ter anotações. Até mesmo a anotação pode ter uma anotação. Seja qual for o caso, a anotação precede o resto da declaração.

Quando aplicamos uma anotação, fornecemos valores aos seus membros. Por exemplo, aqui está um exemplo de **MyAnno** sendo aplicada a um método:

```
// Anotação de um método.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() { // ... }
```

Essa anotação está vinculada ao método **myMeth()**. Observe com atenção sua sintaxe. O nome da anotação, precedido por um **@**, é seguido por uma lista entre parênteses com inicializações de membros. Para um membro receber um valor, ele é atribuído ao seu nome. Logo, no exemplo, o string “Annotation Example” é atribuído ao membro **str** de **MyAnno**. Observe que não há parênteses após **str** nessa atribuição.

Quando o membro de uma anotação recebe um valor, só seu nome é usado. Portanto, os membros da anotação parecem campos nesse contexto.

Você pode dar ao membro de uma anotação um valor padrão que será usado se nenhum valor for especificado quando a anotação for aplicada. Um valor padrão é indicado pela inclusão de uma cláusula **default** na declaração do membro. Ela tem esta forma geral:

tipo membro() default valor;

Aqui, *valor* deve ser de um tipo compatível com *tipo*. Por exemplo, esta é **MyAnno**, com **val()** recebendo um valor padrão igual a 42:

```
// Dá a val() um valor padrão
@interface MyAnno {
    String str();
    int val() default 42;
}
```

Agora, você poderia usar **MyAnno** desta forma:

```
@MyAnno(str = "Annotation Example")
```

Nesse caso, o valor de **val()** tem como padrão 42, mas é claro que você também pode dar um valor diferente, como antes. Por exemplo,

```
@MyAnno(str = "Annotation Example", val = 100)
```

também é válido.

Se você tiver uma anotação com um único membro chamado **value**, poderá usar uma forma “abreviada” para especificar seu valor. É só passar o valor para esse membro quando a anotação for aplicada – sem ser preciso especificar o nome **value**. Por exemplo, dado o código

```
@interface MySingle {
    int value();
}
```

você pode dar a **value()** o valor 100 quando a anotação for aplicada, desta forma:

```
@MySingle(100)
```

Observe que **value =** não precisou ser especificado.

Um último ponto: as anotações que não têm parâmetros são chamadas de *anotações marcadoras*. Elas são especificadas sem a passagem de nenhum argumento e sem o uso de parênteses. Sua única finalidade é a de marcar uma declaração com algum atributo.

Anotações internas

Java define muitas anotações internas. A maioria é especializada, mas oito são de uso geral. Quatro são importadas de **java.lang.annotation**: **@Retention**, **@Documented**, **@Target** e **@Inherited**. Quatro, **@Override**, **@Deprecated**, **@SafeVarargs** e **@SuppressWarnings**, estão incluídas em **java.lang**. Elas são mostradas na Tabela 13-1.

Aqui está um exemplo que usa **@Deprecated** para marcar a classe **MyClass** e o método **getMsg()** como substituídos. (O termo *substituído* significa obsoleto e não apropriado para uso em códigos novos.) Quando você tentar compilar esse programa, avisos relatarão o uso dos elementos substituídos.

Tabela 13-1 Anotações internas de uso geral

Anotação	Descrição
@Retention	Especifica a política de retenção associada à anotação. A política de retenção determina quanto tempo uma anotação estará presente durante o processo de compilação e implantação.
@Documented	Anotação marcadora que informa a uma ferramenta que uma anotação deve ser documentada. Foi projetada para ser usada apenas como anotação de uma declaração de anotação.
@Target	Especifica os tipos de declarações aos quais uma anotação pode ser aplicada. Foi projetada para ser usada apenas como anotação de outra anotação. @Target recebe um argumento, que deve ser uma constante da enumeração ElementType , que define várias constantes, como CONSTRUCTOR , FIELD e METHOD . O argumento determina os tipos de declarações aos quais a anotação pode ser aplicada.
@Inherited	Anotação marcadora que faz a anotação de uma superclasse ser herdada por uma subclasse.
@Override	Método com a anotação @Override deve sobrepor o método de uma superclasse. Se não o fizer, isso resultará em um erro de tempo de compilação. É usada para garantir que um método da superclasse seja realmente sobreposto e não apenas sobrecarregado. É uma anotação marcadora.
@Deprecated	Anotação marcadora que indica que um recurso está obsoleto e foi substituído por uma forma mais nova.
@SafeVarargs	Anotação marcadora que indica que não ocorrerá uma ação insegura relacionada a um parâmetro varargs de um método ou construtor. Só pode ser aplicada a construtores ou métodos estáticos ou finais. (Adicionada pelo JDK 7.)
@SupressWarnings	Especifica que um ou mais avisos que podem ser emitidos pelo compilador devem ser suprimidos. Os avisos a serem suprimidos são especificados por nome, na forma de string.

```
// Exemplo que usa @Deprecated.

// Substitui uma classe.
@Deprecated ←———— Marca uma classe como substituída.
class MyClass {
    private String msg;

    MyClass(String m) {
        msg = m;
    }

    // Substitui o método de uma classe.
    @Deprecated ←———— Marca um método como substituído.
    String getMsg() {
        return msg;
    }
}
```

```
// ...
}

class AnnoDemo {
    public static void main(String[] args) {
        MyClass myObj = new MyClass("test");

        System.out.println(myObj.getMsg());
    }
}
```

EXERCÍCIOS

1. Diz-se que as constantes de enumeração são *autotipadas*. O que isso significa?
2. Que classe todas as enumerações herdam automaticamente?
3. Dada a enumeração a seguir, escreva um programa que use **values()** para exibir uma lista das constantes e seus valores ordinais.

```
enum Tools {
    SCREWDRIVER, WRENCH, HAMMER, PLIERS
}
```

4. A simulação do semáforo desenvolvida na seção Tente isto 13-1 pode ser melhorada com algumas alterações simples que se beneficiem dos recursos de classe da enumeração. Na versão mostrada, a duração de cada sinal era controlada pela classe **TrafficLightSimulator** com os valores sendo embutidos no método **run()**. Altere isso para que a duração de cada sinal seja armazenada pelas constantes da enumeração **TrafficLightColor**. Para fazê-lo, você deverá adicionar um construtor, uma variável de instância privada e um método chamado **getDelay()**.
5. Defina boxing e unboxing. Como o *autoboxing/unboxing* afeta essas ações?
6. Altere o fragmento a seguir para que use o *autoboxing*.

```
| Short val = new Short(123);
```

7. Uma anotação é sintaticamente baseada em uma _____.
8. O que é uma anotação marcadora?
9. Uma anotação só pode ser aplicada a métodos. Verdadeiro ou falso?
10. Dada uma enumeração **MyEnum**, qual seria uma maneira fácil de descobrir quantos valores existem nela?
11. Reimplemente o método **changeColor()** da classe **TrafficLightSimulator** que vimos na seção Tente isto 13-1 para que, em vez de uma instrução **switch**, use **values()** e **ordinal()** para determinar a próxima cor a ser atribuída a **tlc**.
12. Considere a classe **Counter** a seguir:

```
class Counter {
    boolean up;
```

```

int count;

Counter(boolean b, int c) {
    up = b;
    count = c;
}

public int count() {
    if(up)
        return count++;
    else
        return count--;
}

// Código de teste
public static void main(String[] args) {
    Counter c1 = new Counter(true, 10);
    Counter c2 = new Counter(false, 10);

    for(int i = 0; i < 10; i++)
        System.out.println(c1.count() + ", " + c2.count());
}
}

```

Como você pode ver, um objeto **Counter** conta em ordem crescente ou decrescente, dependendo do argumento **boolean** passado para o construtor. Uma implementação alternativa não usaria um parâmetro **boolean**; usaria, em vez disso, uma enumeração **Direction** com dois valores: **UP** e **DOWN**. Reimplemente **Counter** dessa forma. Cite uma vantagem de uma implementação sobre a outra.

13. O programa a seguir usa constantes inteiras nomeadas. Reescreva-o para que use uma enumeração. Ele deve continuar tendo a mesma entrada e saída.

```

import java.io.*;

class Castle {
    public static final int NORTH = 0;
    public static final int SOUTH = 1;
    public static final int EAST = 2;
    public static final int WEST = 3;

    public static void main(String[] args) throws IOException {
        int direction;

        System.out.println("From which direction is the enemy attacking?");
        System.out.println(" 0 = North, 1 = South, 2 = East, 3 = West");
        direction = System.in.read() - '0';

        // agora retorne a resposta
        System.out.print("The attack is from the following direction: ");
        switch(direction) {

```

```
        case NORTH:
            System.out.println("NORTH");
            break;
        case SOUTH:
            System.out.println("SOUTH");
            break;
        case EAST:
            System.out.println("EAST");
            break;
        case WEST:
            System.out.println("WEST");
    }
}
```

14. Crie uma enumeração **DayOfWeek** com sete valores entre **SUNDAY** e **SATURDAY**. Adicione um método **isWorkDay()** à classe **DayOfWeek** que retorne **true** se o valor em que for chamado estiver entre **MONDAY** e **FRIDAY**. Por exemplo, a chamada **DayOfWeek.SUNDAY.isWorkDay()** retorna **false**.
15. **Byte**, **Short**, **Integer**, **Long**, **Float** e **Double** herdam todos os métodos da classe **Number**, já que são subclasses dela. Logo, **Double**, por exemplo, não tem apenas um método **doubleValue**, também tem os métodos **byteValue()**, **shortValue()**, **intValue()**, **longValue()** e **floatValue()**. O que será exibido pelo segmento de código a seguir? Se algum dos valores for incomum, explique-o.

```
public class NumberTester {
    public static void main(String[] args) {
        Double d = new Double(123456.789);

        System.out.println(d.byteValue());
        System.out.println(d.shortValue());
        System.out.println(d.intValue());
        System.out.println(d.longValue());
        System.out.println(d.floatValue());
        System.out.println(d.doubleValue());
    }
}
```

16. No código abaixo, as variáveis **a** e **b** são inicializadas para referenciar o mesmo objeto **Integer** com valor 3. Em seguida, o objeto **Integer** referenciado por **b** é incrementado para 4. Para concluir, os valores de **a** e **b** são exibidos e vemos “3 4”. Por que não é exibido o mesmo valor tanto para **a** quanto para **b**? As variáveis **a** e **b** não deveriam ter o mesmo valor, já que referenciam o mesmo objeto? O que está ocorrendo aqui?

```
Integer a = 3;
Integer b = a;
b++;
System.out.println(a + " " + b); // exibe "3 4"
```

17. Quais das instruções a seguir são válidas?
- A. Object o = 3;
 - B. Number n = 3;
 - C. Float f = (float) 3;
 - D. Integer i = (Integer) 3;
 - E. Integer j = o + I; // usa as declarações de variáveis anteriores
18. Considere as duas declarações de classes abaixo. Elas serão compiladas sem erro? Explique.

```
class MySuper {  
    void myHello(String s) {  
        System.out.println(s);  
    }  
}  
  
class MySub extends MySuper {  
    @Override  
    void myHello(int x) {  
        System.out.println(x);  
    }  
}
```

14

Tipos genéricos

PRINCIPAIS HABILIDADES E CONCEITOS

- Entender os benefícios dos genéricos
- Criar uma classe genérica
- Aplicar parâmetros de tipo limitado
- Usar argumentos curingas
- Aplicar curingas limitados
- Criar um método genérico
- Criar um construtor genérico
- Criar uma hierarquia genérica
- Criar uma interface genérica
- Utilizar tipos brutos
- Aplicar a inferência de tipos com o operador losango
- Entender a técnica *erasure*
- Evitar erros de ambiguidade
- Conhecer as restrições dos genéricos

Desde sua versão original, muitos recursos novos foram adicionados a Java. Todos melhoraram e expandiram seu escopo, mas talvez o que teve impacto mais profundo seja o *tipo genérico*, porque seus efeitos foram sentidos em toda a linguagem. Introduzidos pelo JDK 5, os genéricos adicionaram um elemento de sintaxe totalmente novo e causaram mudanças em muitas das classes e métodos da API principal. Não é exagero dizer que sua inclusão basicamente reformulou a natureza de Java.

Atualmente, os genéricos são parte integrante da programação Java e todos os programadores da linguagem precisam ter um conhecimento sólido do assunto. Além disso, o conceito de tipos genéricos já integra a base da programação moderna em geral. Competência no uso dos genéricos também é requerida para usarmos com eficácia o Collections Framework. Os genéricos são não só um dos tópicos mais sofisticados de Java, como também um dos mais importantes.

FUNDAMENTOS DOS TIPOS GENÉRICOS

Na verdade, com o termo *genéricos* queremos nos referir aos *tipos parametrizados*. Os tipos parametrizados são importantes porque nos permitem criar classes, interfaces e métodos em que o tipo de dado com o qual operam é especificado como parâmetro. Uma classe, interface ou método que usa um parâmetro de tipo é chamado de *genérico*, como em *classe genérica* ou *método genérico*.

Uma vantagem importante do código genérico é que ele funciona automaticamente com o tipo de dado passado para seu parâmetro de tipo. Muitos algoritmos são logicamente iguais, não importando o tipo de dado ao qual estão sendo aplicados. Por exemplo, o mecanismo que dá suporte a uma pilha é o mesmo sem importar se ela está armazenando itens de tipo **Integer**, **String**, **Object** ou **Thread**. Com os genéricos, você pode definir um algoritmo uma única vez, independentemente do tipo de dado, e então aplicá-lo a uma ampla variedade de tipos de dados sem nenhum esforço adicional.

É importante entender que Java sempre permitiu a criação de classes, interfaces e métodos generalizados usando referências de tipo **Object**. Como **Object** é a superclasse de todas as outras classes, uma referência **Object** pode referenciar qualquer tipo de objeto. Logo, em códigos anteriores aos genéricos, classes, interfaces e métodos generalizados usavam referências **Object** para operar com vários tipos de dados. O problema é que eles não faziam isso com *segurança de tipos*, já que coerções eram necessárias para converter explicitamente **Object** no tipo de dado que estava sendo tratado. Portanto, era possível gerar accidentalmente discrepâncias de tipo. Os genéricos adicionam a segurança de tipos que estava faltando, porque tornam essas coerções automáticas e implícitas. Resumindo, eles expandem nossa habilidade de reutilizar código e nos permitem fazê-lo de maneira segura e confiável.

Exemplo simples de genérico

Antes de discutir mais teoria, é melhor examinarmos um exemplo simples de genérico. O programa a seguir define duas classes: a primeira é a classe genérica **Gen** e a segunda é **GenDemo**, que usa **Gen**.

```
// Classe genérica simples.
// Aqui, T é um parâmetro de tipo que será substituído pelo
// tipo real quando um objeto de tipo Gen for criado.
class Gen<T> { ←
    T ob; // declara uma referência a um objeto de tipo T
    // Passa para o construtor uma referência
    // a um objeto de tipo T.
    Gen(T o) {
        ob = o;
    }

    // Retorna ob.
    T getob() {
        return ob;
    }
}
```

Declara uma classe genérica. T é o parâmetro de tipo genérico.

```

// Exibe o tipo de T.
void showType() {
    System.out.println("Type of T is " +
                       ob.getClass().getName());
}

// Demonstra a classe genérica.
class GenDemo {
    public static void main(String[] args) {
        // Cria uma referência Gen para Integers.
        Gen<Integer> iOb; ← Cria uma referência
                            a um objeto de tipo
                            Gen<Integer>.

        // Cria um objeto Gen<Integer> e atribui sua
        // referência a iOb. Observe o uso do autoboxing
        // no encapsulamento do valor 88 dentro de um objeto Integer.
        iOb = new Gen<Integer>(88); ← Instancia um
                                       objeto de tipo
                                       Gen<Integer>.

        // Exibe o tipo de dado usado por iOb.
        iOb.showType();

        // Obtém o valor de iOb. Observe que
        // nenhuma coerção é necessária.
        int v = iOb.getob();
        System.out.println("value: " + v);

        System.out.println(); ← Cria uma referência e um
                            objeto de tipo Gen<String>.

        // Cria um objeto Gen para Strings.
        Gen<String> strOb = new Gen<String>("Generics Test"); ←

        // Exibe o tipo de dado usado por strOb.
        strOb.showType();

        // Obtém o valor de strOb. Novamente, observe
        // que nenhuma coerção é necessária.
        String str = strOb.getob();
        System.out.println("value: " + str);
    }
}

```

A saída produzida pelo programa é mostrada abaixo:

```
Type of T is java.lang.Integer
```

```
value: 88
```

```
Type of T is java.lang.String
```

```
value: Generics Test
```

Examinemos esse programa em detalhes. Primeiro, observe como **Gen** é declarada pela linha a seguir:

```
| class Gen<T> {
```

Aqui, **T** é o nome de um *parâmetro de tipo*. Esse nome é usado como espaço reservado para o tipo real que será passado para **Gen** quando um objeto for criado. Logo, **T** será usado dentro de **Gen** sempre que o parâmetro de tipo for necessário. Observe que **T** está dentro de <>. Essa sintaxe pode ser generalizada. Sempre que um parâmetro de tipo estiver sendo declarado, ele será especificado dentro de colchetes angulares. Já que **Gen** usa um parâmetro de tipo, é uma classe genérica.

Na declaração de **Gen**, não há um significado especial no nome **T**. Qualquer identificador válido poderia ter sido usado, mas o uso de **T** é tradicional. Com frequência os nomes dos parâmetros de tipo são compostos apenas por um caractere maiúsculo. Outros nomes de parâmetros de tipo normalmente usados são **V** e **E**.

Em seguida, **T** é usado para declarar um objeto chamado **ob**, como mostrado abaixo:

```
| T ob; // declara um objeto de tipo T
```

Como explicado, **T** é um espaço reservado para o tipo real que será especificado quando um objeto **Gen** for criado. Logo, **ob** será um objeto do tipo passado para **T**. Por exemplo, se o tipo **String** for passado para **T**, então, nesse caso, **ob** será de tipo **String**.

Agora, considere o construtor de **Gen**:

```
| Gen(T o) {
    ob = o;
}
```

Observe que seu parâmetro, **o**, é de tipo **T**. Ou seja, o tipo real de **o** será determinado pelo tipo passado para **T** quando um objeto **Gen** for criado. Além disso, já que tanto o parâmetro **o** quanto a variável membro **ob** são de tipo **T**, ambos terão o mesmo tipo quando um objeto **Gen** for criado.

O parâmetro de tipo **T** também pode ser usado para especificar o tipo de retorno de um método, como ocorre com o método **getob()**, mostrado aqui:

```
| T getob() {
    return ob;
}
```

Já que **ob** também é de tipo **T**, seu tipo é compatível com o tipo de retorno especificado por **getob()**.

O método **showType()** exibe o tipo de **T**. Ele faz isso chamando **getName()** no objeto **Class** retornado pela chamada a **getClass()** em **ob**. Não usamos esse recurso antes, logo, vamos examiná-lo em detalhes. Você deve lembrar que, no Capítulo 7, vimos que a classe **Object** define o método **getClass()**. Portanto, **getClass()** é membro de todos os tipos de classe. Ele retorna um objeto **Class** correspondente ao tipo de classe do objeto em que foi chamado. **Class** é uma classe definida dentro de **java.lang** que encapsula informações sobre outra classe. Ela define vários métodos que podem ser usados na

obtenção de informações sobre uma classe no tempo de execução. Entre eles, está o método `getName()`, que retorna uma representação do nome da classe na forma de string.

A classe `GenDemo` demonstra a classe genérica `Gen`. Primeiro, ela cria uma versão de `Gen` para inteiros, como vemos abaixo:

```
| Gen<Integer> iOb;
```

Examine bem essa declaração. Primeiro, observe que o tipo `Integer` é especificado dentro de colchetes angulares após `Gen`. Nesse caso, `Integer` é um *argumento de tipo* que é passado para o parâmetro de tipo de `Gen`, que é `T`. Isso cria uma versão de `Gen` em que todas as referências a `T` são convertidas para referências a `Integer`. Logo, para essa declaração, `ob` é de tipo `Integer`, e o tipo de retorno de `getob()` também.

Antes de prosseguirmos, é preciso dizer que o compilador Java não cria realmente versões diferentes de `Gen` ou de qualquer outra classe genérica. Embora seja útil pensar assim, não é o que acontece. Em vez disso, o compilador remove todas as informações do tipo genérico, substituindo pelas coerções necessárias, para fazer o código *se comportar como se* uma versão específica de `Gen` fosse criada. Portanto, na verdade, há apenas uma versão de `Gen` no programa. O processo de remover informações do tipo genérico se chama *erasure*; ele será discutido posteriormente neste capítulo.

A próxima linha atribui a `iOb` uma referência a uma instância de uma versão `Integer` da classe `Gen`.

```
| iOb = new Gen<Integer>(88);
```

Observe que, quando o construtor de `Gen` é chamado, o argumento de tipo `Integer` também é especificado. Isso é necessário porque o objeto (nesse caso, `iOb`) ao qual a referência está sendo atribuída é de tipo `Gen<Integer>`. Logo, a referência retornada por `new` também deve ser de tipo `Gen<Integer>`. Se não for, ocorrerá um erro de tempo de compilação. Por exemplo, a atribuição a seguir causará um erro de tempo de compilação:

```
| iOb = new Gen<Double>(88.0); // Erro!
```

Uma vez que `iOb` é de tipo `Gen<Integer>`, não pode ser usada para referenciar um objeto de `Gen<Double>`. Essa verificação de tipos é um dos principais benefícios dos genéricos porque assegura a segurança dos tipos.

Como os comentários do programa informam, a atribuição

```
| iOb = new Gen<Integer>(88);
```

faz uso do *autoboxing* para encapsular o valor 88, que é um `int`, em um `Integer`. Isso funciona porque `Gen<Integer>` cria um construtor que recebe um argumento `Integer`. Já que um `Integer` é esperado, Java encapsulará automaticamente 88 dentro dele. É claro que a atribuição também poderia ter sido escrita explicitamente, da seguinte forma:

```
| iOb = new Gen<Integer>(new Integer(88));
```

No entanto, não nos beneficiaríamos usando essa versão.

Em seguida, o programa exibe o tipo de `ob` dentro de `iOb`, que é `Integer`. Depois, obtém o valor de `ob` usando a linha abaixo:

```
| int v = iOb.getob();
```

Já que o tipo de retorno de `getob()` é `T`, que foi substituído por `Integer` quando `iOb` foi declarada, ele também é `Integer`, que é encapsulado em `int` quando atribuído a `v`

(que é um **int**). Logo, não há necessidade de converter o tipo de retorno de **getob()** para **Integer**.

Agora, **GenDemo** declara um objeto de tipo **Gen<String>**:

```
| Gen<String> strOb = new Gen<String>("Generics Test");
```

Já que o argumento de tipo é **String**, **T** é substituído por **String** dentro de **Gen**. Isso cria (conceitualmente) uma versão **String** de **Gen**, como as linhas restantes do programa demonstram.

Genéricos só funcionam com objetos

Na declaração de uma instância de um tipo genérico, o argumento de tipo passado para o parâmetro de tipo deve ser um tipo de referência. Você não pode usar um tipo primitivo, como **int** ou **char**. Por exemplo, com **Gen**, é possível passar qualquer tipo de classe para **T**, mas você não pode passar um tipo primitivo para **T**. Logo, a declaração a seguir é inválida:

```
| Gen<int> intOb = new Gen<int>(53); // Erro, não pode usar um tipo primitivo
```

Certamente, não poder especificar um tipo primitivo não é uma restrição grave, porque você pode usar os encapsuladores de tipos (como fez o exemplo anterior) para encapsular um tipo primitivo. Além disso, o mecanismo Java de *autoboxing* e *autounboxing* torna o uso do encapsulador de tipos transparente.

Tipos genéricos diferem de acordo com seus argumentos de tipo

Um ponto-chave que devemos entender sobre os tipos genéricos é que uma referência de uma versão específica de um tipo genérico não tem compatibilidade de tipo com outra versão do mesmo tipo genérico. Por exemplo, supondo o programa que acabamos de mostrar, a linha de código abaixo está errada e não será compilada:

```
| iOb = strOb; // Errado!
```

Ainda que tanto **iOb** quanto **strOb** sejam de tipo **Gen<T>**, são referências a tipos diferentes porque seus parâmetros de tipo diferem. Isso faz parte da maneira como os genéricos adicionam segurança de tipos e evitam erros.

Classe genérica com dois parâmetros de tipo

Você pode declarar mais de um parâmetro de tipo em um tipo genérico. Para especificar dois ou mais parâmetros de tipo, apenas use uma lista separada por vírgulas. Por exemplo, a classe **TwoGen** abaixo é uma variação da classe **Gen** que tem dois parâmetros de tipo:

```
// Classe genérica simples com dois parâmetros de tipo: T e V.
class TwoGen<T, V> { ← Usa dois
    T ob1;
    V ob2;

    // Passa para o construtor referências a
    // objetos de tipo T e V.
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
}
```

```

// Exibe os tipos de T e V.
void showTypes() {
    System.out.println("Type of T is " +
        ob1.getClass().getName());
    System.out.println("Type of V is " +
        ob2.getClass().getName());
}

T getob1() {
    return ob1;
}

V getob2() {
    return ob2;
}
}

// Demonstra TwoGen.
class SimpGen {
    public static void main(String[] args) {

        TwoGen<Integer, String> tgObj = ←————— Aqui, Integer é
            new TwoGen<Integer, String>(88, "Generics"); passado para T e String
            é passado para V.

        // Exibe os tipos.
        tgObj.showTypes();

        // Obtém e exibe valores.
        int v = tgObj.getob1();
        System.out.println("value: " + v);

        String str = tgObj.getob2();
        System.out.println("value: " + str);
    }
}

```

A saída desse programa é mostrada abaixo:

```

Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics

```

Observe como **TwoGen** é declarada:

```
| class TwoGen<T, V> {
```

Ela especifica dois parâmetros de tipo, **T** e **V**, separados por uma vírgula. Uma vez que há dois parâmetros de tipo, dois argumentos de tipo devem ser passados para **TwoGen** quando um objeto for criado, como mostrado a seguir:

```
| TwoGen<Integer, String> tgObj =
|     new TwoGen<Integer, String>(88, "Generics");
```

Nesse caso, **T** é substituído por **Integer** e **V** é substituído por **String**. Embora aqui os dois argumentos de tipo sejam diferentes, é possível que ambos sejam iguais. Por exemplo, a linha de código a seguir é válida:

```
| TwoGen<String, String> x = new TwoGen<String, String>("A", "B");
```

Nesse exemplo, tanto **T** quanto **V** seriam de tipo **String**. Claro, se os argumentos de tipo fossem sempre iguais, dois parâmetros de tipo seriam desnecessários.

A forma geral de uma classe genérica

A sintaxe dos genéricos mostrada nos exemplos anteriores pode ser generalizada. Esta é a sintaxe de declaração de uma classe genérica:

```
class nome-classe<lista-parâm-tipo> { // ...
```

No contexto de uma atribuição, esta é a sintaxe de declaração de uma referência a uma classe genérica e criação de uma instância:

```
nome-classe<lista-arg-tipo> nome-var =
    new nome-classe<lista-arg-tipo> (lista-arg-cons);
```

Verificação do progresso

1. O tipo de dado tratado por uma classe genérica é passado para ela por um _____.
2. Um parâmetro de tipo pode receber um tipo primitivo?
3. Supondo a classe **Gen** que vimos no exemplo anterior, mostre como declarar uma referência **Gen** que opere com dados de tipo **Double**.

Pergunte ao especialista

P Já consigo ver que os genéricos são realmente um recurso poderoso. Eles são exclusivos de Java ou outras linguagens de computador dão suporte a um conceito semelhante?

R O conceito geral existente por trás dos genéricos é suportado por outras linguagens. Por exemplo, C++ dá suporte ao código genérico com o uso de *templates*. (Na verdade, o que Java chama de tipo parametrizado, C++ chama de template.) No entanto, os genéricos Java e os modelos C++ não são iguais e há algumas diferenças básicas entre as duas abordagens. Geralmente, a abordagem Java é mais fácil de usar. Outra linguagem que dá suporte a códigos genéricos é C#. Sua abordagem é mais parecida com a de Java.

Respostas:

1. parâmetro de tipo
2. Não.
3. `Gen<Double> d_obj;`

TIPOS LIMITADOS

Nos exemplos anteriores, os parâmetros de tipo podiam ser substituídos por qualquer tipo de classe. Em muitos casos isso é bom, mas às vezes é útil limitar os tipos que podem ser passados para um parâmetro de tipo. Por exemplo, suponhamos que você quisesse criar uma classe genérica que armazenasse um valor numérico e pudesse executar várias funções matemáticas, como calcular o recíproco ou obter o componente fracionário. Você também quer usar a classe para calcular esses valores para qualquer tipo de número, inclusive inteiro, **ponto flutuante** e **dupla precisão** (`Integer`, `Float` e `Double`). Logo, quer especificar o tipo dos números genericamente, usando um parâmetro de tipo. Para criar essa classe, você poderia testar algo assim:

```
// NumericFns tenta (sem sucesso) criar
// uma classe genérica que possa executar
// várias funções numéricas, como calcular
// o recíproco ou o componente fracionário, dado qualquer tipo de número.
class NumericFns<T> {
    T num;

    // Passa para o construtor uma referência a
    // um objeto numérico.
    NumericFns(T n) {
        num = n;
    }

    // Retorna o recíproco.
    double reciprocal() {
        return 1 / num.doubleValue(); // Erro!
    }

    // Retorna o componente fracionário.
    double fraction() {
        return num.doubleValue() - num.intValue(); // Erro!
    }

    // ...
}
```

Infelizmente, como foi escrita, `NumericFns` não será compilada, porque os dois métodos gerarão erros de tempo de compilação. Primeiro, examinemos o método `reciprocal()`, que tenta retornar o recíproco de `num`. Para fazê-lo, ele deve dividir 1 pelo valor de `num`. O valor de `num` é obtido com uma chamada a `doubleValue()`, que obtém a versão `double` do objeto numérico armazenado em `num`. Já que todas as classes numéricas, como `Integer` e `Double`, são subclasses de `Number`, e `Number` define o método `doubleValue()`, esse método está disponível para todas as classes de encapsuladores numéricos. O problema é que o compilador não tem como saber que você pretende criar objetos `NumericFns` usando somente tipos numéricos. Logo, quando você tentar compilar `NumericFns`, um erro será relatado indicando que o método `doubleValue()` é desconhecido. O mesmo tipo de erro ocorre duas vezes em `fraction()`, que deve chamar tanto `doubleValue()` quanto `intValue()`. As duas


```

System.out.println("Fractional component of iOb is " +
    iOb.fraction());

System.out.println();

NumericFns<Double> dOb = ← Double também pode
    new NumericFns<Double>(5.25); ser usado.

System.out.println("Reciprocal of dOb is " +
    dOb.reciprocal());
System.out.println("Fractional component of dOb is " +
    dOb.fraction());

// Essa parte não será compilada porque String não é
// subclasse de Number.
// NumericFns<String> strOb = new NumericFns<String>("Error"); ←
}

A saída é mostrada aqui:
```

String não pode ser
usado porque não é
subclasse de Number.

```

Reciprocal of iOb is 0.2
Fractional component of iOb is 0.0

Reciprocal of dOb is 0.19047619047619047
Fractional component of dOb is 0.25
```

Observe como **NumericFns** agora é declarada por essa linha:

```
| class NumericFns<T extends Number> {
```

Como agora o tipo **T** é limitado por **Number**, o compilador Java sabe que todos os objetos de tipo **T** podem chamar **doubleValue()** porque esse é um método declarado por **Number**. Isso já é por si só uma grande vantagem. No entanto, como bônus, a restrição de **T** também impede que objetos **NumericFns** não numéricos sejam criados. Por exemplo, se você remover os símbolos de comentário da linha do fim do programa e tentar recompilar, verá erros de tempo de compilação, porque **String** não é subclasse de **Number**.

Os tipos limitados são particularmente úteis quando é necessário assegurar que um parâmetro de tipo seja compatível com outro. Por exemplo, considere a classe a seguir chamada **Pair**, que armazena dois objetos que devem ser compatíveis:

```

class Pair<T, V extends T> { ← Aqui, V deve ser do
    T first;
    V second;

    Pair(T a, V b) {
        first = a;
        second = b;
    }

    // ...
}
```

Observe que **Pair** usa dois parâmetros de tipo, **T** e **V**, e que **V** estende **T**. Ou seja, **V** será igual a **T** ou a uma subclasse de **T**. Isso assegura que os dois argumentos do construtor de **Pair** sejam objetos do mesmo tipo ou de tipos relacionados. Por exemplo, as construções a seguir são válidas:

```
// Isto está certo porque T e V são Integer.  
Pair<Integer, Integer> x = new Pair<Integer, Integer>(1, 2);  
  
// Isto está certo porque Integer é uma subclasse de Number.  
Pair<Number, Integer> y = new Pair<Number, Integer>(10.4, 12);
```

No entanto, a mostrada aqui não é válida:

```
// Esta linha causa um erro, porque String não é  
// subclasse de Number  
Pair<Number, String> z = new Pair<Number, String>(10.4, "12");
```

Nesse caso, **String** não é subclasse de **Number**, o que viola o limite especificado por **Pair**.

Verificação do progresso

1. A palavra-chave _____ especifica um limite para um argumento de tipo.
2. Como deve ser declarado um tipo genérico **T** que tenha que ser subclasse de **Thread**?
3. Dado o código

```
| class X<T, V extends T> {  
  
    a declaração a seguir está correta?  
  
    |X<Integer, Double> x = new X<Integer, Double>(10, 1.1);
```

USANDO ARGUMENTOS CURINGAS

Mesmo sendo útil, às vezes a segurança de tipos pode invalidar construções perfeitamente aceitáveis. Dada a classe **NumericFns** mostrada no fim da seção anterior, suponhamos que você quisesse adicionar um método chamado **absEqual()** que retornasse **true** se dois objetos **NumericFns** contivessem números cujos valores absolutos fossem iguais. Além disso, você quer que esse método funcione apropriadamente, não importando o tipo de número que cada objeto contém. Por exemplo, se um objeto tiver o valor **Double** 1,25 e o outro tiver o valor **Float** -1,25, **absEqual()** retornará **true**. Uma maneira de implementar **absEqual()** é passar para ele um argumento **Nu-**

Respostas:

1. **extends**
2. **T extends Thread**
3. Não, porque **Double** não é subclasse de **Integer**.

NumericFns e então comparar o valor absoluto desse argumento com o valor absoluto do objeto chamador, só retornando verdadeiro se os valores forem iguais. Digamos que você quisesse poder chamar **absEqual()**, como mostrado aqui:

```
NumericFns<Double> dOb = new NumericFns<Double>(1.25);
NumericFns<Float> fOb = new NumericFns<Float>(-1.25);

if(dOb.absEqual(fOb))
    System.out.println("Absolute values are the same.");
else
    System.out.println("Absolute values differ.");
```

À primeira vista, criar **absEqual()** parece uma tarefa fácil. Infelizmente, os problemas começam a surgir assim que tentamos declarar um parâmetro de tipo **NumericFns**. Que tipo devemos especificar como parâmetro de **NumericFns**? Inicialmente, poderíamos pensar em uma solução como a dada a seguir, em que **T** é usado como parâmetro de tipo:

```
// Este código não funcionará!
// Determina se os valores absolutos de dois objetos são iguais.
boolean absEqual(NumericFns<T> ob) {
    if(Math.abs(num.doubleValue()) ==
        Math.abs(ob.num.doubleValue())) return true;

    return false;
}
```

Aqui, o método padrão **Math.abs()** é usado para obter o valor absoluto de cada número e então os valores são comparados. O problema dessa abordagem é que ela só funcionará com outros objetos **NumericFns** cujo tipo for igual ao do objeto chamador. Por exemplo, se o objeto chamador for de tipo **NumericFns<Integer>**, o parâmetro **ob** também deve ser de tipo **NumericFns<Integer>**. Ele não pode ser usado para comparar um objeto de tipo **NumericFns<Double>**. Portanto, essa abordagem não cria uma solução geral (isto é, genérica).

Para criar um método **absEqual()** genérico, você deve usar outro recurso dos genéricos Java: o *argumento curinga*. O argumento curinga é especificado pelo símbolo **?** e representa um tipo desconhecido. Com o uso de um curinga, veja uma maneira de criar o método **absEqual()**:

```
// Determina se os valores absolutos de dois
// objetos são iguais.
boolean absEqual(NumericFns<?> ob) { ← Observe o curinga.
    if(Math.abs(num.doubleValue()) ==
        Math.abs(ob.num.doubleValue())) return true;

    return false;
}
```

Aqui, **NumericFns<?>** equivale a qualquer tipo de objeto **NumericFns**, permitindo que dois objetos **NumericFns**, sejam quais forem, tenham seus valores absolutos comparados. O programa a seguir demonstra isso:

```

// Usa um curinga.
class NumericFns<T extends Number> {
    T num;

    // Passa para o construtor uma referência
    // a um objeto numérico.
    NumericFns(T n) {
        num = n;
    }

    // Retorna o recíproco.
    double reciprocal() {
        return 1 / num.doubleValue();
    }

    // Retorna o componente fracionário.
    double fraction() {
        return num.doubleValue() - num.intValue();
    }

    // Determina se os valores absolutos de dois
    // objetos são iguais.
    boolean absEqual(NumericFns<?> ob) {
        if(Math.abs(num.doubleValue()) ==
           Math.abs(ob.num.doubleValue())) return true;

        return false;
    }

    // ...
}

// Demonstra um curinga.
class WildcardDemo {
    public static void main(String[] args) {

        NumericFns<Integer> iOb =
            new NumericFns<Integer>(6);

        NumericFns<Double> dOb =
            new NumericFns<Double>(-6.0);

        NumericFns<Long> lOb =
            new NumericFns<Long>(5L);

        System.out.println("Testing iOb and dOb.");
        if(iOb.absEqual(dOb)) ←———— Nesta chamada,
            System.out.println("Absolute values are equal."); o tipo curinga
        else                                     equivale a Double.
            System.out.println("Absolute values differ.");
    }
}

```

```
System.out.println();

System.out.println("Testing iOb and dOb.");
if(iOb.absEqual(dOb)) ← Nessa chamada, o
    System.out.println("Absolute values are equal.");
else
    System.out.println("Absolute values differ.");

}
}
```

A saída é mostrada abaixo:

```
Testing iOb and dOb.
Absolute values are equal.

Testing iOb and lOb.
Absolute values differ.
```

No programa, observe estas duas chamadas a **absEqual()**:

```
if(iOb.absEqual(dOb))

if(iOb.absEqual(lOb))
```

Na primeira chamada, **iOb** é um objeto de tipo **NumericFns<Integer>** e **dOb** é um objeto de tipo **NumericFns<Double>**. No entanto, com o uso de um curinga, é possível **iOb** passar **dOb** na chamada a **absEqual()**. O mesmo se aplica à segunda chamada, em que um objeto de tipo **NumericFns<Long>** é passado.

Um último ponto: é importante entender que o curinga não afeta os tipos de objetos **NumericFns** que podem ser criados. Isso é controlado pela cláusula **extends** na declaração de **NumericFns**. O curinga apenas permite que qualquer tipo **NumericFns válido** seja usado.

CURINGAS LIMITADOS

Os argumentos curingas podem ser limitados de maneira semelhante ao que ocorre com o parâmetro de tipo. Um curinga limitado é particularmente importante quando estamos criando um método projetado para operar somente com objetos que sejam subclasses de uma superclasse específica. Para entender o porquê, examinemos um exemplo simples. Considere o conjunto de classes a seguir:

```
class A {
    // ...
}

class B extends A {
    // ...
}

class C extends A {
    // ...
```

```

    }

    // Observe que D NÃO estende A.
    class D {
        // ...
    }
}

```

Aqui, a classe **A** é estendida pelas classes **B** e **C**, mas não por **D**.

Em seguida, considere a classe genérica simples mostrada abaixo:

```

// Classe genérica simples.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }
}

```

Gen usa um parâmetro de tipo, que especifica o tipo de objeto armazenado em **ob**. Já que **T** é ilimitado, seu tipo é irrestrito. Isto é, **T** pode ser de qualquer tipo de classe.

Agora, suponhamos que você quisesse criar um método que recebesse como argumento qualquer tipo de objeto **Gen**, contanto que seu parâmetro de tipo seja **A** ou subclasse de **A**. Em outras palavras, você quer criar um método que opere somente com objetos de **Gen<tipo>**, em que *tipo* é **A** ou subclasse de **A**. Para fazê-lo, deve usar um curinga limitado. Por exemplo, veja um método chamado **test()** que só aceita como argumento objetos **Gen** cujo parâmetro de tipo é **A** ou subclasse de **A**:

```

// Aqui, o símbolo ? equivalerá a A ou a qualquer tipo
// de classe que estenda A.
static void test(Gen<? extends A> o) {
    // ...
}

```

A classe a seguir demonstra os tipos de objetos **Gen** que podem ser passados para **test()**.

```

class UseBoundedWildcard {
    // Aqui, o símbolo ? equivalerá a A ou a qualquer tipo
    // de classe que estenda A.
    static void test(Gen<? extends A> o) { ←————— Usa um curinga
        // ...
    }

    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();
        D d = new D();

        Gen<A> w = new Gen<A>(a);
        Gen<B> w2 = new Gen<B>(b);
    }
}

```

```

Gen<C> w3 = new Gen<C>(c);
Gen<D> w4 = new Gen<D>(d);

// Estas chamadas a test() estão corretas.
test(w); [                         Estas chamadas são
test(w2); ]                         válidas porque w, w2 e w3
test(w3); ]                         são subclasses de A.

// Não pode chamar test() com w4 porque
// ele não é um objeto de uma classe
// que herde A.
//   test(w4); // Erro! ←           Não é válido porque w4
}                                     não é subclasse de A.
}

```

Em **main()**, objetos de tipo **A**, **B**, **C** e **D** são criados. Em seguida, eles são usados na criação de quatro objetos **Gen**, um para cada tipo. Para concluir, quatro chamadas a **test()** são feitas, com a última sendo desativada por um comentário. As três primeiras chamadas são válidas porque **w**, **w2** e **w3** são objetos **Gen** cujo tipo é **A** ou subclasse de **A**. No entanto, a última chamada a **test()** não é válida, porque **w4** é um objeto de tipo **D**, que não é derivado de **A**. Logo, o método não aceitará **w4** devido à restrição do curinga.

Em geral, para estabelecer o limite superior de um curinga, usamos o tipo de expressão abaixo:

`<? extends superclasse>`

na qual *superclasse* é o nome da classe que serve como limite superior. Lembre-se, essa é uma cláusula inclusiva porque a classe que forma o limite superior (especificada por *superclasse*) também faz parte do limite.

Você também pode especificar um limite inferior para um curinga adicionando uma cláusula **super** à sua declaração. Esta é a forma geral:

`<? super subclasse>`

Nesse caso, só classes que sejam superclasses de *subclasse* são argumentos aceitáveis. A cláusula é inclusiva.

Verificação do progresso

- Para especificar um argumento curinga, devemos usar _____.
- Um argumento curinga equivale a qualquer tipo de referência. Verdadeiro ou falso?
- Um curinga pode ser limitado?
- Nesta expressão, que tipos podem ser representados pelo curinga?

```
|void myMeth(XYZ<? extends Thread> trdOb) { // ...
```

Respostas:

- o símbolo ?
- Verdadeiro.
- Sim.
- O curinga pode representar o tipo **Thread** ou uma subclasse de **Thread**.

Pergunte ao especialista

P Posso converter uma instância de uma classe genérica em outra?

R Sim, mas seu uso será um pouco limitado. Você pode converter uma instância de uma classe genérica em outra, mas só se as duas forem compatíveis e seus argumentos de tipo forem iguais. Por exemplo, imagine uma classe genérica chamada **Gen** declarada da forma a seguir:

```
| class Gen<T> { // ...
```

Em seguida, suponha que **x** fosse declarada como mostrado aqui:

```
| Gen<Integer> x = new Gen<Integer>()
```

Então, esta conversão seria válida:

```
| (Gen<Integer>) x // válido
```

porque **x** é uma instância de **Gen<Integer>**. Mas a conversão

```
| (Gen<Long>) x // inválido
```

não seria válida, porque **x** não é instância de **Gen<Long>**.

MÉTODOS GENÉRICOS

Como os exemplos anteriores mostraram, os métodos de uma classe genérica podem fazer uso do parâmetro de tipo da classe e, portanto, são automaticamente genéricos de acordo com o parâmetro de tipo. Entretanto, podemos declarar um método genérico que use um ou mais parâmetros de tipo exclusivamente seus. Além disso, podemos criar um método genérico embutido em uma classe não genérica.

O programa a seguir declara uma classe não genérica chamada **GenericMethodDemo** e um método genérico estático dentro dessa classe chamado **arraysEqual()**. Esse método determina se dois arrays contêm os mesmos elementos, na mesma ordem. Pode ser usado para comparar dois arrays, sejam eles quais forem, desde que sejam de tipos iguais ou compatíveis.

```
// Demonstra um método genérico simples.
class GenericMethodDemo {
    // Determina se o conteúdo de dois arrays é igual.
    static <T, V extends T> boolean arraysEqual(T[] x, V[] y) {
        // Se os tamanhos dos arrays forem diferentes, os arrays também serão.
        if(x.length != y.length) return false;

        for(int i=0; i < x.length; i++)
            if(!x[i].equals(y[i])) return false; // os arrays são diferentes

        return true; // os conteúdos dos arrays são equivalentes
    }
}
```

Um método genérico.

```

public static void main(String[] args) {

    Integer[] nums = { 1, 2, 3, 4, 5 };
    Integer[] nums2 = { 1, 2, 3, 4, 5 };
    Integer[] nums3 = { 1, 2, 7, 4, 5 };
    Integer[] nums4 = { 1, 2, 7, 4, 5, 6 };

    if(arraysEqual(nums, nums)) ←———— Os argumentos de tipo de
        System.out.println("nums equals nums");
        T e V são determinados
        implicitamente quando o
        método é chamado.

    if(arraysEqual(nums, nums2))
        System.out.println("nums equals nums2");

    if(arraysEqual(nums, nums3))
        System.out.println("nums equals nums3");

    if(arraysEqual(nums, nums4))
        System.out.println("nums equals nums4");

    // Cria um array de Doubles
    Double[] dvals = { 1.1, 2.2, 3.3, 4.4, 5.5 };

    // Essa parte não será compilada, porque nums e dvals
    // não são do mesmo tipo.
    //     if(arraysEqual(nums, dvals))
    //         System.out.println("nums equals dvals");
}
}

```

A saída do programa é mostrada aqui:

```

| nums equals nums
| nums equals nums2

```

Examinemos **arraysEqual()** mais de perto. Em primeiro lugar, observe como ele é declarado pela linha a seguir:

```
| static <T, V extends T> boolean arraysEqual(T[] x, V[] y) {
```

Os parâmetros de tipo são declarados *antes* do tipo de retorno do método. Em segundo lugar, observe que o tipo **V** tem como limite superior **T**. Logo, **V** deve ser igual ao tipo **T** ou ser subclasse de **T**. Esse relacionamento impõe que **arraysEqual()** só seja chamado com argumentos compatíveis entre si. Observe também que **arraysEqual()** é estático, o que permite que seja chamado independentemente de qualquer objeto. É bom ressaltar, no entanto, que os métodos genéricos podem ser estáticos ou não estáticos. Não há restrições com relação a isso.

Agora, observe como **arraysEqual()** é chamado dentro de **main()** com o uso da sintaxe de chamada comum, sem necessidade de especificação de argumentos de tipo. Isso ocorre porque os tipos dos argumentos são identificados automaticamente e os tipos de **T** e **V** são ajustados de acordo. Por exemplo, na primeira chamada,

```
| if(arraysEqual(nums, nums))
```

o tipo base do primeiro argumento é **Integer**, o que faz **T** ser substituído por **Integer**. O tipo base do segundo argumento também é **Integer**, o que também o faz substituir **V**. Logo, a chamada a **arraysEqual()** é válida e os dois arrays podem ser comparados.

Vejamos então o código desativado por comentário, mostrado a seguir:

```
//    if(arraysEqual(nums, dvals))
//        System.out.println("nums equals dvals");
```

Se você remover o símbolo de comentário e tentar compilar o programa, verá uma mensagem de erro. Isso ocorre porque o parâmetro de tipo **V** é limitado por **T** na cláusula **extends** da declaração de **V**. Ou seja, **V** deve ser igual ao tipo **T** ou ser subclasse de **T**. Nesse caso, o primeiro argumento é de tipo **Integer**, o que transforma **T** em **Integer**, mas o segundo argumento é de tipo **Double**, que não é subclasse de **Integer**. Isso invalida a chamada a **arraysEqual()** e resulta em um erro de discrepância de tipos no tempo de compilação.

A sintaxe usada na criação de **arraysEqual()** pode ser generalizada. Esta é a sintaxe de um método genérico:

```
<lista-parâm-tipo>tipo-ret nome-mét(lista-parâm) { // ...
```

Não importa o caso, *lista-parâm-tipo* é sempre uma lista de parâmetros de tipo separada por vírgulas. Observe que, para um método genérico, a lista de parâmetros de tipo precede o tipo de retorno.

CONSTRUTORES GENÉRICOS

Um construtor pode ser genérico mesmo que sua classe não o seja. Por exemplo, no programa a seguir, a classe **Summation** não é genérica, mas seu construtor é.

```
// Usa um construtor genérico.
class Summation {
    private int sum;

    <T extends Number> Summation(T arg) { ←————— Construtor genérico.
        sum = 0;

        for(int i=0; i <= arg.intValue(); i++)
            sum += i;
    }

    int getSum() {
        return sum;
    }
}

class GenConsDemo {
    public static void main(String[] args) {
        Summation ob = new Summation(4.0);

        System.out.println("Summation of 4.0 is " + ob.getSum());
    }
}
```

A classe **Summation** calcula e encapsula a soma do valor numérico passado para seu construtor. Nesse caso, estamos usando o total de N que é igual à soma de todos os números inteiros entre 0 e N . Já que **Summation()** especifica um parâmetro de tipo que é limitado por **Number**, um objeto **Summation** pode ser construído com o uso de qualquer tipo numérico, inclusive **Integer**, **Float** ou **Double**. Qualquer que seja o tipo numérico usado, seu valor será convertido em **Integer** com uma chama da a **intValue()** e a soma será calculada. Portanto, não é necessário que a classe **Summation** seja genérica; apenas um construtor genérico é necessário.

HIERARQUIAS DE CLASSES GENÉRICAS

Uma classe genérica pode fazer parte de uma hierarquia de classes da mesma maneira que uma classe não genérica. Logo, ela pode agir como uma superclasse ou ser uma subclasse. A diferença-chave entre hierarquias genéricas e não genéricas é que, em uma hierarquia genérica, qualquer argumento de tipo que for requerido por uma superclasse genérica deve ser passado para cima na hierarquia por todas as subclasses. Isso é semelhante à maneira como os argumentos de construtor devem ser passados para cima em uma hierarquia.

Aqui está um exemplo simples de hierarquia que usa uma superclasse genérica:

```
// Uma hierarquia de classes genéricas simples.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }
}

// Uma subclasse de Gen.
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}
```

Nessa hierarquia, **Gen2** estende a classe genérica **Gen**. Observe como **Gen2** é declarada pela linha a seguir:

```
| class Gen2<T> extends Gen<T> {
```

O parâmetro de tipo **T** é especificado por **Gen2** e também é passado para **Gen** na cláusula **extends**. Ou seja, qualquer que seja o tipo passado para **Gen2**, ele também será passado para **Gen**. Por exemplo, a declaração

```
| Gen2<Integer> num = new Gen2<Integer>(100);
```

passa **Integer** como parâmetro de tipo para **Gen**. Logo, o objeto **ob** da parte de **Gen2** referente a **Gen** será de tipo **Integer**.

Observe também que **Gen2** não usa o parâmetro de tipo **T** exceto para passá-lo para a superclasse **Gen**. Logo, mesmo se a subclasse de uma superclasse genérica não tivesse de ser genérica, continuaria tendo que especificar o(s) parâmetro(s) de tipo requeridos por sua superclasse genérica. É claro que a subclasse pode adicionar parâmetros de tipo próprios, se necessário.

Também é perfeitamente aceitável uma classe não genérica ser superclasse de uma subclasse genérica. Por exemplo:

```
// Uma classe não genérica pode ser superclasse
// de uma subclasse genérica.

// Classe não genérica.
class NonGen {
    int num;

    NonGen(int i) {
        num = i;
    }

    int getnum() {
        return num;
    }
}

// Subclasse genérica.
class Gen<T> extends NonGen {
    T ob; // declara um objeto de tipo T

    // Passa para o construtor uma referência a
    // um objeto de tipo T.
    Gen(T o, int i) {
        super(i);
        ob = o;
    }

    // Retorna ob.
    T getob() {
        return ob;
    }
}
```

Observe como **Gen** herda **NonGen** na declaração a seguir:

```
| class Gen<T> extends NonGen {
```

Como **NonGen** não é genérica, não é especificado um argumento de tipo. Logo, ainda que **Gen** declare o parâmetro de tipo **T**, **NonGen** não precisa dele (nem pode usá-lo). Ou seja, **NonGen** é herdada por **Gen** normalmente. Condições especiais não são aplicáveis.

Pergunte ao especialista

P Em uma hierarquia de classes genéricas, um método com um parâmetro de tipo pode ser sobreposto?

R Sim. Um método genérico de uma hierarquia de classes genéricas pode ser sobreposto como qualquer outro método. Considere esse exemplo em que o método **getob()** é sobreposto:

```
// Sobreposição de um método genérico de uma classe genérica.
class Gen<T> {
    T ob; // declara um objeto de tipo T

    // Passa para o construtor uma referência a
    // um objeto de tipo T.
    Gen(T o) {
        ob = o;
    }

    // Retorna ob.
    T getob() {
        System.out.print("Gen's getob(): " );
        return ob;
    }
}

// Subclasse de Gen que sobreponde getob().
class Gen2<T> extends Gen<T> {

    Gen2(T o) {
        super(o);
    }

    // Sobrepõe getob().
    T getob() {
        System.out.print("Gen2's getob(): " );
        return ob;
    }
}

// Demonstra a sobreposição de um método genérico.
class OverrideDemo {
    public static void main(String[] args) {

        // Uma referência Gen que pode apontar para qualquer tipo de objeto Gen.
        Gen<?> gRef;

        // Cria um objeto Gen para Integers.
        Gen<Integer> iOb = new Gen<Integer>(88);

        // Cria um objeto Gen2 para Integers.
        Gen2<Integer> iOb2 = new Gen2<Integer>(99);
    }
}
```

```
// Cria um objeto Gen2 para Strings.
Gen2<String> strOb2 = new Gen2<String> ("Generics Test");

gRef = iOb;
System.out.println(gRef.getob());

gRef = iOb2;
System.out.println(gRef.getob());

gRef = strOb2;
System.out.println(gRef.getob());
}
```

A saída é mostrada aqui:

```
Gen's getob(): 88
Gen2's getob(): 99
Gen2's getob(): Generics Test
```

No programa, observe que a referência genérica **gRef** é usada para chamar **getob()** em objetos **Gen** e **Gen2**. Como a saída confirma, a versão sobreposta de **getob()** é chamada para objetos de tipo **Gen2**, mas a versão da superclasse é chamada para objetos de tipo **Gen**.

Verificação do progresso

1. Um método ou construtor pode ser genérico mesmo se sua classe não for?
2. Mostre como declarar um método genérico chamado **myMeth()** que use um argumento de tipo genérico. Faça-o retornar um argumento desse tipo.
3. Uma classe genérica não pode ser herdada. Verdadeiro ou falso?

INTERFACES GENÉRICAS

Além das classes e métodos genéricos, você também pode ter interfaces genéricas. As interfaces genéricas são especificadas como as classes genéricas. Abaixo, temos um exemplo. Ele cria uma interface chamada **Containment**, que pode ser implementada por classes que armazenem um ou mais valores. Também declara um método chamado **contains()** que determina se um valor especificado está contido no objeto chamador.

```
// Exemplo de interface genérica.

// Uma interface genérica que lida com armazenamento.
```

Respostas:

1. Sim.
2. $<T> T myMeth(T o)$
3. Falso.

```

// Esta interface requer que a classe
// usuária tenha um ou mais valores.
interface Containment<T> { ← Interface genérica.
    // O método contains() verifica se
    // um item especificado está contido dentro
    // de um objeto que implementa Containment.
    boolean contains(T o);
}

// Implementa Containment usando um array para
// armazenar os valores.
class MyClass<T> implements Containment<T> { ← Toda classe que
    T[] arrayRef;                                implemente uma
                                                interface genérica
                                                também deve ser
                                                genérica.

    MyClass(T[] o) {
        arrayRef = o;
    }

    // Implementa contains().
    public boolean contains(T o) {
        for(T x : arrayRef)
            if(x.equals(o)) return true;
        return false;
    }
}

class GenIFDemo {
    public static void main(String[] args) {
        Integer[] x = { 1, 2, 3 };

        MyClass<Integer> ob = new MyClass<Integer>(x);

        if(ob.contains(2))
            System.out.println("2 is in ob");
        else
            System.out.println("2 is NOT in ob");

        if(ob.contains(5))
            System.out.println("5 is in ob");
        else
            System.out.println("5 is NOT in ob");

        // A parte a seguir não é válida porque ob
        // é um objeto Containment de tipo Integer e 9.25 é
        // um valor Double.
        // if(ob.contains(9.25)) // Inválido!
        //     System.out.println("9.25 is in ob");
    }
}

```

A saída é mostrada aqui:

```
| 2 is in ob
| 5 is NOT in ob
```

Embora a maioria dos aspectos desse programa seja de fácil compreensão, algumas observações importantes devem ser feitas. Primeiro, observe que **Containment** é declarada assim:

```
| interface Containment<T> {
```

Normalmente, uma interface genérica é declarada da mesma forma que uma classe genérica. No caso em questão, o parâmetro de tipo **T** especifica o tipo dos objetos contidos.

Em seguida, **Containment** é implementada por **MyClass**. Observe a declaração de **MyClass**, mostrada aqui:

```
| class MyClass<T> implements Containment<T> {
```

Em geral, quando uma classe implementa uma interface genérica, essa classe também deve ser genérica, pelo menos ao ponto de usar um parâmetro de tipo passado para a interface. Por exemplo, a tentativa a seguir de declarar **MyClass** está incorreta:

```
| class MyClass implements Containment<T> { // Errado!
```

Essa declaração está errada porque **MyClass** não declara um parâmetro de tipo, ou seja, não há como passar um para **Containment**. O identificador **T** é desconhecido e o compilador relatará um erro. É claro que, se uma classe implementar um *tipo específico* de interface genérica, como mostrado abaixo,

```
| class MyClass implements Containment<Double> { // Correto
```

a classe que o está implementando não precisa ser genérica.

Como era de se esperar, o(s) parâmetro(s) de tipo especificado(s) por uma interface genérica pode(m) ser limitado(s). Isso nos permite limitar o tipo de dado para o qual a interface pode ser implementada. Por exemplo, se quiséssemos limitar **Containment** aos tipos numéricos, poderíamos declará-la assim:

```
| interface Containment<T extends Number> {
```

Nesse caso, qualquer classe usuária deve passar para **Containment** um argumento de tipo com o mesmo limite. Por exemplo, agora **MyClass** deve ser declarada como mostrado aqui:

```
| class MyClass<T extends Number> implements Containment<T> {
```

Preste atenção na maneira como o parâmetro de tipo **T** é declarado por **MyClass** e então passado para **Containment**. Já que agora **Containment** requer um tipo que estenda **Number**, a classe que a está implementando (**MyClass**, nesse exemplo) deve especificar o mesmo limite. Além disso, uma vez que esse limite seja estabelecido, não há necessidade de especificá-lo novamente na cláusula **implements**. Na verdade,

seria errado fazê-lo. Por exemplo, a declaração seguinte está incorreta e não será compilada:

```
// Este código está errado!
class MyClass<T extends Number>
    implements Containment<T extends Number> { // Errado!
```

Uma vez que o parâmetro de tipo tiver sido estabelecido, ele será passado para a interface sem nenhuma modificação.

Esta é a sintaxe generalizada de uma interface genérica:

```
interface nome-interface<lista-parâm-tipo> { // ...}
```

Aqui, *lista-parâm-tipo* é uma lista de parâmetros de tipo separada por vírgulas. Quando uma interface genérica for implementada, você deve especificar os argumentos de tipo, como mostrado abaixo:

```
class nome-classe<lista-parâm-tipo>
    implements nome-interface<lista-parâm-tipo> {
```

TENTE ISTO 14-1 Crie uma pilha genérica simples

```
IGenSimpleStack.java
SimpleStackExc.java
GenSimpleStack.java
GenSimpleStackDemo.java
```

Uma das vantagens mais arrojadas que os genéricos trazem à programação é a possibilidade de construção de um código confiável e reutilizável. Como mencionado no início deste capítulo, muitos algoritmos são iguais, não importando o tipo de dados em que são usados. Por exemplo, a funcionalidade básica de uma pilha é a mesma, não importando que tipo de objetos está sendo armazenado. Em vez de criar uma classe de pilha separada para cada tipo de objeto, você pode construir uma solução genérica para ser usada com qualquer tipo. Portanto, o ciclo de desenvolvimento composto por projeto, codificação, teste e depuração só ocorrerá uma vez quando você criar uma solução genérica – e não repetidamente sempre que uma pilha for necessária para um novo tipo de dado.

Neste projeto, você adaptará o exemplo de pilha simples que vem desenvolvendo desde a seção Tente isto 5-2, tornando-a genérica. O projeto representa a evolução final da pilha. Ele inclui uma interface genérica que define as operações da pilha, uma implementação de pilha genérica de tamanho fixo e as classes de exceção. Essa versão da pilha será chamada de **GenSimpleStack**.

PASSO A PASSO

1. A primeira etapa da criação de uma pilha genérica é criar uma interface genérica que descreva as duas operações da pilha. A versão genérica da inter-

face de pilha se chama **IGenSimpleStack** e é mostrada abaixo. Insira essa interface em um arquivo chamado **IGenSimpleStack.java**.

```
// Interface genérica para uma pilha simples.
public interface IGenSimpleStack<T> {

    // Insere um item na pilha.
    void push(T item) throws StackFullException;

    // Retira um item da pilha.
    T pop() throws StackEmptyException;

    // Retorna true se a pilha estiver vazia.
    boolean isEmpty();

    // Retorna true se a pilha estiver cheia.
    boolean isFull();
}
```

Observe que essa interface é semelhante à versão não genérica desenvolvida na seção Tente isto 10-1, exceto pelo tipo de dado ser especificado por **T** em vez de **char**.

2. A pilha genérica pode usar as mesmas classes de exceção não genéricas desenvolvidas na seção Tente isto 10-1. Lembre-se, elas encapsulam os dois erros de pilha cheia ou vazia. Não são classes genéricas porque são iguais não importando o tipo de dado que está armazenado na pilha. Para sua conveniência, estamos mostrando-as abaixo novamente. Elas devem estar no arquivo **SimpleStackExc.java**.

```
// Exceção para erros de pilha cheia.
class StackFullException extends Exception {
    int size;

    StackFullException(int s) {
        super("Stack Full");
        size = s;
    }

    public String toString() {
        return "\nStack is full. Maximum size is " + size;
    }
}

// Exceção para erros de pilha vazia.
class StackEmptyException extends Exception {

    StackEmptyException() {
        super("Stack Empty");
    }

    public String toString() {
        return "\nStack is empty.";
    }
}
```

3. Agora, crie um arquivo chamado **GenSimpleStack.java**. Nesse arquivo, insira o código a seguir, que implementa uma pilha genérica simples de tamanho fixo:

```
// Pilha genérica simples de tamanho fixo.
class GenSimpleStack<T> implements IGenSimpleStack<T> {
    private T[] data; // esse array contém a pilha
    private int tos; // índice do topo da pilha

    // Constrói uma pilha vazia com o array dado como espaço de
    // armazenamento.
    GenSimpleStack(T[] arrayRef) {
        data = arrayRef;
        tos = 0;
    }

    // Insere um item na pilha.
    public void push(T obj) throws StackFullException {
        if(isFull())
            throw new StackFullException(data.length);

        data[tos] = obj;
        tos++;
    }

    // Retira um item da pilha.
    public T pop() throws StackEmptyException {
        if(isEmpty())
            throw new StackEmptyException();

        tos--;
        return data[tos];
    }

    // Retorna true se a pilha estiver vazia.
    public boolean isEmpty() {
        return tos==0;
    }

    // Retorna true se a pilha estiver cheia.
    public boolean isFull() {
        return tos==data.length;
    }
}
```

GenSimpleStack é uma classe genérica com parâmetro de tipo **T**, que especifica o tipo de dado armazenado na pilha. Observe que **T** também é passado para a interface **IGenSimpleStack**.

Preste atenção no construtor de **GenSimpleStack**. Ele recebe uma referência a um array que será usado para conter a pilha. Logo, para construir um **GenSimpleStack**, primeiro você terá de criar um array de tipo compatível com os objetos a serem armazenados e de tamanho suficiente para conter

a quantidade a ser inserida na pilha. Por exemplo, a sequência a seguir mostra como criar uma pilha de 10 elementos contendo strings:

```
| String[] strArray = new String[10];
| GenSimpleStack<String> strStack = new GenSimpleStack<String>(strArray);
```

4. Para demonstrar **GenSimpleStack**, crie um arquivo chamado **GenSimpleStack-Demo.java** e insira nele o código a seguir.

```
/*
   Tente isto 14-1

   Demonstra uma classe genérica de pilha simples.
*/

class GenSimpleStackDemo {
    public static void main(String[] args) {
        int i;
        Integer[] nums = new Integer[5];
        String[] strs = new String[3];

        // primeiro cria uma pilha de inteiros
        GenSimpleStack<Integer> intStack = new GenSimpleStack<Integer>(nums);

        System.out.println("Demonstrating Integer stack.");

        // usa intStack
        try {

            System.out.print("Pushing: ");
            // insere inteiros em intStack
            for(i = 0; !intStack.isFull(); i++) {
                System.out.print(i);
                intStack.push(i);
            }

            System.out.println();

            // retira inteiros de intStack
            System.out.print("Popping: ");
            while(!intStack.isEmpty())
                System.out.print(intStack.pop());

            System.out.println();
        } catch (StackFullException exc) {
            System.out.println(exc);
        } catch (StackEmptyException exc) {
            System.out.println(exc);
        }
        // em seguida, cria uma pilha para strings
        GenSimpleStack<String> strStack = new GenSimpleStack<String>(strs);

        System.out.println("\nDemonstrating String stack.");
    }
}
```

```

// agora, usa strStack
try {

    System.out.println("Pushing: alpha beta gamma");

    // insere strings em strStack
    strStack.push("alpha");
    strStack.push("beta");
    strStack.push("gamma");

    // remove Strings de strStack
    System.out.print("Popping: ");
    while(!strStack.isEmpty())
        System.out.print(strStack.pop() + " ");

    System.out.println();
} catch (StackFullException exc) {
    System.out.println(exc);
} catch (StackEmptyException exc) {
    System.out.println(exc);
} finally {
}
}

```

5. Compile todos os arquivos e execute **GenSimpleStackDemo**. Você verá a saída mostrada aqui:

```

Demonstrating Integer stack.
Pushing: 01234
Popping: 43210

Demonstrating String stack.
Pushing: alpha beta gamma
Popping: gamma beta alpha

```

TIPOS BRUTOS E CÓDIGO LEGADO

Como o suporte aos genéricos não existia antes do JDK 5, era necessário que Java fornecesse algum meio de os códigos antigos anteriores aos genéricos fazerem a transição. Resumindo, todos os códigos anteriores aos genéricos tinham que ser ao mesmo tempo funcionais e compatíveis com novos códigos que usassem os genéricos. Ou seja, os códigos pré-genéricos devem funcionar com os genéricos e os códigos genéricos têm de funcionar com os códigos pré-genéricos.

Para realizar a transição para os genéricos, Java permite que uma classe genérica seja usada sem nenhum argumento de tipo. Isso cria um *tipo bruto* para a classe. Esse tipo bruto é compatível com códigos legados, que não têm conhecimento dos

genéricos. A principal desvantagem do uso do tipo bruto é a segurança de tipos dos genéricos ser perdida.

Veja um exemplo que mostra um tipo bruto em ação:

```
// Demonstra um tipo bruto.
class Gen<T> {
    T ob; // declara uma referência a um objeto de tipo T

    // Passa para o construtor uma referência a
    // um objeto de tipo T.
    Gen(T o) {
        ob = o;
    }

    // Retorna ob.
    T getob() {
        return ob;
    }
}

// Demonstra o tipo bruto.
class RawDemo {
    public static void main(String[] args) {

        // Cria um objeto Gen para Integers.
        Gen<Integer> iOb = new Gen<Integer>(88);

        // Cria um objeto Gen para Strings.
        Gen<String> strOb = new Gen<String>("Generics Test");

        // Cria um objeto Gen de tipo bruto e dá a ele
        // um valor Double.                                         Quando nenhum
                                                               argumento de tipo é
                                                               fornecido, um tipo
                                                               bruto é criado.
        Gen raw = new Gen(new Double(98.6)); ←

        // Essa coerção é necessária porque o tipo é desconhecido.
        double d = (Double) raw.getob();
        System.out.println("value: " + d);

        // O uso de um tipo bruto pode levar a exceções de tempo de execução.
        // Aqui estão alguns exemplos.

        // A coerção a seguir causa um erro de tempo de execução!
        //     int i = (Integer) raw.getob(); // erro de tempo de execução
                                                               Os tipos brutos
                                                               sobrepõem a
                                                               segurança de tipos.

        // Essa atribuição sobrepõe a segurança de tipos.
        strOb = raw; // Correto, mas pode gerar erros ←
                                                               segurança de tipos.

        //     String str = strOb.getob(); // erro de tempo de execução

        // Essa atribuição também sobrepõe a segurança de tipos.
        raw = iOb; // Correto, mas pode gerar erros
        //     d = (Double) raw.getob(); // erro de tempo de execução
    }
}
```

Esse programa contém várias coisas interessantes. Primeiro, um tipo bruto da classe genérica **Gen** é criado pela declaração a seguir:

```
| Gen raw = new Gen(new Double(98.6));
```

Observe que nenhum argumento de tipo é especificado. Isso cria um objeto **Gen** cujo tipo **T** é substituído por **Object**.

Um tipo bruto não garante a segurança de tipos. Logo, uma variável de tipo bruto pode receber uma referência a qualquer tipo de objeto **Gen**. O inverso também é permitido, em que uma variável de um tipo **Gen** específico pode receber uma referência a um objeto **Gen** bruto. No entanto, as duas operações são potencialmente inseguras, porque o mecanismo de verificação de tipos dos genéricos é ignorado.

Essa falta de segurança de tipos é ilustrada pelas linhas desativadas por comentário no fim do programa. Examinemos cada caso. Primeiro, considere a situação a seguir:

```
// int i = (Integer) raw.getob(); // erro de tempo de execução
```

Nessa instrução, o valor de **ob** dentro de **raw** é obtido e induzido para **Integer**. O problema é que **raw** contém um valor **Double** e não um valor inteiro. No entanto, isso não pode ser detectado no tempo de compilação, porque o tipo de **raw** é desconhecido. Logo, essa instrução falha no tempo de execução.

A próxima sequência atribui a **strOb** (uma referência de tipo **Gen<String>**) uma referência a um objeto **Gen** bruto:

```
| strOb = raw; // Correto, mas pode provocar erros
// String str = strOb.getob(); // erro de tempo de execução
```

A atribuição está sintaticamente correta, mas é questionável. Uma vez que **strOb** é de tipo **Gen<String>**, supõe-se que contenha um **String**. No entanto, após a atribuição, o objeto referenciado por **strOb** armazena um **Double**. Logo, quando for feita uma tentativa de atribuir o conteúdo de **strOb** a **str**, isso resultará em um erro de tempo de execução, porque agora **strOb** contém um **Double**. Ou seja, a atribuição de uma referência bruta a uma referência genérica ignora o mecanismo de segurança de tipos.

A sequência abaixo inverte o caso anterior:

```
| raw = iOb; // Correto, mas pode provocar erros
// d = (Double) raw.getob(); // erro de tempo de execução
```

Aqui, uma referência genérica é atribuída a uma variável de referência bruta. Embora seja sintaticamente correto, pode levar a problemas, como ilustrado pela segunda linha. Nesse caso, agora **raw** referencia um objeto que contém um objeto **Integer**, mas a coerção assume que ele contém um **Double**. Esse erro não pode ser evitado no tempo de compilação. Em vez disso, causa erro de tempo de execução.

Devido ao risco inherente ao uso dos tipos brutos, **javac** exibe *avisos de não verificação* quando um tipo bruto é usado de uma maneira que possa ameaçar a segurança de tipos. No programa anterior, essas linhas geraram avisos de não verificação:

```
| Gen raw = new Gen(new Double(98.6));
|
| strOb = raw; // correto, mas pode provocar erros
```

Na primeira linha, é a chamada ao construtor de **Gen** sem um argumento de tipo que gera o aviso. Na segunda linha, é a atribuição de uma referência bruta a uma variável genérica que o gera.

À primeira vista, poderíamos achar que essa linha também deve gerar um aviso de não verificação, mas ela não gera:

```
| raw = iOb; // correto, mas pode provocar erros
```

Nenhum aviso do compilador é emitido, porque a atribuição não causa *mais* danos à segurança de tipos do que os já ocorridos quando **raw** foi criada.

Um último ponto: você deve limitar o uso de tipos brutos aos casos em que tiver de combinar código legado com código genérico mais recente. Os tipos brutos são apenas um recurso de transição e não algo que deva ser usado em código novo.

INFERÊNCIA DE TIPOS COM O OPERADOR LOSANGO

A partir do JDK 7, podemos encurtar a sintaxe usada na criação de uma instância de um tipo genérico. Para começar, lembremos da classe **TwoGen** mostrada anteriormente neste capítulo. Uma parte será mostrada aqui por conveniência. Observe que ela usa dois tipos genéricos.

```
class TwoGen<T, V> {
    T ob1;
    V ob2;

    // Passa para o construtor uma
    // referência a um objeto de tipo T.
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    // ...
}
```

Em versões de Java anteriores ao JDK 7, para criar uma instância de **TwoGen** temos de usar uma instrução semelhante a esta:

```
TwoGen<Integer, String> tgOb =
    new TwoGen<Integer, String>(42, "testing");
```

Aqui, os argumentos de tipo (que são **Integer** e **String**) são especificados duas vezes: primeiro, quando **tgOb** é declarada, e depois, quando uma instância de **TwoGen** é criada na chamada a seu construtor via **new**. Como os genéricos foram introduzidos pelo JDK 5, essa é a forma requerida por todas as versões de Java anteriores ao JDK 7. Embora não esteja errada, ela é um pouco mais prolixo do que precisaria ser. Como, na cláusula **new**, o tipo dos argumentos pode ser inferido facilmente, não há razão para que eles sejam especificados uma segunda vez. Para resolver essa situação, o JDK 7 adiciona um elemento sintático que nos permite evitar a segunda especificação.

No JDK 7, a declaração anterior pode ser reescrita assim

```
| TwoGen<Integer, String> tgOb = new TwoGen<>(42, "testing");
```

Observe que a parte que cria a instância usa simplesmente <>, que é uma lista de argumentos de tipo vazia. Isso se chama operador *losango*. Ele solicita ao compilador que infira os argumentos de tipo requeridos pelo construtor. A principal vantagem dessa sintaxe de inferência de tipos é que ela encura o que às vezes gera instruções de declaração muito longas. É particularmente útil para tipos genéricos que especificam limites.

O exemplo anterior pode ser generalizado. Quando a inferência de tipos é utilizada, a sintaxe para a declaração de criação de uma referência e de uma instância genéricas tem a forma geral a seguir:

```
nome-classe<lista-arg-tipo> nome-var = new nome-classe<>(lista-arg-cons);
```

Aqui, a lista de argumentos de tipo da cláusula **new** está vazia.

Embora seja mais usada em instruções de declaração, a inferência de tipos também pode ser aplicada à passagem de parâmetros. Por exemplo, se o método a seguir for adicionado a **TwoGen**:

```
boolean isSame(TwoGen<T, V> o) {
    if(o.b1 == o1.b1 && o.b2 == o1.b2) return true;
    else return false;
}
```

Então, a chamada abaixo será válida no JDK 7:

```
|if(tgOb.isSame(new TwoGen<>(42, "testing"))) System.out.println("Same");
```

Nesse caso, os argumentos de tipo passados para **isSame()** podem ser inferidos. Eles não precisam ser especificados novamente.

Já que o operador losango foi adicionado pelo JDK 7 e não funcionará com compiladores mais antigos, os outros exemplos de genéricos deste livro continuarão usando a sintaxe completa na declaração de instâncias de classes genéricas. Assim, funcionarão com qualquer compilador Java que dê suporte aos genéricos. O uso da sintaxe completa também deixa muito claro o que está sendo criado, o que é útil quando o exemplo de código é mostrado. É claro que, em um código seu, o uso da sintaxe de inferência de tipos otimizará as declarações.

Verificação do progresso

- Se uma interface genérica for implementada por uma classe, essa classe também deve ser genérica. Verdadeiro ou falso?
- Um parâmetro de tipo de uma interface genérica não pode ser limitado. Verdadeiro ou falso?
- Dado o código

```
|class XYZ<T> { // ...
```

mostre como declarar um objeto **ob** do tipo bruto **XYZ**.

- A inferência de tipos usa o operador _____.

Respostas:

- Verdadeiro.
- Falso.
- `XYZ ob = new XYZ();`
- losango

ERASURE

Geralmente, não é necessário o programador saber os detalhes de como o compilador Java transforma o código-fonte em código-objeto. No entanto, no caso dos genéricos, algum conhecimento geral do processo é importante, já que ele explica por que os recursos genéricos funcionam do modo como funcionam – e por que às vezes seu comportamento surpreende. Logo, é útil discutirmos brevemente como os genéricos são implementados em Java.

Uma restrição importante que conduziu a maneira de os genéricos serem adicionados à Java foi a necessidade de compatibilidade com versões anteriores da linguagem. Resumindo: o código genérico tinha que ser compatível com códigos não genéricos preexistentes. Logo, qualquer alteração na sintaxe da linguagem Java, ou na JVM, não poderia invalidar códigos mais antigos. A maneira de Java implementar os genéricos respeitando essa restrição é pela técnica *erasure*.

Em geral, *erasure* funciona assim: quando o código Java é compilado, todas as informações de tipos genéricos são removidas (em inglês, ‘erased’). Ou seja, é feita a substituição dos parâmetros de tipo por seu tipo limitado, que é **Object** quando nenhum limite explícito é especificado, e a aplicação das coerções apropriadas (como determinado pelos argumentos de tipo) para que seja mantida a compatibilidade com os tipos especificados pelos argumentos. O compilador também impõe a compatibilidade de tipos. Essa abordagem dos genéricos não permite que existam parâmetros de tipo no tempo de execução. Eles são simplesmente um mecanismo do código-fonte.

ERROS DE AMBIGUIDADE

A inclusão dos genéricos fez surgir um novo tipo de erro do qual você deve se proteger: a *ambiguidade*. Erros de ambiguidade ocorrem quando o erasure faz duas declarações genéricas aparentemente distintas produzirem o mesmo tipo, causando um conflito. Veja um exemplo que envolve a sobrecarga de métodos:

```
// Ambiguidade causada pelo erasure em
// métodos sobrecarregados.
class MyGenClass<T, V> {
    T ob1;
    V ob2;

    // ...

    // Estes dois métodos sobrecarregados são ambíguos
    // e não serão compilados.
    void set(T o) { ←
        ob1 = o;
    }
    void set(V o) { ←
        ob2 = o;
    }
}
```

Observe que **MyGenClass** declara dois tipos genéricos: **T** e **V**. Dentro de **MyGenClass**, é feita uma tentativa de sobrecarregar **set()** com base em parâmetros de tipo **T**

e V. Isso é considerado correto porque T e V parecem ser tipos diferentes. No entanto, há dois problemas de ambiguidade aqui.

Em primeiro lugar, do modo que MyGenClass foi criada, não é necessário que T e V sejam tipos diferentes. Por exemplo, é perfeitamente correto (em princípio) construir um objeto MyGenClass como mostrado abaixo:

```
| MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

Nesse caso, tanto T quanto V serão substituídos por String. Isso torna as duas versões de set() idênticas, o que, certamente, é um erro.

O segundo e mais grave problema é que a remoção de tipos de set() reduz as duas versões ao seguinte:

```
| void set(Object o) { // ...
```

Logo, a sobrecarga de set() como tentada em MyGenClass é inherentemente ambígua. A solução nesse caso é usar dois nomes de método distintos em vez de tentar sobrecarregar set().

Verificação do progresso

1. Erasure _____ todos os parâmetros de tipo, substituindo-os pelos tipos limitados e aplicando as coerções apropriadas.
2. Por padrão, o tipo limitado de um parâmetro de tipo é _____.
3. A ambiguidade pode ocorrer quando a remoção de tipos faz duas declarações aparentemente distintas produzirem o mesmo tipo. Verdadeiro ou falso?

ALGUMAS RESTRIÇÕES DOS GENÉRICOS

Há algumas restrições das quais você deve lembrar ao usar genéricos. Elas envolvem a criação de objetos de um parâmetro de tipo, membros estáticos, exceções e arrays. Todas serão examinadas aqui.

Parâmetros de tipos não podem ser instanciados

Não é possível criar uma instância de um parâmetro de tipo. Por exemplo, considere a classe a seguir:

```
// Não é possível criar uma instância de T.
class Gen<T> {
    T ob;
    Gen() {
        ob = new T(); // Inválido!!!
    }
}
```

Respostas:

1. remove
2. Object
3. Verdadeiro.

Aqui, não é válido tentar criar uma instância de **T**. A razão deve ser fácil de entender: o compilador não tem como saber que tipo de objeto criar. **T** é simplesmente um espaço reservado.

Restrições aos membros estáticos

Nenhum membro **static** pode usar um parâmetro de tipo declarado pela classe externa. Por exemplo, os dois membros **static** desta classe não são válidos:

```
class Wrong<T> {
    // Errado, não podemos ter variáveis estáticas de tipo T.
    static T ob;

    // Errado, nenhum método estático pode usar T.
    static T getob() {
        return ob;
    }
}
```

Embora você não possa declarar membros **static** que usem um parâmetro de tipo declarado pela classe que os contêm, *pode* declarar métodos genéricos **static**, que definem seus próprios parâmetros de tipo, como foi feito anteriormente neste capítulo.

Restrições aos arrays genéricos

Há duas restrições importantes dos genéricos aplicáveis aos arrays. Em primeiro lugar, você não pode instanciar um array cujo tipo do elemento seja um parâmetro de tipo. Em segundo lugar, não pode criar um array de referências genéricas específicas de um tipo. O programa a seguir mostra as duas situações:

```
// Genéricos e arrays.
class Gen<T extends Number> {
    T ob;

    T[] vals; // Correto

    Gen(T o, T[] nums) {
        ob = o;

        // Esta instrução não é válida.
        // vals = new T[10]; // não pode criar um array de tipo T

        // Mas esta instrução está correta.
        vals = nums; // É correto atribuir referências de um array existente
    }
}

class GenArrays {
    public static void main(String[] args) {
        Integer[] n = { 1, 2, 3, 4, 5 };
```

```
Gen<Integer> iOb = new Gen<Integer>(50, n);

// Não pode criar um array de referências genéricas específicas
// de um tipo.
// Gen<Integer>[] gens = new Gen<Integer>[10]; // Errado!

// Isto é correto.
Gen<?>[] gens = new Gen<?>[10]; // Correto
}
```

Como o programa mostra, é válido declarar uma referência a um array de tipo T, como esta linha faz:

```
| T[] vals; // Correto
```

Mas você não pode instanciar um array de tipo T, como esta linha desativada por comentário tenta:

```
| // vals = new T[10]; // não pode criar um array de tipo T
```

Não podemos criar um array de tipo T porque não há como o compilador saber que tipo de array deve ser realmente criado. No entanto, podemos passar para **Gen()** uma referência a um array de tipo compatível quando um objeto for criado e atribuir essa referência a **vals**, como o programa faz nesta linha:

```
| vals = nums; // Correto atribuir referência a array existente
```

Isso funciona porque o array passado para **Gen()** tem um tipo conhecido, que será o mesmo tipo de T no momento de criação do objeto. Dentro de **main()**, observe que você não pode declarar um array de referências a um tipo genérico específico. Isto é, a linha a seguir

```
| // Gen<Integer>[] gens = new Gen<Integer>[10]; // Errado!
```

não será compilada.

Restrições a exceções genéricas

Uma classe genérica não pode estender **Throwable**. Ou seja, você não pode criar classes de exceção genéricas.

EXERCÍCIOS

1. Os genéricos são importantes para Java porque permitem a criação de código
 - A. com segurança de tipos
 - B. reutilizável
 - C. confiável
 - D. todas as alternativas acima
2. Um tipo primitivo pode ser usado como argumento de tipo?

3. Mostre como declarar uma classe chamada **FlightSched** que use dois parâmetros genéricos.
4. Usando a resposta à Questão 3, altere o segundo parâmetro de tipo de **FlightSched** para que seja preciso estender **Thread**.
5. Agora, altere **FlightSched** para que seu segundo parâmetro de tipo seja subclasse do primeiro parâmetro de tipo.
6. No que diz respeito aos genéricos, o que é o símbolo ? e o que ele faz?
7. O argumento curinga pode ser limitado?
8. Um método genérico chamado **MyGen()** tem um parâmetro de tipo. Além disso, **MyGen()** tem um parâmetro cujo tipo é o do parâmetro de tipo. Ele também retorna um objeto desse parâmetro de tipo. Mostre como declarar **MyGen()**.
9. Dada a interface genérica

```
| interface IGenIF<T, V extends T> { // ...
```

mostre a declaração de uma classe chamada **MyClass** que implemente **IGenIF**.

10. Dada uma classe genérica chamada **Counter<T>**, mostre como criar um objeto de seu tipo bruto.
11. Existem parâmetros de tipo no tempo de execução?
12. Quando uma classe genérica é herdada, seus parâmetros de tipo também devem ser especificados pela subclasse. Verdadeiro ou falso?
13. O que é <>?
14. Com o uso do JDK 7, como a linha a seguir pode ser simplificada?

```
| MyClass<Double, String> obj = new MyClass<Double, String>(1.1, "Hi");
```

15. Quais das linhas de código abaixo são declarações válidas de métodos genéricos? Se alguma não for válida, explique por quê.
 - A. void print<T>(T x) { System.out.println(x); }
 - B. static <T> void print(T x) { System.out.println(x); }
 - C. <T> T getT(T t) { return null; }
 - D. <?> void print(Object x) { System.out.println(x); }
 - E. <V extends T> void print(V x) { System.out.println(x); }
 - F. <T extends Object> void print(T x) { System.out.println(x); }

16. No corpo de um método genérico com parâmetro de tipo **T**, você não pode usar a instrução de atribuição

```
| T[] x = new T[10];
```

porque não pode criar arrays com o tipo **T**. Funcionaria inicializar **x** da forma a seguir?

```
| T[] x = (T[]) new Object[10];
```

17. Neste capítulo, foi mencionado que uma classe não genérica pode ter uma subclasse genérica. O oposto também é verdadeiro? Isto é, uma classe genérica pode ter uma subclasse não genérica?

18. Neste capítulo, foi mencionado que uma classe não genérica pode ter um método genérico. O oposto também é verdadeiro? Isto é, uma classe genérica pode ter um método não genérico?
19. É válido criar uma classe genérica com parâmetro de tipo **T** e nunca usar **T** na implementação da classe? Por exemplo, a definição de classe a seguir é válida?

```
class MyClass<T> {
    MyClass() { }
    void printName() { System.out.println("MyClass"); }
}
```

20. Na seção Tente isto 14-1, vimos como criar uma classe genérica de pilha simples. Talvez você tenha interesse em saber que a biblioteca Java já inclui uma classe de pilha completa chamada **Stack**, que fica no pacote **java.util**. Nas versões de Java anteriores aos genéricos, a classe **Stack** armazenava qualquer tipo de objeto. Seu método **push()** usava **Object** como o tipo do parâmetro e o tipo de retorno de seu método **pop()** também era **Object**. Para usar essa pilha para armazenar strings, você teria que executar a coerção de tipos. Por exemplo, após inserir um string, só poderia recuperá-lo como um string com instruções na forma

```
| String value = (String) stack.pop();
```

Quais são as desvantagens de fazê-lo dessa maneira? Isto é, por que a classe **Stack** genérica é melhor?

21. Sendo **Obj<T>** uma classe genérica com um construtor sem argumentos, quais são as diferenças, se houver alguma, entre as quatro instruções abaixo?
- A. **Obj x = new Obj<Object>();**
 - B. **Obj<Object> x = new Obj<Object>();**
 - C. **Obj<Object> x = new Obj<>();**
 - D. **Obj<Object> x = new Obj();**
22. A seguir temos uma tentativa de tornar a classificação de bolha um método genérico para que possa classificar todos os tipos de arrays em vez de apenas arrays de inteiros, como foi feito no Capítulo 5. No entanto, ela não será compilada. Descubra o que não é válido no método.

```
<T> void bubbleSort(T[] data) {
    for(int a=1; a < data.length; a++)
        for(int b=data.length-1; b >= a; b--) {
            if(data[b-1] > data[b]) { // se fora de ordem
                // troca valores
                T x = data[b-1];
                data[b-1] = data[b];
                data[b] = x;
            }
        }
}
```

23. Considere as classes **C** e **CDemo** mostradas aqui.

```
class C<T> {
    T data;
    C(T t) { data = t; }
}

class CDemo {
    public static void main(String[] args) {
        C<Object> co = new C<String>("Hi");
        C<Integer> ci = new C<33>(44);
        C<int> cint = new C<>(3);
        C<String> cs = new C<>("Hi");
        String[] s = {"a", "b", "c"};
        C<String[]> csa = new C<>(s);
    }
}
```

Quais das instruções do método **main()** de **CDemo** são inválidas? Explique o que há de errado com as instruções inválidas. Considere que está usando o JDK 7.

24. As classes a seguir, uma genérica e a outra não, podem ser usadas na criação de um objeto que armazene um **Number**. Quais são as vantagens, se houver alguma, de uma classe sobre a outra?

```
class C<T extends Number> {
    T data;

    C(T t) { data = t; }

    T getData() { return data; }
}

class C {
    Number data;

    C(Number t) { data = t; }

    Number getData() { return data; }
}
```

25. Implemente um método genérico chamado **containsNull()** que use um array de tipo **T[]** como parâmetro e retorne **true** se algum dos valores do array for **null**. Quais são as vantagens, se houver alguma, desse método sobre uma versão não genérica dele que você possa implementar substituindo **T** em qualquer local por **Object**?

26. Se **A** é superclasse de **B** e **C** é superclasse de **D**, quais das opções abaixo são verdadeiras?

- A. **A<C>** é superclasse de **B<C>**
- B. **A<C>** é superclasse de **A<D>**
- C. **A<C>** é superclasse de **B<D>**

27. A definição de classe a seguir não será compilada. Por quê?

```
class Keeper<T> {
    T t;

    Keeper(T t) {
        this.t = t;
    }

    public boolean equals(T other) {
        return other == t;
    }
}
```

28. Suponhamos que uma classe **B** fosse subclasse da classe **A**. Suponhamos também que **A** tivesse o método abaixo

```
| <T> void getX(T t) { ... }
```

e a classe **B** tivesse este método

```
| <V> void getX(V v) { ... }
```

O método **getX()** é sobreescrito em **B** ou o método **getX()** de **B** sobrepõe o **getX()** herdado?

29. Dê um exemplo em que um parâmetro de tipo curinga **<? super T>** para o limite inferior seja útil.

Applets e as outras palavras-chave Java

PRINCIPAIS HABILIDADES E CONCEITOS

- Entender os aspectos básicos dos applets
- Criar um esqueleto de applet
- Inicializar e encerrar applets
- Atualizar applets
- Exibir informações na janela de status
- Passar parâmetros para um applet
- Conhecer as outras palavras-chave Java: **transient**, **volatile**, **instanceof**, **native**, **strictfp** e **assert**.

Este capítulo termina nosso estudo dos elementos principais da linguagem Java. Ele começa descrevendo o applet, que foi uma inovação de Java. O applet ajudou a moldar a Internet em seus primórdios e ainda está em uso atualmente. Os applets estão tão intimamente associados a Java que nenhum curso de programação em Java estaria completo sem uma introdução a eles. O capítulo termina com uma descrição das palavras-chave Java restantes, como **instanceof** e **native**, que ainda não foram descritas neste livro.

ASPECTOS BÁSICOS DOS APPLETS

Os applets diferem dos tipos de programa mostrados nos capítulos anteriores. Como explicado no Capítulo 1, eles são programas pequenos projetados para transmissão pela Internet e execução dentro de um navegador. Já que a máquina virtual Java se encarrega da execução de todos os programas Java, inclusive applets, os applets oferecem uma maneira segura de baixar e executar dinamicamente programas pela Web.

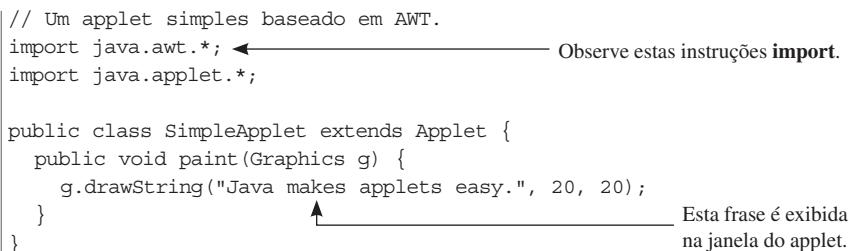
Antes de começarmos, é necessário explicar que há dois tipos gerais de applets: os baseados apenas no Abstract Window Toolkit (AWT) e os baseados em Swing. Ambos dão suporte à criação de uma interface gráfica de usuário (GUI). AWT é o kit de ferramentas de GUI original e Swing é a alternativa leve e moderna de Java. Este capítulo usou applets baseados em AWT para introduzir os fundamentos da programação de applets porque eles são os mais fáceis de descrever. Posteriormente, na

Parte II, Swing e applets baseados em Swing serão abordados. É importante entender, no entanto, que os applets baseados em Swing usam a mesma arquitetura dos applets baseados em AWT. Além disso, Swing fica acima de AWT. Logo, as informações e técnicas apresentadas aqui descrevem a base da programação de applets e grande parte dela se aplica aos dois tipos de applets.

Examinaremos um applet simples antes de discutir teorias ou detalhes. Ele executa uma tarefa: exibe o string “Java makes applets easy” dentro de uma janela.

```
// Um applet simples baseado em AWT.
import java.awt.*; ← Observe estas instruções import.
import java.applet.*;

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Java makes applets easy.", 20, 20);
    }
}
```



↑ Esta frase é exibida na janela do applet.

O applet começa com duas instruções **import**. A primeira importa as classes do AWT. Os applets interagem com o usuário (direta ou indiretamente) através do AWT e não por classes de I/O baseadas no console. O AWT dá suporte a uma interface gráfica de usuário baseada em janela. Como era de se esperar, ele é grande e sofisticado e uma discussão detalhada não faz parte do escopo deste livro. Felizmente, já que criaremos apenas applets muito simples, usaremos o AWT de maneira limitada. A outra instrução **import** importa o pacote **java.applet**. Esse pacote contém a classe **Applet**. Qualquer applet que criarmos deve ser (direta ou indiretamente) subclasse de **Applet**.

A próxima linha do programa declara a classe **SimpleApplet**. Essa classe deve ser declarada como **public**, porque será acessada por código externo.

Dentro de **SimpleApplet**, **paint()** é declarado. Esse método faz parte da classe **Component** (que é superclasse de **Applet**) do AWT e é sobreposto pelo applet. A classe **Component** define a funcionalidade básica comum a todos os componentes de GUI. É uma classe muito grande, mas a única parte dela que teremos de usar aqui é **paint()**. O método **paint()** é chamado sempre que o applet tem de exibir sua saída. Isso pode ocorrer por várias razões. Por exemplo, a janela em que o applet está sendo executado pode ser sobreposta por outra janela e depois aparecer novamente. Ou ainda, a janela do applet pode ser minimizada e então restaurada. O método também é chamado quando o applet começa a ser executado. Qualquer que seja a causa, sempre que o applet tiver que exibir sua saída, **paint()** será chamado. O método **paint()** tem um parâmetro de tipo **Graphics**. (**Graphics** é outra classe do AWT.) Esse parâmetro contém o *contexto gráfico*, que descreve o ambiente gráfico em que o applet está sendo executado. O contexto é usado sempre que a saída do applet é requerida.

Nota: O método **paint()** só é usado por applets baseados em AWT para exibir a saída em uma janela. Os applets Swing usam um mecanismo diferente, que é descrito no Capítulo 21.

Dentro de **paint()**, há uma chamada a **drawString()**, que é membro da classe **Graphics**. Esse método exibe um string começando no local X,Y especificado. Ele tem a forma geral a seguir:

```
void drawString(String mensagem, int x, int y)
```

Aqui, *mensagem* é o string a ser exibido começando em *x,y*. Em uma janela Java, o canto superior esquerdo é o local 0,0. A chamada a **drawString()** no applet faz a mensagem ser exibida começando no local 20,20.

Observe que o applet não tem um método **main()**. Diferentemente dos programas mostrados anteriormente neste livro, os applets não começam a ser executados em **main()**. Na verdade, a maioria dos applets sequer tem um método **main()**. Em vez disso, um applet começa a ser executado quando o nome de sua classe é passado para um navegador ou outro programa habilitado para a execução de applets.

Após você ter digitado o código-fonte de **SimpleApplet**, poderá fazer a compilação da mesma forma que compilaria outros programas. No entanto, a execução de **SimpleApplet** envolve um processo diferente. Há duas maneiras pelas quais você pode executar um applet: dentro de um navegador ou com uma ferramenta de desenvolvimento especial que exiba applets. A ferramenta fornecida com o JDK Java padrão se chama **appletviewer**; nós a usaremos para executar os applets desenvolvidos neste capítulo. Certamente você também pode executá-los em um navegador, mas o **appletviewer** é muito mais fácil de usar durante o desenvolvimento.

Uma maneira de executar um applet (em um navegador Web ou no **appletviewer**) é criar um pequeno arquivo de texto HTML contendo uma tag que o carregue. Atualmente, a Oracle recomenda o uso da tag APPLET para esse fim. (A tag OBJECT também pode ser usada e há outras estratégias de implantação disponíveis. Consulte a documentação Java para ver as informações mais recentes.) Com o uso da tag APPLET, este é o arquivo HTML que executará **SimpleApplet**:

```
|<applet code="SimpleApplet" width=200 height=60>
|</applet>
```

As instruções **width** e **height** especificam as dimensões da área de exibição usada pelo applet.

Para executar **SimpleApplet** com o visualizador de applets, você terá que executar esse arquivo HTML. Por exemplo, se o arquivo HTML anterior se chamassem **StartApp.html**, a linha de comando a seguir executaria **SimpleApplet**:

```
|appletviewer StartApp.html
```

Embora não seja errado usar um arquivo HTML autônomo para a execução de um applet, há uma maneira mais fácil quando se usa o **appletviewer**: apenas inclua um comentário contendo a tag APPLET perto do início do arquivo de código-fonte de seu applet. Se você usar esse método, o arquivo-fonte de **SimpleApplet** ficará com a aparência a seguir:

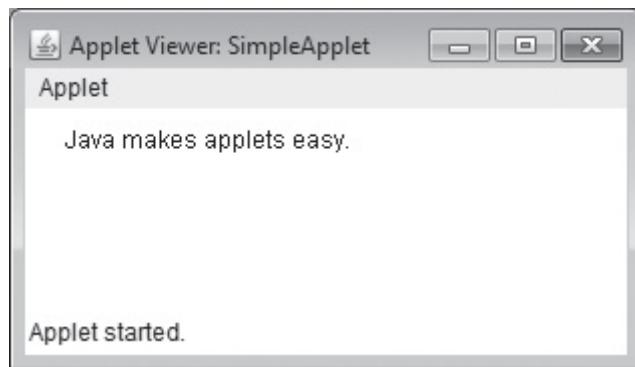
```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Java makes applets easy.", 20, 20);
    }
}
```

Este HTML é usado pelo **appletviewer**
para execução do applet.

Agora, você pode executar o applet passando o nome do arquivo-fonte para o **appletviewer**. Por exemplo, esta linha de comando exibirá **SimpleApplet**:

```
| appletviewer SimpleApplet.java
```

A janela produzida por **SimpleApplet**, como exibida pelo **appletviewer**, pode ser vista na ilustração a seguir:



Ao usar o **appletviewer**, lembre-se de que ele fornece a moldura da janela. Applets executados em um navegador não terão uma moldura visível.

Recapitulemos os pontos-chave do applet:

- Todos os applets são, direta ou indiretamente, subclasses de **Applet**.
- Os applets não precisam de um método **main()**.
- Os applets devem ser executados em um visualizador de applets ou em um navegador compatível com Java.
- I/O do usuário não é executado com as classes Java de I/O de fluxo. Em vez disso, os applets usam a interface fornecida pelo AWT (ou por Swing).

Verificação do progresso

1. O que é um applet?
2. O que **paint()** faz?
3. Que pacote deve ser incluído quando criamos um applet?
4. Como os applets são executados?

Respostas:

1. Um applet é um tipo especial de programa Java projetado para transmissão pela Internet e que é executado dentro de um navegador.
2. O método **paint()** exibe a saída em uma janela de applet baseado no AWT.
3. O pacote **java.applet** deve ser incluído quando criamos um applet.
4. Os applets são executados por um navegador ou por ferramentas especiais, como o **appletviewer**.

ESQUELETO DE APPLET COMPLETO

Apesar de a classe **SimpleApplet** mostrada anteriormente ser um applet, ela não contém todos os elementos requeridos pela maioria dos applets. Na verdade, todos os applets, exceto os mais simples, sobrepõem um conjunto de métodos que fornecem o mecanismo básico pelo qual o navegador ou o visualizador de applets interage com o applet e controla sua execução. Esses métodos de ciclo de vida são **init()**, **start()**, **stop()** e **destroy()**, e são definidos por **Applet**. Normalmente um quinto método, **paint()**, é sobreposto por applets baseados no AWT, embora não seja um método de ciclo de vida. Como mencionado, ele é herdado da classe **Component** do AWT. Já que implementações padrão de todos esses métodos são fornecidas, os applets não precisam sobrepor os métodos que não usam. Esses quatro métodos de ciclo de vida, mais **paint()**, foram reunidos no esboço abaixo:

```
// Esboço de applet baseado no AWT.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
    // Chamado primeiro.
    public void init() {
        // inicialização
    }

    /* Segundo a ser chamado, após init(). Também é chamado sempre
       que o applet é reiniciado. */
    public void start() {
        // inicia ou retoma a execução
    }

    // Chamado quando o applet é interrompido.
    public void stop() {
        // suspende a execução
    }

    /* Chamado quando o applet é encerrado. Este é o último
       método executado. */
    public void destroy() {
        // executa atividades de encerramento
    }

    // Chamado quando a janela de um applet baseado no AWT deve ser restaurada.
    public void paint(Graphics g) {
        // volta a exibir o conteúdo da janela
    }
}
```

Embora esse esboço não faça coisa alguma, ele pode ser compilado e executado.

*Nota: A sobreposição de **paint()** quase sempre ocorre em applets baseados no AWT. Applets Swing usam um mecanismo de atualização diferente.*

INICIALIZAÇÃO E ENCERRAMENTO DO APPLET

É importante saber a ordem em que os diversos métodos mostrados no esboço de applet anterior são executados. Quando um applet é iniciado, os métodos a seguir são chamados nessa sequência:

1. **init()**
2. **start()**
3. **paint()**

Quando um applet é encerrado, ocorre a seguinte sequência de chamadas de método:

1. **stop()**
2. **destroy()**

Examinemos esses métodos com mais detalhes.

O método **init()** é o primeiro a ser chamado. Em **init()** o applet inicializará variáveis e executará qualquer outra atividade de inicialização.

O método **start()** é chamado após **init()**. Também é chamado para reiniciar um applet depois de ele ter sido interrompido, como quando o usuário retorna a uma página Web já exibida que contém um applet. Logo, **start()** pode ser chamado mais de uma vez durante o ciclo de vida de um applet.

O método **paint()** já foi descrito e é chamado sempre que a saída de um applet baseado no AWT tem que ser exibida novamente.

Quando a página que contém o applet é abandonada, o método **stop()** é chamado. Você usará **stop()** para suspender qualquer thread filha criada pelo applet e para executar outras atividades necessárias à inserção do applet em um estado ocioso seguindo. Lembre-se, uma chamada a **stop()** não significa que o applet será encerrado, porque ele pode ser reiniciado com uma chamada a **start()** se o usuário voltar à página.

O método **destroy()** é chamado quando o applet não é mais necessário. Ele é usado para executar qualquer operação de encerramento requerida pelo applet.

ASPECTO-CHAVE DA ARQUITETURA DE UM APPLET

Como o esboço de applet deixou claro, a arquitetura de um applet é diferente dos programas de console mostrados na primeira parte deste livro. Uma das diferenças-chave é que um applet espera o sistema de tempo de execução chamar um de seus métodos. Por exemplo, ele só exibe saídas se seu método **paint()** for chamado. Uma vez que um método definido pelo applet for chamado, o applet deve tomar as medidas apropriadas e então retornar o controle rapidamente para o sistema. Esse é um ponto crucial. Geralmente, o applet não deve entrar em um “modo” de operação em que mantenha o controle por um período extenso. Isso

poderia fazer outras partes do applet pararem de responder. Em situações em que o applet tiver que executar uma tarefa repetitiva por conta própria (por exemplo, exibindo a rolagem de uma mensagem em sua janela), você deve iniciar uma thread de execução adicional.

Verificação do progresso

1. Quais são os quatro métodos de ciclo de vida que a maioria dos applets sobrepõe?
2. O que um applet deve fazer quando **start()** for chamado?
3. O que um applet deve fazer quando **stop()** for chamado?

SOLICITANDO ATUALIZAÇÃO

Como regra geral, um applet baseado no AWT só exibe saídas em sua janela quando seu método **paint()** é chamado pelo sistema de tempo de execução. Isso levanta uma questão interessante: como o applet pode fazer sua janela ser atualizada quando as informações mudam? Por exemplo, se um applet está exibindo um banner móvel, que mecanismo ele usa para atualizar a janela sempre que o banner rola? Como explicado, uma das restrições básicas de arquitetura impostas a um applet é que ele deve devolver o controle rapidamente para o sistema Java de tempo de execução. Ele não pode criar um laço dentro de **paint()** que role repetidamente o banner, por exemplo. Isso impediria que o controle passasse novamente para o sistema de tempo de execução. Dada essa restrição, parece que, na melhor das hipóteses, será difícil exibir saídas na janela do applet. Felizmente, não é esse o caso. Sempre que um applet tem que atualizar as informações exibidas em sua janela, ele apenas chama **repaint()**.

O método **repaint()** é definido pela classe **Component** do AWT e herdado por **Applet**. Ele faz o sistema de execução chamar o método **paint()** do applet. Logo, para que outra parte de um applet exiba saídas na janela, simplesmente armazene a saída e então chame **repaint()**. Isso acionará uma chamada ao método **paint()**, que poderá exibir as informações armazenadas. Por exemplo, se parte do applet tiver que exibir um string, poderá armazená-lo em uma variável **String** e então chamar **repaint()**. Dentro de **paint()**, você exibirá o string usando **drawString()**.

A versão mais simples de **repaint()** é a mostrada abaixo:

```
void repaint()
```

Essa versão faz a janela inteira ser atualizada. Um exemplo que demonstra **repaint()** pode ser visto na seção Tente isto 15-1.

Respostas:

1. Os métodos de ciclo de vida são **init()**, **start()**, **stop()** e **destroy()**.
2. Quando **start()** for chamado, o applet deve ser iniciado ou reiniciado.
3. Quando **stop()** for chamado, o applet deve ser pausado.

Pergunte ao especialista

P É possível um método que não seja **paint()** exibir saídas na janela de um applet?

R Sim. Para fazê-lo, você deve obter um contexto gráfico chamando **getGraphics()** na instância do applet e então usá-lo para exibir saídas na janela. No entanto, na maioria dos aplicativos baseados no AWT, é melhor e mais fácil exibir a saída com **paint()** e chamar **repaint()** quando o conteúdo mudar.

TENTE ISTO 15-1 Applet de banner simples

`Banner.java`

Apresentaremos um applet de banner simples para demonstrar **repaint()**. Esse applet rola uma mensagem, da direita para a esquerda, ao longo de sua janela. Uma vez que a rolagem da mensagem é uma tarefa repetitiva, ela é executada por uma thread separada, criada pelo applet quando este é iniciado. Ele também mostra uma maneira pela qual uma thread separada pode ser usada para executar uma tarefa contínua em um applet.

PASSO A PASSO

1. Crie um arquivo chamado **Banner.java**.
2. Comece a criação do applet de banner com as linhas a seguir:

```
/*
Tente isto 15-1

Applet de banner simples.

Este applet cria uma thread que rola
a mensagem contida em msg da direita para a esquerda
ao longo de sua janela.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="Banner" width=300 height=50>
</applet>
*/

public class Banner extends Applet implements Runnable {
    String msg = " Java Rules the Web ";
    Thread t;
    boolean stopFlag;

    // Inicializa t com null.
    public void init() {
        t = null;
    }
}
```

Observe que **Banner** estende **Applet**, como esperado, mas também implementa **Runnable**. Isso é necessário porque o applet criará uma segunda thread de execução que será usada para rolar o banner. A mensagem que será rolada no banner está contida na variável **String msg**. Uma referência à thread que executa o applet está armazenada em **t**. A variável booleana **stopFlag** é usada para interromper o applet. Dentro de **init()**, a variável de referência de thread **t** é configurada com **null**.

3. Adicione o método **start()** mostrado a seguir:

```
// Inicia a thread quando precisamos do applet.
public void start() {
    t = new Thread(this);
    stopFlag = false;
    t.start();
}
```

O sistema de tempo de execução chama **start()** para iniciar a execução do applet. Dentro de **start()**, uma nova thread de execução é criada e atribuída à variável **Thread t**. A variável **stopFlag** é então configurada com **false** e, em seguida, a thread é iniciada por uma chamada a **t.start()**. Lembre-se de que **t.start()** é uma chamada a um método definido por **Thread**, que faz **run()** ser executado. Ela não aciona uma chamada à versão de **start()** definida por **Applet**. São dois métodos diferentes.

4. Adicione o método **run()**, como mostrado aqui:

```
// Ponto de entrada da thread que executa o banner.
public void run() {
    // Solicita uma atualização a cada quarto de segundo.
    for( ; ; ) {
        try {
            repaint();
            Thread.sleep(250);
            if(stopFlag) break;
        } catch(InterruptedException exc) {}
    }
}
```

Em **run()**, **repaint()** é chamado repetidamente, com um retardo de um quarto de segundo entre as chamadas. Cada chamada a **repaint()** faz o método **paint()** ser chamado. A variável **stopFlag** é verificada a cada iteração. Quando ela é igual a **true**, o método **run()** é encerrado.

5. Adicione o código de **stop()** como mostrado abaixo:

```
// Pausa o banner.
public void stop() {
    stopFlag = true;
    t = null;
}
```

Se um navegador estiver exibindo o applet quando uma nova página for visualizada, o método **stop()** será chamado para configurar **stopFlag** com **true** e fazer **run()** ser encerrado. Ele também configura **t** com **null**. Logo, não há mais uma referência ao objeto **Thread** e ele pode ser reciclado na próxima vez que o coletor de lixo for executado. Esse é o mecanismo usado para encerrar a thread quando sua página não está mais sendo exibida. Quando o applet é exibido novamente, **start()** é chamado mais uma vez, iniciando uma nova thread para executar o banner.

6. Para concluir, adicione o método **paint()** mostrado aqui:

```
// Exibe o banner.
public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;

    g.drawString(msg, 50, 30);
}
```

Dentro de **paint()**, os caracteres do string contido em **msg** são girados para a esquerda e então exibidos. Já que **paint()** será executado a cada quarto de segundo, o resultado final é que o conteúdo de **msg** será rolado da direita para a esquerda em uma exibição móvel constante.

7. O applet de banner inteiro é mostrado aqui:

```
/*
Tente isto 15-1

Applet de banner simples.

Este applet cria uma thread que rola
a mensagem contida em msg da direita para a esquerda
ao longo de sua janela.

*/
import java.awt.*;
import java.applet.*;
/*
<applet code="Banner" width=300 height=50>
</applet>
*/

public class Banner extends Applet implements Runnable {
    String msg = " Java Rules the Web ";
    Thread t;
    boolean stopFlag;

    // Inicializa t com null.
    public void init() {
        t = null;
```

```

        }

    // Inicia a thread quando precisamos do applet.
    public void start() {
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }

    // Ponto de entrada da thread que executa o banner.
    public void run() {
        // Solicita uma nova atualização a cada quarto de
        segundo.
        for( ; ; ) {
            try {
                repaint();
                Thread.sleep(250);
                if(stopFlag) break;
            } catch(InterruptedException exc) {}
        }
    }

    // Pausa o banner.
    public void stop() {
        stopFlag = true;
        t = null;
    }

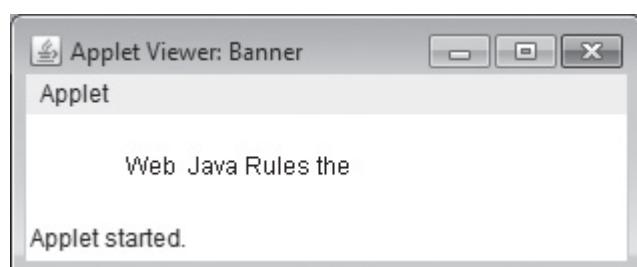
    // Exibe o banner.
    public void paint(Graphics g) {
        char ch;

        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;

        g.drawString(msg, 50, 30);
    }
}
}

```

Um exemplo da saída é mostrado abaixo:



8. Quando você executar o applet, pode notar que a rolagem apresenta uma oscilação ocasional. Você pode eliminar essa oscilação usando uma técnica chamada *buffer duplo*. Com o uso do buffer duplo, seu programa criará um segundo contexto gráfico para você preparar a saída. Em seguida, a saída completa será exibida na tela de uma só vez, o que eliminará a oscilação. O termo *buffer duplo* vem do fato de a tela ser considerada um buffer para pixels e, portanto, a imagem fora da tela seria um segundo buffer. As técnicas necessárias à implementação de um buffer duplo não fazem parte do escopo deste capítulo. Além disso, quando Swing é usado, o buffer duplo é implementado automaticamente, logo, não é algo que você mesmo terá que implementar. (Essa é uma das vantagens de Swing.) No entanto, como desafio extra, você pode fazer uma pesquisa e tentar adicionar o buffer duplo a esse exemplo por conta própria.

USANDO A JANELA DE STATUS

Além de exibir informações em sua janela, o applet também pode exibir uma mensagem na janela de status do navegador ou visualizador de applets em que estiver sendo executado. Para fazê-lo, chame **showStatus()**, que é definido por **Applet**, com o string que deseja exibir. A forma geral de **showStatus()** é a seguinte:

```
void showStatus(String msg)
```

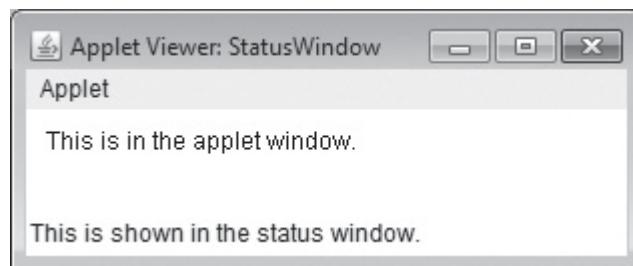
Aqui, *msg* é o string a ser exibido.

A janela de status é um bom local para darmos ao usuário um retorno sobre o que está ocorrendo no applet, sugerirmos opções ou possivelmente relatarmos alguns tipos de erros. Também é de grande ajuda na depuração, porque fornece um meio fácil de exibirmos informações sobre o applet.

O applet a seguir demonstra **showStatus()**:

```
// Usando a janela de status.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet{
    // Exibe a mensagem na janela do applet.
    public void paint(Graphics g) {
        g.drawString("This is in the applet window.", 10, 20);
        showStatus("This is shown in the status window.");
    }
}
```

Um exemplo da saída do programa é mostrado abaixo:



PASSANDO PARÂMETROS PARA APPLETS

Você pode passar parâmetros para um applet. Na verdade, isso é muito comum. Com frequência um parâmetro especifica alguma configuração ou atributo associado ao applet. Por exemplo, voltando ao applet **Banner** mostrado na seção Tente isto 15-1, você poderia controlar a velocidade de rolagem passando o retardo em vez de embutí-lo no código do applet. Também poderia passar a mensagem a ser exibida.

Para passar um parâmetro para um applet, use o atributo PARAM da tag APPLET, especificando o nome e o valor do parâmetro. Para recuperar um parâmetro, use o método **getParameter()**, definido por **Applet**. Sua forma geral é a seguinte:

```
String getParameter(String nomeParam)
```

Aqui, *nomeParam* é o nome do parâmetro. O método retorna o valor do parâmetro na forma de um objeto **String**. Logo, no caso de valores numéricos e **boolean**, você terá que converter o string em seus formatos binários. (Uma maneira de fazer isso é usar um método **parse** definido por um encapsulador de tipo, como **Integer.parseInt()**.) Se o parâmetro especificado não puder ser encontrado, **null** será retornado. Portanto, certifique-se de confirmar se o valor retornado por **getParameter()** é válido. Além disso, verifique parâmetros convertidos em um valor numérico, confirmando se ocorreu uma conversão válida.

Veja um exemplo que demonstra a passagem de parâmetros:

```
// Passa um parâmetro para um applet.
import java.awt.*;
import java.applet.*;

/*
<applet code="Param" width=300 height=80>
<param name=author value="Herb and Dale"> _____
<param name=purpose value="Demonstrate Parameters"> _____
<param name=version value=2> _____
</applet>
*/
public class Param extends Applet {
    String author;
```

Estes parâmetros
HTML são passados
para o applet.

```

String purpose;
int ver;

public void start() {
    String temp;

    author = getParameter("author");
    if(author == null) author = "not found";

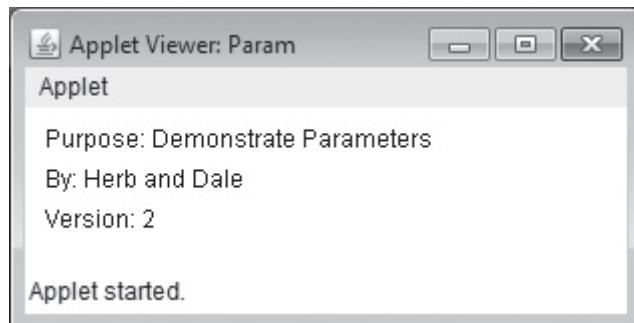
    purpose = getParameter("purpose");
    if(purpose == null) purpose = "not found"; ← É importante verificar se
                                                o parâmetro existe!

    temp = getParameter("version");
    try {
        if(temp != null)
            ver = Integer.parseInt(temp);
        else
            ver = 0;
    } catch(NumberFormatException exc) { ← Também é importante verificar se
                                                as conversões numéricas foram
                                                bem-sucedidas.
        ver = -1; // código de erro
    }
}

public void paint(Graphics g) {
    g.drawString("Purpose: " + purpose, 10, 20);
    g.drawString("By: " + author, 10, 40);
    g.drawString("Version: " + ver, 10, 60);
}
}

```

Um exemplo da saída desse programa é mostrado a seguir:



Verificação do progresso

1. Como podemos fazer o método **paint()** de um applet ser chamado?
2. Onde **showStatus()** exibe um string?
3. Que método é usado na obtenção de um parâmetro especificado na tag APPLET?

Pergunte ao especialista

P Além do que já foi discutido, há mais recursos suportados por **Applet**?

R Sim. Além dos métodos descritos nas seções anteriores, **Applet** contém vários outros. Por exemplo, você pode carregar uma imagem gráfica usando **getImage()**. Para carregar um clipe de áudio, use **getAudioClip()**, e para reproduzi-lo, use **play()**. Além disso, **Applet** herda uma parcela significativa da funcionalidade suportada pelo AWT. Como mencionado, **Applet** estende a classe **Component**, que contém mais de 100 métodos. Também herda **Container** e **Panel**. Essas são mais duas classes do AWT. **Container** especifica os recursos de um componente que será usado para conter (isto é, armazenar) outros componentes. **Panel** é um contêiner simples. Se quiser, examine essas classes na documentação Java. Isso lhe dará uma ideia da rica funcionalidade que ancora o suporte Java a interfaces gráficas de usuário.

AS OUTRAS PALAVRAS-CHAVE JAVA

Os capítulos anteriores descreveram a maioria das palavras-chave definidas por Java, mas ainda faltam algumas. Elas são mostradas aqui:

- **volatile**
- **transient**
- **instanceof**
- **strictfp**
- **assert**
- **native**

Essas palavras-chave atendem necessidades especializadas e não são usadas neste livro. No entanto, para darmos uma visão completa, todas serão descritas aqui.

Respostas:

1. Para fazer **paint()** ser chamado, chame **repaint()**.
2. O método **showStatus()** exibe a saída na janela de status.
3. Para obter um parâmetro, chame **getParameter()**.

Modificador volatile

O modificador **volatile** informa ao compilador que uma variável pode ser alterada inesperadamente por outra thread. Em um programa com várias threads, às vezes duas ou mais threads compartilham a mesma variável. A título de eficiência, cada thread pode manter sua própria cópia da variável compartilhada, possivelmente em um registro da CPU. A cópia real (ou *mestra*) da variável é atualizada em vários momentos, como quando um método **synchronized** é alcançado. Embora essa abordagem funcione bem, em alguns casos, tudo o que importa é que a cópia mestra de uma variável reflita sempre o estado atual e que ele seja usado por todas as threads. Para assegurar que isso ocorra, declare a variável como **volatile**.

Modificador transient

A palavra-chave **transient** é um modificador de tipo que indica que uma variável de instância não faz parte do estado persistente de um objeto. Quando uma variável de instância é declarada como **transient**, seu valor não precisa participar da *serialização*, que é o mecanismo pelo qual o estado de um objeto é salvo.

instanceof

Às vezes, é útil saber o tipo de um objeto durante o tempo de execução. Por exemplo, você pode ter uma thread que gere vários tipos de objetos e outra que os processe. Nessa situação, pode ser útil que a thread processadora saiba o tipo de cada objeto ao recebê-lo. Outra situação em que é importante conhecer o tipo de um objeto no tempo de execução envolve a coerção. Em Java, uma coerção inválida causa erro de tempo de execução e muitas coerções inválidas podem ser detectadas no tempo de compilação. No entanto, hierarquias de classes podem produzir coerções inválidas que só são detectadas no tempo de execução. Uma vez que uma referência da superclasse pode referenciar objetos da subclasse, nem sempre é possível saber no tempo de compilação se uma coerção envolvendo uma referência da superclasse é ou não válida. A palavra-chave **instanceof** resolve esses tipos de problemas.

Na sintaxe Java formal, **instanceof** é um operador. Ele tem esta forma geral:

refobj instanceof tipo

Aqui, *refobj* é uma referência à instância de uma classe, e *tipo* é um tipo de classe ou interface. Se *refobj* for do tipo especificado ou puder ser convertida para o tipo especificado, a expressão **instanceof** produzirá como resultado **true**, caso contrário, seu resultado será **false**. Logo, **instanceof** é o meio pelo qual o programa pode obter informações de tipo de um objeto no tempo de execução.

strictfp

Uma das palavras-chave mais estranhas é **strictfp**. Ela requer que os cálculos Java de ponto flutuante sigam rigorosamente o padrão IEEE 754. Exceto para constantes numéricas, normalmente o modelo de ponto flutuante Java não requer a adesão rigorosa a esse padrão. Para assegurar uma conformidade rigorosa, modifique a declaração de classe, método ou interface com **strictfp**.

assert

A palavra-chave **assert** é usada durante o desenvolvimento do programa na criação de uma *asserção*, uma condição que deve ser verdadeira durante a execução do programa. Por exemplo, se tivéssemos um método que precisasse sempre retornar um valor inteiro positivo, poderíamos verificar isso declarando que o valor de retorno é maior do que zero com o uso de uma instrução **assert**. No tempo de execução, se a condição for realmente verdadeira, nenhuma ação ocorrerá. No entanto, se a condição for falsa, um **AssertionError** será lançado. As asserções costumam ser usadas durante os testes para sabermos se alguma condição esperada está sendo atendida. Geralmente, elas não são usadas em código liberado.

A palavra-chave **assert** tem duas formas. A primeira é mostrada abaixo:

```
assert condição;
```

Aqui, *condição* é uma expressão que deve produzir um resultado **booleano**. Se o resultado for verdadeiro, a asserção é verdadeira e nenhuma outra ação ocorrerá. Se a condição for falsa, a asserção falhou e um objeto **AssertionError** padrão será lançado. Por exemplo,

```
| assert n > 0;
```

Se **n** for menor ou igual a zero, um **AssertionError** será lançado. Caso contrário, não ocorrerá ação alguma.

A segunda forma de **assert** é esta:

```
assert condição: expr;
```

Nessa versão, *expr* é um valor que é passado para o construtor de **AssertionError**. Esse valor será convertido em seu formato string e exibido se uma asserção falhar. Normalmente, especificamos um string para *expr*, mas qualquer expressão de um tipo diferente de **void** é permitida, contanto que defina uma conversão de string correta.

Para ativar a verificação de asserções no tempo de execução, você deve especificar a opção **-ea**. Por exemplo, para ativar o uso de asserções para a classe **Sample**, execute-a usando a seguinte linha:

```
| java -ea Sample
```

Métodos nativos

Apesar de raro, em alguns momentos podemos querer chamar uma sub-rotina escrita em uma linguagem que não seja Java. Normalmente, esse tipo de sub-rotina existe como código executável para a CPU e o ambiente em que estamos trabalhando – isto é, código nativo. Por exemplo, poderíamos querer chamar uma sub-rotina de código nativo para obter um tempo de execução mais rápido, ou querer usar uma biblioteca especializada de terceiros, como um pacote estatístico. No entanto, já que os programas Java são compilados para bytecode, que é então interpretado (ou compilado para código de máquina dinamicamente) pelo sistema de tempo de execução Java, parece impossível chamar uma sub-rotina de código nativo de dentro do programa Java. Porém, essa conclusão é falsa. Java fornece a palavra-chave **native**, que é usada para declarar métodos de código nativo. Uma vez declarados, esses métodos podem ser chamados de dentro do programa Java como chamaríamos qualquer outro método Java.

Para declarar um método nativo, preceda sua declaração com o modificador **native**, mas não defina um corpo para o método. Por exemplo:

```
| public native int doSomething();
```

Uma vez que você tiver declarado um método nativo, você deve fornecê-lo. Com frequência os métodos nativos são escritos na linguagem C. Após o método nativo ser criado, é necessário seguir uma série de etapas um pouco complexas para vinculá-lo ao código Java. O uso de métodos nativos é definitivamente uma técnica de programação avançada e não faz parte do escopo deste livro.

Pergunte ao especialista

P Já que estamos no tópico das palavras-chave, tenho uma pergunta sobre **this**. Estava examinando alguns exemplos de código Java na Internet e notei uma forma de **this** que usa parênteses. Por exemplo,

```
| this(x);
```

Pode me dizer o que essa instrução faz?

R A forma de **this** a que você está se referindo permite que um construtor chame outro dentro da mesma classe. A forma geral desse uso de **this** é mostrada aqui:

```
this(lista-arg)
```

Quando **this()** é executado, o construtor sobrecarregado que tem a mesma lista de parâmetros especificada por *lista-arg* é executado primeiro. Em seguida, se houver alguma outra instrução dentro do construtor original, ela será executada. A chamada a **this()** deve ser a primeira instrução dentro do construtor. Veja um exemplo simples:

```
class MyClass {
    int a;
    int b;

    // Inicializa a e b individualmente.
    MyClass(int i, int j) {
        a = i;
        b = j;
    }

    // Usa this() para inicializar a e b com o mesmo valor.
    MyClass(int i) {
        this(i, i); // chama MyClass(I, i)
    }
}
```

Em **MyClass**, somente o primeiro construtor atribui diretamente um valor a **a** e **b**. O segundo apenas chama o primeiro. Portanto, quando esta instrução é executada,

```
| MyClass mc = new MyClass(8);
```

a chamada a **MyClass(8)** executa **this(8, 8)**, que seria o mesmo que uma chamada a **MyClass(8, 8)**.

Chamar construtores sobrecarregados usando **this()** pode ser útil, porque evita a duplicação desnecessária de código. No entanto, é preciso ter cuidado. Construtores que chamam **this()** são executados um pouco mais lentamente do que os que contêm todo o seu código de inicialização em sequência. Isso ocorre porque o mecanismo de chamada e retorno usado quando o segundo construtor é chamado adiciona sobrevida. Lembre-se de que a criação de objetos afeta todos os usuários da classe. Se a classe for usada para criar grandes quantidades de objetos, é preciso comparar cuidadosamente os benefícios de um código menor e o maior tempo necessário à criação de um objeto. À medida que você for ganhando mais experiência em Java, esse tipo de decisão parecerá mais fácil de tomar.

Há duas restrições das quais é preciso lembrar ao usar **this()**. Em primeiro lugar, você não pode usar uma variável de instância da classe do construtor em uma chamada a **this()**. Em segundo, não pode usar **super()** e **this()** no mesmo construtor porque as duas devem ser a primeira instrução do construtor.

EXERCÍCIOS

1. Que método é chamado quando um applet é executado pela primeira vez? E qual método é chamado quando um applet é removido do sistema?
2. Explique por que um applet deve usar várias threads se tiver que ser executado continuamente.
3. Melhore o projeto da seção Tente isto 15-1 para que exiba o string passado como parâmetro. Adicione um segundo parâmetro que especifique o retardo (em milissegundos) existente entre cada giro da mensagem.
4. Desafio extra: crie um applet que exiba a hora atual, atualizada a cada segundo. Para fazê-lo, você deverá pesquisar um pouco. Aí vai uma dica para ajudá-lo a começar: uma maneira de obter a hora atual é usar um objeto **Calendar**, que faz parte do pacote **java.util**. (A classe **Calendar** é descrita no Capítulo 24.) Você já deve ter chegado a um ponto em que pode examinar a classe **Calendar** por conta própria e usar seus métodos para resolver esse problema.
5. Para solicitar que a janela de um applet seja reexibida, que método você deve chamar?
6. Descreva brevemente a palavra-chave **assert**.
7. Cite uma razão que explique por que um método nativo pode ser útil para alguns tipos de programas.
8. Que operador você pode usar para determinar o tipo de um objeto no tempo de execução?
9. Crie um **Applet** que exiba o padrão X a seguir. Ajuste o espaçamento entre os asteriscos e o tamanho do **Applet** para que o padrão preencha-o quase todo.



10. O método **paint()** da classe **Applet** tem um objeto **Graphics** como parâmetro. Nos exemplos deste capítulo, o objeto **Graphics** foi usado no desenho de um string. No entanto, pode fazer muito mais do que apenas desenhar strings. Por exemplo, ele também tem os métodos a seguir:

```
void drawLine(int xInicial, int yInicial, int xFinal, int yFinal)  
void drawRect(int esquerda, int topo, int largura, int altura)
```

Crie um applet que use esses dois métodos para desenhar uma casa.

11. Modifique o applet **Banner** da seção Tente isto 15-1 para que exiba dois banners rolantes. Use a mensagem que quiser em cada banner, mas faça um banner girar para a frente e o outro para trás.
12. Como mencionado neste capítulo, cinco dos métodos definidos para a classe **Applet** são **init()**, **start()**, **paint()**, **stop()** e **destroy()**. Agora suponhamos que um usuário abrisse uma página Web contendo um applet e o exibisse e ocultasse. O usuário também se alterna entre várias páginas Web, revisitando com frequência a do applet. Quais dos cinco métodos serão chamados com mais frequência e quais com menos frequência?
13. O método **getClass()** de **Object** pode ser usado para informar se um objeto **x** é instância de uma classe chamada **A**, como descrito a seguir:

```
| if( x.getClass() == A.class ) { ... }
```

Qual é a principal diferença entre essa instrução **if** e a mostrada abaixo?

```
| if( x instanceof A ) { ... }
```

14. O que aconteceria se você verificasse se **null** é instância de uma classe usando o operador **instanceof**? Por exemplo, a condição a seguir é verdadeira?

```
| (null instanceof String)
```

15. Crie um método **countTypes()** que use um array de **Objects** como parâmetro. Ele usa **instanceof** para determinar quantos dos valores do array são **Integers**, quantos são **Numbers** ou uma subclasse de **Number** que não seja **Double**, quantos são **Strings** e quantos são de algum outro tipo. Em seguida, exibe quantos de cada um foram encontrados. A soma dos três últimos números exibidos deve ser igual ao tamanho do array.

16. O que será exibido pelo programa abaixo? Por quê?

```
class Assertions {  
    public static void main(String[] args) {  
        assert 3 < 0 : "Oops";  
        System.out.println("End of method.");  
    }  
}
```

Introdução ao projeto orientado a objetos

PRINCIPAIS HABILIDADES E CONCEITOS

- Saber as propriedades de um software de alta qualidade
- Usar nomes apropriadamente
- Saber como aumentar a coesão
- Saber como reduzir a vinculação
- Saber como separar responsabilidades
- Entender as invariantes de classe
- Criar documentação interna e externa apropriada
- Entender o padrão Expert
- Saber como usar o encapsulamento e a ocultação de informações
- Saber a Lei de Demeter
- Saber o Princípio Aberto-Fechado
- Saber quando usar a herança *versus* a delegação
- Entender os padrões de projeto Adapter e Observer

Direcionamos o foco dos 15 capítulos anteriores para os elementos principais da linguagem de programação Java, inclusive suas palavras-chave, sintaxe e técnicas básicas. A essa altura, você já sabe como criar um programa que seja compilado corretamente e execute as ações desejadas. No entanto, os programas que criou são muito curtos. Neste capítulo, lhe apresentaremos outro aspecto da programação que será particularmente importante à medida que você começar a criar programas maiores e mais complexos, seja em cursos posteriores ou no local de trabalho. Esse aspecto é o projeto apropriado de programas orientados a objetos. Destacaremos principalmente questões de projeto, como: que objetos devem interagir com outros objetos, que objetos devem manter que dados, que objetos devem ter acesso a esses dados e que objetos devem poder tratar os dados. Isto é, vamos nos concentrar na maneira adequada de dividir as responsabilidades entre os vários objetos do sistema e em como fazê-los colaborar uns com os outros para resolver problemas com sucesso. Também abordaremos algumas questões de implementação relacionadas à obtenção de um projeto apropriado.

UM SOFTWARE ELEGANTE E POR QUE ISSO IMPORTA

O quanto é importante dedicar tempo a projetar seu software apropriadamente antes de implementá-lo? Considere os programadores que criam rapidamente algum código sem planejar ou projetar antes. Eles podem justificar suas ações dizendo que os programas são muito curtos e que sabem o que está ocorrendo no código. Além disso, podem dizer que o código não vai mais ser usado ou visualizado por outra pessoa, logo, por que perder tempo desnecessariamente com o projeto antes de criá-lo?

No caso de pequenos programas “rápidos e chatos”, como os curtos scripts de shell que só são usados uma vez, os programadores podem estar corretos. Geralmente não é produtivo passar muito tempo criando a versão mais elegante possível. A única coisa importante é que o código funcione corretamente. No entanto, você também precisa lembrar de que o código que os programadores consideraram como “descartável” muitas vezes não é descartado. Ele acaba sendo copiado e colado em outro programa ou se torna a base de um programa mais geral e complexo. Nesses casos, o tempo dedicado a projetar o software apropriadamente seria um tempo gasto produtivamente.

Projetar um software que seja pensado para uso intenso e por um longo período requer um investimento considerável de tempo e energia. Por exemplo, veja as bibliotecas de classes (como o pacote Swing de Java que conheceremos na próxima parte deste livro), que são usadas extensivamente por desenvolvedores de aplicativos. Não é suficiente que as classes da biblioteca não tenham erros; também é importante que tenham um bom projeto. Um projeto pobre para qualquer uma dessas classes pode causar problemas para todos os desenvolvedores que usarem o pacote. Por exemplo, se um recurso valioso ou importante for omitido, os desenvolvedores terão que criar seu próprio código, possivelmente de maneira muito desajeitada, para obter o que as classes da biblioteca deveriam ter fornecido.

Em sistemas muito grandes, em que vários programadores são envolvidos, é ainda mais importante dedicar um tempo significativo à análise do problema e ao projeto da solução antes de codificar algo. Nesses casos, não é apenas uma pessoa que conhece todas as partes do programa; em vez disso, cada programador trabalha em uma pequena parte dele. Se a solução não tiver sido bem projetada, uma alteração (a correção de um problema ou uma melhoria, por exemplo) feita por um programador em uma linha de código pode facilmente introduzir falhas no código criado por outros programadores.

Quando um programa tem milhares ou milhões de linhas de código, falhas são inevitáveis. O problema real é como reduzir a quantidade de falhas que ocorrem quando o código é criado, como aumentar a detecção e remoção das falhas que acabam entrando e como reduzir a quantidade de novas falhas introduzidas accidentalmente sempre que o código é modificado. Além disso, esse processo de diminuir e aumentar não é algo que só ocorra uma vez. O software muda continuamente devido aos remendos introduzidos para a correção de falhas ou às melhorias adicionadas a ele. Em outras palavras, um teste completo para a remoção de falhas é importante no desenvolvimento de um software, mas ele é tão importante quanto projetar e escrever o software de modo que o menor número possível de falhas seja introduzido e de modo que seja fácil modificar o código posteriormente sem introduzir novas falhas.

Portanto, por que os desenvolvedores não projetam e escrevem softwares de uma maneira que reduza as falhas? A resposta é que, mesmo que tentem, forças atuam contra eles. Fazer o trabalho corretamente exige tempo e dinheiro a curto prazo e os benefícios só aparecem depois. Enquanto isso, os projetos de software estão cada vez sob mais pressão para serem concluídos rapidamente e postos em produção antes de a janela de oportunidade de vendas fechar. Como resultado, com frequência o projeto inicial do software é especificado inadequadamente e, portanto, não é construída uma base sólida. E após o software ter entrado em produção, a pressão pela correção rápida das falhas e pela introdução de melhorias não permite reformulações mais importantes. Consequentemente, um sistema de software tende a se tornar uma “grande bola de lama”, que, como Foote e Yoder descrevem em <<http://www.laputan.org/mud/>>, é uma “selva de código emaranhado, como uma vasta fita ou arame estruturado acidentalmente e malfeito”. Com o tempo, a degradação desse tipo de software de bola de lama dificulta cada vez mais encontrar e corrigir falhas e adicionar melhorias, custando mais tempo e dinheiro e resultando em mais pressão para o adiamento de uma reformulação em grande escala. E assim o ciclo vicioso prossegue.

Há outras forças que empurram os softwares em direção às bolas de lama. Elas são a falta de habilidade, conhecimento e experiência dos desenvolvedores sobre como criar um software de alta qualidade. Quando os desenvolvedores não têm experiência no projeto de sistemas de software de qualquer tamanho ou complexidade, ou quando criam um aplicativo empresarial sem ter conhecimento da área específica e de suas necessidades e requisitos, é fácil o software ficar confuso. Mesmo se os desenvolvedores entenderem totalmente um sistema, a sujeira não irá embora se eles não tiverem as ferramentas (habilidades ou conhecimento) para limpá-la.

O que pode ser feito para combatermos a tendência de os sistemas de software se tornarem bolas de lama? Resolver as pressões impostas por custo e tempo não faz parte do escopo do capítulo. Nossa preocupação são as habilidades, o conhecimento e a experiência de que os desenvolvedores precisam para fazer um trabalho de alta qualidade. Neste capítulo, apresentaremos algumas das coisas que um desenvolvedor de software deve saber para poder projetar sistemas de alta qualidade de modo a combater as forças que levam a um resultado confuso.

Propriedades de um software elegante

Como saber a diferença entre um software bem ou mal projetado? Infelizmente, não há uma definição precisa para software “bem projetado”. Um bom projeto de software é tanto ciência quanto arte. À medida que as pessoas obtêm mais experiência na profissão de criação de softwares, elas desenvolvem uma percepção da diferença entre um software de alta ou baixa qualidade. Isto é, desenvolvem um senso de estética para o projeto e a implementação de softwares.

Mesmo que não tenhamos como definir precisamente o que é um software bem projetado, muitas propriedades dele podem ser dadas. Neste capítulo, consideraremos as propriedades a seguir de um software de alta qualidade:

- Usável – fácil de o cliente usar.
- Completo – atende todas as necessidades do cliente.
- Robusto – lida com situações incomuns elegantemente.

- Eficiente – consome um período de tempo e outros recursos de maneira aceitável.
- Escalável – será executado de maneira correta e eficiente mesmo quando o problema crescer em vários graus de magnitude.
- Legível – fácil de um engenheiro de software entender o projeto e o código.
- Reutilizável – pode ser reutilizado em outras configurações.
- Simples – não é necessariamente complexo.
- De fácil manutenção – os defeitos podem ser encontrados e corrigidos facilmente sem a introdução de novos defeitos.
- Extensível – pode ser melhorado facilmente sem danificar o código existente.

Usaremos o termo *elegante* para descrever um software com essas propriedades. Esse termo é apropriado porque é um prazer para os desenvolvedores fazer manutenção em um sistema de software bem projetado e implementado. Eles conseguem encontrar facilmente os locais em que alterações têm que ser feitas e podem fazê-las com poucas preocupações de que introduzam novas falhas. Por outro lado, um sistema mal projetado pode causar acessos de ira nos engenheiros de software que se virem forçados a cuidar de sua manutenção.

Neste capítulo, vamos nos preocupar mais com as últimas seis propriedades. Discutiremos diretrizes e princípios comuns de projeto que podem ajudar a dar ao software tais propriedades. Essas diretrizes e princípios evoluíram com o passar dos anos, conforme seus seguidores foram notando que certas coisas funcionavam bem, e outras, mal.

Verificação do progresso

1. Em sistemas de software grandes, tudo o que importa no software é que ele funcione corretamente. Verdadeiro ou falso?
2. Liste três propriedades de um software elegante.

Pergunte ao especialista

P Você pode me indicar o caminho certo para eu aprender mais sobre os princípios e padrões de projeto?

R Se você quiser aprender mais, há muitos pontos de partida interessantes. *Effective Java*, de Joshua Block (2ª edição, 2008, Addison-Wesley), é um bom recurso para o aprendizado das melhores práticas de criação de um código Java elegante. Um texto clássico sobre o projeto orientado a objetos em geral é *Object-Oriented Software Construction*, de Bertrand Meyer (2ª edição, 1997, Prentice Hall). A referência clássica dos padrões de projeto (que serão discutidos no fim deste capítulo) é *Projeto Patterns*, de Eric Gamma, R. Helm, R. Johnson e J. Vlissides (1995, Addison-Wesley). Você também pode encontrar muitos recursos excelentes na Internet.

Respostas:

1. Falso.
2. Quaisquer das propriedades a seguir: completo, robusto, eficiente, escalável, legível, reutilizável, simples, de fácil manutenção e extensível.

MÉTODOS ELEGANTES

Antes de conhecer a maneira apropriada de projetar um grande sistema de software orientado a objetos, inclusive a divisão adequada do sistema em classes com responsabilidades e colaborações convenientes, temos de entender o que torna uma classe individual elegante. Qual é o papel de uma classe? Que tipo de responsabilidades ela deve ter? Isto é, que dados ela deve manter e o que tem de poder fazer com esses dados e com outros? Em outras palavras, qual deve ser seu comportamento? Além disso, uma vez que tivermos tomado essa decisão, como devemos implementar esse comportamento como métodos?

Comecemos com os métodos. A elegância de um método está relacionada ao seu tipo de retorno, ao nome dos parâmetros ou ao corpo do método? Na verdade, está relacionada a todas essas partes. Para nos ajudar a entender as questões existentes em torno da elegância dos métodos, consideraremos primeiro um exemplo simples.

Suponhamos que estivéssemos desenvolvendo uma nova classe **DataHolder** que, quando solicitada, deve poder inserir novos valores em um array de inteiros referenciado por sua variável de instância privada **data**. Isto é, esse ato de inserção tem de fazer parte do comportamento da classe. Como resultado, temos de adicionar um ou mais métodos à classe para executar a inserção. Há vários métodos que poderíamos criar para dar à classe esse comportamento. Por exemplo, poderíamos criar um método **insert()** que use como parâmetro o novo valor inteiro a ser inserido e um índice inteiro indicando onde ele deve ser inserido. O método apenas insere o valor no local apropriado do array **data**. Outra alternativa seria dividir o processo em três métodos separados, em que o primeiro método, **increaseCapacity()**, aumentaria o array, se necessário; o segundo, **shift()**, deslocaria os outros inteiros para dar espaço para o novo inteiro; e o terceiro, **put()**, inseriria o novo inteiro no local recém-liberado. Além disso, poderíamos deixar para o usuário a chamada aos três métodos individualmente ou criar um método **insert()** adicional que chamassem esses outros três métodos.

Também poderíamos usar um nome que não fosse “insert” para o método ou alterar a ordem ou o número de parâmetros. Por exemplo, poderíamos sobrecarregar o método para que dois valores pudessem ser inseridos ao mesmo tempo. Também poderíamos ter vários tipos de retorno. O método poderia ter **void** como tipo de retorno ou, por exemplo, retornar informações indicando se foi bem-sucedido na inserção do novo valor. Qual dessas opções é a melhor? Voltaremos continuamente a esse exemplo nesta seção do capítulo e, no processo, responderemos a essa pergunta e à pergunta mais geral sobre o que torna um método elegante.

Convenções de nomenclatura

Vários princípios da criação de software podem nos ajudar a comparar e julgar as abordagens mencionadas no parágrafo anterior. Um princípio é *usar nomes que revelem a finalidade*. Ele será examinado com mais detalhes porque não se aplica só aos métodos. Pode ser aplicado a qualquer parte do software que use nomes, inclusive métodos, variáveis, classes e interfaces.

Um nome de método deve indicar a finalidade do método, isto é, o que o método deve fazer. O nome não deve indicar *como* o método atingirá seu objetivo e sim *qual* é o objetivo. Um método que não retorne um valor (um método com tipo de

retorno **void**) deve ter um nome composto por um verbo ou um verbo e seu complemento, como **print()** ou **setName()**. Nomes vagos como **doIt()** são inadequados. Um método que retorne um valor deve ter um nome que reflita o valor que está sendo retornado, por exemplo, **length()** ou **name()**. Esse tipo de método também poderia ter um verbo e seu complemento como nome, a convenção sendo a palavra “get” seguida do valor retornado. Por exemplo, um método que retorne o tamanho de alguma estrutura de dados poderia receber o nome **size()** ou **getSize()**. Com relação ao nosso exemplo **DataHolder**, o nome **insert()** é satisfatório, pois descreve sucintamente o que o método deve fazer.

Classes e interfaces devem ter nomes que refletem o papel ou a finalidade de objetos desses tipos. O nome de uma classe é a primeira pista que o leitor tem do papel real que ela desempenha em um projeto e, portanto, vale a pena dedicar algum tempo na busca de um nome apropriado. Normalmente os nomes de classes são substantivos, como **Date** ou **ComputerCard**. Quase sempre interfaces têm nomes que terminam em “-able” ou “-ible”, como **Cloneable** ou **Iterable**.

Para concluir, as variáveis têm de ser nomeadas apropriadamente para melhorar a legibilidade. Considere um programa com uma variável chamada **nT**. Um argumento a favor de um nome assim seria que ele é curto e, portanto, economiza tempo de digitação e ajuda a encurtar as linhas de código. Contudo, o nome não é significativo no que diz respeito a indicar a função da variável, e o leitor é forçado a memorizá-la. Por outro lado, chamar a variável de **numberOfThreads** ou **NumThreads** ou até mesmo **numThrds** em vez de **nT** aumenta drasticamente a facilidade de entender o código.

Uma variável **boolean** deve ter um nome apropriado para o valor que representa. Geralmente não deve representar a negação de um valor, então, por exemplo, não devemos chamar nossa variável de **notYetDone**, que poderia resultar na necessidade de expressões como a negativa dupla **!notYetDone**. Em vez disso, seria melhor chamá-la de **done** e usar a expressão **!done** sempre que **notYetDone** tivesse de ser usada. É claro que o nome **done** também não é ideal, já que a palavra não contém informações suficientes para ajudar o leitor a entender o papel da variável. O nome da variável deve informar *que* atividade é ou não executada. Por exemplo, se essa variável estiver sendo usada para indicar que uma imagem gráfica acabou de ser carregada e já pode ser vista, um nome melhor seria **doneLoading**.

Lembre-se, nomes ruins não levam apenas a enganos; eles tornam o sistema inteiro mais difícil de entender, o que vai contra nosso desejo de ter um software legível.

Coesão dos métodos

Outro princípio da criação de softwares é: *os métodos devem fazer apenas uma coisa e fazê-la bem*. Isto é, um método não deve executar duas ou mais tarefas a menos que façam parte de uma mesma ação coesa. Por exemplo, faz sentido o método **insert()** mencionado acima aumentar o array, mover os outros valores para dar espaço para o novo valor e adicionar o novo valor, porque tudo isso faz parte de uma ação coesa. Por outro lado, um método que tanto insira novos itens em um array **data** quanto determine se um string está na forma de um endereço de e-mail não é coeso. A principal razão para evitar a criação desse tipo de método é que, com frequência, queremos

executar uma das ações, mas não as duas. Nesse caso, um método que execute as duas é desnecessário. Um método é muito mais reutilizável quando faz apenas uma coisa.

Aqui está um exemplo de um método não coeso:

```
void doThisOrThat(boolean flag) {
    if( flag ) {
        //...vinte linhas de código para fazer isso...
    }
    else {
        //...vinte linhas de código nessa outra ação...
    }
}
```

Esse método está claramente tentando fazer duas coisas e usando o flag para determinar qual delas fazer. Seria melhor termos dois métodos auxiliares separados, como **doThis()** e **doThat()**, nenhum deles precisando de um flag. Uma vez que tivermos esses métodos, poderemos reescrever o método acima na forma:

```
void doThisOrThat(boolean flag) {
    if( flag ) doThis();
    else doThat();
}
```

Agora esse código é aceitável (exceto pelos nomes **flag**, **this** e **that**, que não revelam a finalidade), já que o método está apenas agindo como um centro de despacho e, portanto, está fazendo apenas uma coisa e fazendo-a bem.

Um corolário interessante para o princípio da coesão é o princípio *um método deve modificar o estado de um objeto ou de objetos existentes ou retornar um valor, mas não ambos*. Com *estado* de um objeto, queremos dizer os valores de suas variáveis de instância. Novamente, o raciocínio por trás desse princípio é o de que, às vezes, podemos querer modificar o estado de um objeto, mas não retornar um valor; em outras, podemos querer retornar um valor, mas não modificar um objeto. Logo, a título de reutilização, é melhor separar essas ações em dois métodos separados. Observe que a adesão rigorosa a esse princípio requer que não haja modificação de código dentro de métodos que retornem valores. Por exemplo, um método que informe se um array está classificado ou revele qual o maior valor do array retorna um valor e, portanto, não deve modificar o array. Outra maneira de formular esse princípio seria dizendo que métodos que retornam valores não devem ter efeitos colaterais, e métodos com efeitos colaterais não devem retornar valores.

Dito isso, observe que temos usado a terminologia “diretrizes” e “princípios” em vez de “regras”. Ou seja, essas diretrizes devem ser levadas em consideração no projeto ou implementação de um sistema de software, mas não são regras a serem rigidamente seguidas. Em algumas situações, pode ser melhor para o projeto se ignorarmos um ou mais desses princípios. E alguns códigos legados e de biblioteca não os seguem. Por exemplo, o pacote **java.util** tem uma classe **Stack** com um método **pop()** que tanto remove o objeto do topo da pilha quanto o retorna. (E, claro, a classe **SimpleStack** desenvolvida anteriormente neste livro também funciona assim.) Tecnicamente, esse comportamento desobedece ao princípio mencionado acima, mas é aceitável porque é um comportamento sensato nesse caso e, o mais importante, por-

Pergunte ao especialista

P E quanto a um método que tente modificar o estado de um objeto e retornar um valor **boolean** indicando sucesso ou falha na tentativa? Há algo errado nisso?

R Essa abordagem é comum, principalmente em linguagens sem mecanismos de tratamento de exceções. Logo, é correto criar um método assim, ainda que ele não siga a diretriz. Mas os programadores de Java também devem considerar alternar os cursos de ação, como lançar uma exceção se o método não for bem-sucedido. De qualquer forma, independentemente de seu sistema conseguir ou não seguir a diretriz, vale a pena tentar. Considere isso um desafio para ver se você consegue projetar métodos melhor do que projetistas anteriores fizeram.

que esse método vem sendo usado há anos e alterar seu comportamento agora seria muito pior do que deixar que ele continue desobedecendo à diretriz.

Objetos bem-formados

Aqui está outro princípio relativo a métodos elegantes: *um método não privado deve manter um objeto em um estado bem-formado*. Por exemplo, suponhamos que você tivesse uma classe com duas variáveis de instância: um array de inteiros **data** e um inteiro **max** que armazenasse o valor máximo do array **data**. Você deve tomar cuidado para não ter métodos na classe que modifiquem o array sem atualizar também a variável **max**. É a esse tipo de coerência interna que nos referimos quando dizemos que uma classe é “bem-formada”.

Como formular os requisitos de coerência dos objetos de uma classe? Isto é, como saber se a instância de uma classe está bem-formada? Uma boa maneira é criar uma lista das invariantes da classe. Uma *invariante de classe* é uma declaração que fornece os requisitos sobre o estado de instâncias da classe entre chamadas de método públicas. Um exemplo de invariante de classe é a declaração “O valor de **max** e o maior valor do array **data** devem ser iguais”.

Como outro exemplo, considere a implementação a seguir da classe **DataHolder** discutida anteriormente.

```
class DataHolder {
    private int[] data;

    public DataHolder() {
        data = new int[0];
    }

    public void insert(int x, int index) {
        increaseCapacity();
        shift(index);
        put(x, index);
    }

    private void increaseCapacity() {
        int[] newData = new int[data.length+1];
        for(int i = 0; i < data.length; i++)
            newData[i] = data[i];
    }
}
```

```

        data = newData;
    }

private void shift(int index) {
    for(int i = index; i < data.length-1; i++)
        data[i+1] = data[i];
}

private void put(int x, int index) {
    data[index] = x;
}
public int get(int index) {
    return data[index];
}
public int size() {
    return data.length;
}
}

```

Observe que o array **data** está sempre repleto de dados que foram inseridos. Isto é, nunca há locais não usados no array. Essa é uma invariante para objetos da classe **DataHolder**. Se a invariante for rompida, pode ocorrer um comportamento inesperado. Para evitar isso, os três métodos auxiliares **increaseCapacity()**, **shift()** e **put()** foram tornados privados. Como resultado, os objetos dessa classe podem ficar temporariamente malformados, por exemplo, após **increaseCapacity()** ter sido chamado, mas serão restaurados para um estado bem-formado antes de algum método público retornar. Se **increaseCapacity()** ou **shift()** fossem tornados públicos, qualquer usuário do objeto poderia chamá-los isoladamente, causando a malformação do objeto.

Para que as invariantes de classe sejam mantidas, é preciso que as variáveis de instância sejam privadas ou finais se tiverem envolvimento com alguma invariante de classe. Caso contrário, outro objeto poderia accidental ou maliciosamente alterar o valor da variável e gerar o objeto malformado.

Documentação interna

O último princípio que queremos mencionar nesta seção é *incluir uma documentação externa completa para o usuário e uma documentação interna para o desenvolvedor*.

Mesmo que um pacote de software seja bem projetado, se uma documentação não for incluída ele perderá muito de seu valor. Considere a frustração de um projetista de software que sabe que é quase certo que uma biblioteca tenha as ferramentas de que ele precisa, mas não tem como afirmar porque a documentação dos componentes é inadequada. Ou a frustração da pessoa encarregada de corrigir uma falha grande em uma seção de código que contém pouco ou nenhum comentário explicando por que faz as coisas dessa maneira. Um software elegante evita essa duas situações incluindo uma documentação apropriada.

Documentação interna é a documentação para alguém que está examinando o código-fonte. Ela deve fornecer informações não prontamente disponíveis no próprio código. Deve resumir o que está sendo feito, por que está sendo feito e por que está sendo feito dessa maneira específica. Por exemplo, a documentação pode explicar que um método foi implementado de uma determinada maneira para permitir a fácil

modificação posterior, porque ele é mais eficiente dessa forma ou porque essa é a maneira mais simples. Ela deve informar a relação custo/benefício envolvida entre as várias implementações possíveis.

A documentação interna também deve citar claramente a existência de qualquer invariante de classe ou método. Uma pessoa que estiver modificando um método existente sem ter conhecimento dessas invariantes pode criar um código que as invalide.

Uma documentação interna útil também pode dar uma visão geral da implementação. O comentário interno de um método pode explicar o algoritmo usado por ele, se isso já não tiver sido mencionado na documentação externa. Por exemplo, a documentação interna de um método `sort()` poderia informar que o algoritmo de classificação rápida recursiva foi usado.

A documentação interna é composta em grande parte por comentários no código-fonte, mas o próprio código pode ser uma parte útil se for bem escrito. Se o programador usar nomes de métodos e variáveis que revelem as finalidades e combiná-los apropriadamente, o código pode chegar a um bom nível de autodocumentação e, portanto, haverá menos necessidade de documentação adicional. Infelizmente, um código todo autodокументado é um ideal raramente atingido, logo, quase sempre alguns comentários adicionais são necessários.

É claro que os comentários internos devem ser úteis para o entendimento do código. No entanto, também podem atrapalhar o entendimento se, por exemplo, não forem atualizados quando o código mudar.

A documentação interna também pode ajudá-lo a detectar partes elegantes de seus métodos. Por exemplo, quando um código precisa de um grande número de comentários, geralmente isso é indicação de código insatisfatório. Se você perceber que está incluindo muitos comentários, provavelmente o código deve ser reescrito. Para ser mais específico, se estiver escrevendo muitos comentários para resumir seções de código de um método, ele pode estar fazendo coisas demais.

Documentação externa

A *documentação externa* é para usuários do código que não tenham como examinar o código-fonte ou não estejam interessados nele. Ela descreve as classes, interfaces, métodos, campos e pacotes públicos e como usá-los. Também pode incluir documentos de projeto indicando os relacionamentos entre as classes e os papéis desempenhados por elas. Essa documentação, se feita corretamente, deve descrever todos os aspectos do comportamento de cada método de que o chamador precisar.

É importante entender que o único comportamento de um método em que o usuário deve confiar é o comportamento documentado. A confiança deve ser limitada dessa forma porque, durante a manutenção do sistema, um programador pode ter de modificar o método. O responsável pela manutenção pode alterar um comportamento documentado, o que pode causar problemas sérios para quem precisar desse comportamento.

O que deve ser incluído na documentação externa de um método? Essa documentação deve incluir a assinatura completa e o tipo de retorno do método para que o usuário saiba a sintaxe a ser usada ao chamá-lo, mas muitas outras coisas são necessárias. Por exemplo, considere o método da classe `java.lang.Math` com o cabeçalho a seguir:

```
|public static double rint(double a)
```

Para um programador experiente que conheça `rint()`, esse cabeçalho é suficiente para lembrá-lo do tipo do argumento e do tipo de retorno. Mas, para as outras pessoas, o método precisa de documentação adicional. Um nome mais descritivo ajudaria muito a esclarecer qualquer confusão, mas, mesmo com essa alteração, uma descrição textual do que o método retorna e de como ele usa o parâmetro é essencial para garantir o uso apropriado.

Como outro exemplo, considere a documentação a seguir de uma função `nthRoot()`:

```
// retorna a raiz enésima do valor double
public double nthRoot(double value, int n)
```

Essa documentação é inadequada. Ela explica o comportamento principal do método, mas não especifica o que acontece em casos especiais, por exemplo, quando `value` é `-1` e `n` é `2`. Aqui está um exemplo de documentação mais completa:

```
// Retorna a raiz enésima do valor double.
// Se n é par é retornado o valor positivo da enésima raiz.
// Se n é ímpar a raiz enésima terá o mesmo sinal do valor.
// Se n < 0 ou n é ímpar o valor < 0 então será
// lançada uma IllegalArgumentException
public double nthRoot(double value, int n)
```

A documentação externa não deve especificar detalhes demais sobre a implementação do método. Quase sempre esses detalhes são irrelevantes para o usuário e, portanto, apenas tornam a documentação confusa.

Como saber quando a documentação externa de um método é satisfatória? Para responder a essa pergunta, é útil pensarmos no método como um serviço que um objeto dessa classe executará para quem o solicitou/demandou. Para os usuários de sua classe saberem o que o serviço envolve e se querem usá-lo, você precisa especificar claramente o que eles têm de fazer (por exemplo, os argumentos que precisam fornecer) e o que seu objeto fará em troca quando executar o método. Isto é, pense no método como um contrato. Se um usuário fornecer argumentos apropriados, seu método promete executar um serviço específico. Geralmente, os argumentos apropriados são chamados de *pré-condições*. Elas indicam o que deve ocorrer para o método funcionar. É costume chamarmos o serviço específico que será executado de *pós-condição*. Ela indica o que ocorrerá quando o método for executado.

Observe que é incumbência do cliente assegurar que as pré-condições sejam atendidas antes de o método ser chamado. Se o cliente tentar usar o método sem atender às pré-condições, pode acontecer qualquer coisa e o método pode apresentar um comportamento desconhecido, inclusive interromper o programa, ser executado infinitamente ou parecer estar sendo executado corretamente, mas na verdade gerar lixo.

Como exemplo, voltemos ao método `insert()` da classe `DataHolder`. Você deve lembrar de que ele usa dois argumentos: um valor inteiro a ser inserido e um índice inteiro indicando o local da inserção. Pelo menos dois conjuntos de pré-condições e pós-condições são possíveis para esse método, dependendo de como o projetista quiser sua implementação. Aqui estão elas:

```
// Insere o valor dado no index do DataHolder
```

```
// Se index < 0 ou index > size()
//     uma IllegalArgumentException será lançada
void insert(int value, int index)
```

Essa abordagem explica o que acontecerá a todos os argumentos possíveis e, portanto, não há pré-condições. Uma abordagem alternativa seria especificar pré-condições e deixar para o usuário a tarefa de assegurar que os argumentos as atendam. Veja como ficaria essa documentação:

```
// Insere o valor dado no index do DataHolder
// Pré-condição: 0 <= index e index <= size()
void insert(int value, int index)
```

Embora as duas abordagens funcionem, há desvantagens na segunda. Se um método especificar uma pré-condição e o usuário chamá-lo acidentalmente com argumentos inválidos, o resultado pode ser um valor sendo inserido de maneira incorreta. O programa do usuário pode continuar a ser executado com valores incorretos, o que dificultaria a posterior detecção de onde o erro ocorreu ou até mesmo se um erro ocorreu.

A primeira abordagem é um exemplo de uma prática chamada “programação defensiva”. Você sabe que é quase certo que erros ocorrerão e que entradas inválidas serão fornecidas aos métodos, portanto, certifique-se de que seus métodos possam se defender contra essas entradas fazendo algo explícito de uma maneira que seja útil para o usuário.

Como na documentação interna, um problema da criação de documentação externa é mantê-la sincronizada com o código à medida que o projeto evolui. Quando o código é modificado (por exemplo, erros são corrigidos, recursos são adicionados), é tentador, principalmente quando o tempo é curto, adiar a atualização da documentação, o que resulta em uma documentação que não coincide mais com o código. Uma maneira de resolver o problema de manter a documentação atualizada é gerar documentação externa a partir do código-fonte ou *vice-versa* para que eles fiquem sempre sincronizados.

O *Javadoc* é uma ferramenta e técnica de documentação projetada para essa geração de documentos. Ele especifica uma sintaxe para ser usada em comentários no código-fonte, que são então coletados pelo aplicativo Javadoc e convertidos em documentação externa. O Javadoc foi usado para gerar a documentação das APIs Java que pode ser lida online. Infelizmente, seu uso não garante que a documentação externa ficará sempre sincronizada com o código-fonte, mas apenas que ela estará sincronizada com os comentários formatados pelo Javadoc no código.

Como exemplo de notação Javadoc, o cabeçalho do método **nthRoot()** discutido anteriormente é apresentado aqui usando essa notação:

```
/** 
 * Retorna a raiz enésima do valor double.
 * Se n < 0 ou n é par e valor < 0,
 * então uma IllegalArgumentException será lançada.
 *
 * @param o valor double cuja raiz é desejada
 * @param n inteiro que indica a raiz a ser calculada
 *
 * @return a raiz enésima do valor
```

```

*      Se n é par, a raiz enésima positiva é retornada.
*      Se n é ímpar a raiz enésima terá o mesmo sinal do valor.
*
* @throws IllegalArgumentException
*      se n < 0 ou n é par e valor < 0.
*/

```

Não estamos usando a notação Javadoc para documentar o código deste livro por razões de espaço, mas você deve usá-la em cursos posteriores ou quanto terminar a faculdade, no local de trabalho. Uma visão geral do Javadoc pode ser encontrada no Apêndice A.

Verificação do progresso

1. Um método deve fazer apenas uma coisa e fazê-la bem. Verdadeiro ou falso?
2. Instruções que especificam requisitos sobre o estado dos objetos de uma classe entre chamadas de método são denominadas _____.
3. A documentação para o programador que está encarregado de fazer manutenções no código-fonte se chama documentação _____.
4. Uma condição que deve ser atendida antes de uma chamada de método funcionar corretamente é denominada _____.

CLASSES ELEGANTES

Agora que você tem uma compreensão mais exata do que é um método de alta qualidade, podemos explicar melhor o que faz uma classe ou um conjunto de classes ser de alta qualidade. Uma classe pode ter várias responsabilidades e colaboradores. Ela deveria ser dividida em duas ou mais classes menores? Deveria receber mais responsabilidades? Deveria colaborar com outras classes mais ou menos do que colabora agora? A classe deve ser movida para um pacote diferente? Para nos ajudar a responder essas perguntas, apresentaremos mais princípios de projeto.

A coesão das classes e o padrão Expert

O princípio básico que temos que considerar é *classes, assim como métodos, devem fazer apenas uma coisa e fazê-la bem*. Isto é, as classes devem modelar apenas uma ideia.

Essa diretriz não significa que cada classe deve ter exatamente um método. Em vez disso, diz que uma classe deve modelar um conceito e todos os seus métodos devem estar relacionados e ser apropriados a esse conceito. Ou seja, todas as responsabilidades da classe devem servir ao conceito que está sendo modelado por ela. Uma vantagem de seguir essa diretriz é que o comportamento da classe e seu papel em um

Respostas:

1. Verdadeiro.
2. invariantes de classe
3. interna
4. pré-condição

sistema de software ficam muito mais claros para todos. E a classe se torna mais reutilizável quando não está sobrecarregada com responsabilidades e dados irrelevantes para sua finalidade principal.

Por exemplo, uma classe que faz apenas uma coisa muito bem é **String**. Todos os métodos e dados dessa classe estão ligados a somente um conceito, a saber, uma sequência de caracteres. Seus métodos fornecem ferramentas úteis para o tratamento da sequência.

Um exemplo simples de classe malformada seria o de uma classe que fosse responsável pelo armazenamento de todas as informações relacionadas a uma pessoa (por exemplo, nome, endereço, idade) e todas as informações relacionadas ao carro atual dela (como a marca, modelo, cor, ano). Em vez disso, seria mais apropriado termos os dados da pessoa e os dados do carro armazenados em objetos separados, que, se necessário, teriam referências apontando um para o outro.

Outra classe malformada seria uma classe “onipotente” que controlasse todos os objetos de um sistema de software grande. Ela seria uma classe mestra com todas as responsabilidades e as outras classes seriam apenas escravas ou depósitos de dados. Essa abordagem contradiz o paradigma da orientação de objetos que diz que deve haver um controle descentralizado, com as responsabilidades de várias ações espalhadas entre as classes colaboradoras.

Outro termo que pode ser usado para descrever a natureza dedicada de uma classe é *coesão*. Quando falamos que cada classe deve “fazer apenas uma coisa”, queremos dizer que ela deve ter uma alta coesão, de modo que todos os seus comportamentos e responsabilidades estejam solidamente relacionados.

Há outro princípio geral que pode ajudá-lo a decidir que classe ficará com que responsabilidades. Suponhamos que um objeto armazenasse um conjunto de dados e que alguns dos dados do conjunto tivessem de ser excluídos. Que objeto deve fazer a exclusão? A escolha natural seria o objeto que mantém o conjunto. Esse objeto tem acesso aos dados necessários para a execução da tarefa desejada e, portanto, é ele quem deve executá-la. O princípio aqui é *o objeto que contém os dados necessários à execução de uma tarefa deve ser quem a executará*. Esse princípio se chama padrão *Expert*.

Por exemplo, considere um objeto **CarDealer** que armazene, entre outras coisas, um conjunto de objetos **Car** correspondente aos carros atualmente em estoque. Suponhamos que você quisesse saber se há alguma minivan azul no estoque. Você (o usuário) pode descobrir sozinho solicitando a **CarDealer** uma lista de todos os carros e examinando a lista à procura de uma minivan azul. Ou pode perguntar a **CarDealer** se ele tem alguma minivan azul em estoque. Claro que a última opção é melhor. **CarDealer** é o objeto com os dados necessários à execução da tarefa e, portanto, o ideal é que a execute.

Resumindo, o padrão *Expert* está sendo violado sempre que a classe **A** tem um método que obtém vários dados de um objeto da classe **B**, trata esses dados e possivelmente retorna um resultado. O objeto da classe **B** tem todos os dados necessários e, portanto, deveria executar o tratamento, em vez de um objeto da classe **A**. Isto é, provavelmente o método da classe **A** deveria ser movido para a classe **B**.

Uma pista de que seus objetos podem não estar executando suas tarefas apropriadamente é a existência de muitas chamadas a métodos acessadores (métodos `getX()`). Se houver muitas dessas chamadas, você deve se perguntar: o que os outros objetos

estão fazendo com os dados que obtêm quando chamam um desses métodos? Sua classe que tem os métodos `getX()` não deveria estar executando essa tarefa para eles?

Observe como essa questão está relacionada ao nosso princípio de uma classe bem projetada fazendo apenas uma coisa e fazendo-a bem. Um objeto que estiver fazendo bem o seu trabalho, além de recuperar dados, também dará a eles o tratamento necessário em vez de esperar que você mesmo os trate.

Dito isso, há um conflito entre o princípio de fazer apenas uma coisa e fazê-la bem e o padrão Expert. Por exemplo, digamos que um aplicativo fosse o *front end* de um banco de dados. Ele obtém alguns dados no banco de dados e então os exibe em uma janela. O padrão Expert sugere que o banco de dados seja responsável pela exibição de seus próprios dados. No entanto, um objeto de banco de dados também deve ter alta coesão, o que sugere que ele não deve ser responsável por duas atividades completamente diferentes, isto é, gerenciar os dados armazenados e exibi-los em uma GUI. Em muitas de suas classes, você terá que decidir onde traçar a fronteira da coesão (fazer apenas uma coisa) *versus* o padrão Expert (fazer tudo que for possível com os dados para todos).

Evitando duplicação

Na seção anterior, falamos sobre separar as responsabilidades entre as classes e, em particular, assegurar que nenhuma classe tenha muitos tipos de responsabilidades diferentes. Mas há outra questão sobre responsabilidades que ainda não abordamos: há algo errado em duas classes diferentes terem a mesma responsabilidade? Cada responsabilidade deve ser dada a apenas uma classe? A resposta é fácil de adivinhar. Trata-se de outro princípio: *evitar duplicação*.

Essa diretriz também é conhecida como o princípio “DRY”: *don’t repeat yourself* (não se repita). Na verdade, ela diz bem mais do que apenas que as responsabilidades não devem ser duplicadas entre as classes. A duplicação pode ocorrer em muitas formas. Por exemplo, poderíamos ter cópias das mesmas informações, código duplicado dentro de um método ou entre dois métodos, ou métodos duplicados em duas classes diferentes. Além disso, pode haver duplicação de processos, ou seja, a execução repetida desnecessária de um código. Todas essas formas de duplicação resultam em projetos e códigos deselegantes.

O que há realmente de errado na duplicação? Resumindo, um código com duplicação é menos legível e de manutenção mais difícil do que um código sem duplicação.

Para dar uma ideia melhor dos problemas que podem ocorrer, consideremos a duplicação de dados. Por que cópias dos mesmos dados devem ser evitadas? Se várias classes precisarem saber as mesmas informações – por exemplo, se houver vários objetos de administração de escola que precisam acessar os mesmos registros de alunos –, não faz sentido dar a todos eles cópias dessas informações? Em alguns casos faz, se as informações nunca mudarem (isto é, se forem imutáveis). Nesse caso, é correto que haja cópias, embora elas possam desperdiçar espaço, principalmente se forem grandes. No entanto, se as informações duplicadas puderem ser modificadas, será muito fácil as cópias perderem a sincronização; ou seja, uma cópia pode estar atualizada enquanto, por alguma razão, talvez accidentalmente, uma ou mais das outras cópias não estão.

Em vez de duplicar os dados, que tal manter apenas uma cópia deles, mas permitir a existência de vários pontos de acesso a esses dados? Por exemplo, suponhamos que você tivesse um objeto da classe **Company** que fosse responsável por manter um conjunto de objetos **Employee**. Suponhamos que a classe **Company** implementasse seu conjunto com um array. Outros objetos podem ter que usar objetos **Employee** e, portanto, **Company** poderia ter um método **getEmployees()** que retornasse o array de objetos **Employee**. Como resultado, dois objetos teriam referências apontando para o mesmo array. Isso é ruim?

Referências duplicadas apontando para os mesmos dados ocorrem com frequência em sistemas de software. No entanto, um problema é que os outros objetos podem gerar, maliciosamente ou accidentalmente, dados inválidos. No exemplo de **Company**, o que aconteceria se outro objeto modificasse o array removendo **Employees** que deveriam estar presentes ou adicionando novos objetos **Employee** ao mesmo array ou até mesmo adicionando **null**? Nesse caso, **Company** não está protegendo os dados apropriadamente contra a adulteração, seja ela intencional ou não.

O que **Company** deveria fazer em vez de disponibilizar seu array para qualquer um que pedir? Há várias opções:

1. Fazer o método **getEmployees()** retornar uma cópia do array de **Employees**.
2. Presumir que raramente outros objetos precisarão ver todos os objetos **Employee** e ter um método **get** que encontre e retorne apenas um funcionário de cada vez.
3. Fazer a classe **Company** seguir o padrão Expert e se encarregar de executar qualquer tratamento necessário nos objetos **Employee** para os outros objetos para que os **Employees** fiquem sempre ocultos. Isto é, não dar a outros objetos acesso direto a objetos **Employee** e, em vez disso, forçar esses outros objetos a pedirem a **Company** para acessar os objetos **Employee** para eles.

A questão aqui está relacionada a quanto você confia em seus colaboradores. O comportamento padrão de uma classe deve ser desconfiar de todos. Especificamente, se **Company** não confia em outros objetos, não deve deixá-los ver o array, mas apenas os dados que o compõem.

Em um bom projeto, cada conjunto de dados tem uma classe “guardiã de acesso” associada que é responsável por proteger a fonte primária desses dados. Outros objetos devem pedir à guardiã uma referência aos dados quando precisarem acessá-los. Poderão então usar temporariamente ou até mesmo modificar os dados passados para eles, se apropriado. Mas, em geral, não devem ter uma referência permanente à fonte primária dos dados, nem criar ou manter sua própria cópia deles, exceto temporariamente.

Assim como os métodos devem sempre manter os objetos em um estado bem-formado, a guardiã também deve sempre manter seus dados em estado bem-formado, o que requer que ela seja o único objeto com acesso aos seus dados.

Dito isso, às vezes a duplicação de dados é preferível a uma abordagem alternativa. Por exemplo, considere uma situação em que tivéssemos a guardiã de um enorme conjunto de objetos. Se o conjunto mudasse com pouca frequência, mas seu tamanho fosse solicitado frequentemente, faria pouco sentido percorrer o conjunto contando o número de objetos sempre que seu tamanho precisasse ser conhecido. A título de eficiência, seria melhor ter uma variável separada com o tamanho do conjunto que pudesse ser retornada quando o tamanho fosse solicitado. Essa situação viola

a diretriz contra a duplicação porque, além de o próprio conjunto conter a informação de seu tamanho (embora tenhamos que percorrê-lo para obtê-la), a variável adicional também contém essa informação, duplicando-a. No entanto, por razões de eficiência, nesse caso a duplicação é válida.

E quanto às outras formas de duplicação mencionadas, tais como a duplicação de código dentro de um método ou a duplicação de um método? Assim como na duplicação de dados, o problema da duplicação de código é que as cópias podem perder a sincronia quando uma cópia for atualizada mas as outras não. Se você tiver duplicação de código dentro de uma classe, ela pode ser removida pela extração da seção duplicada de todos os métodos e por sua transferência para um novo método auxiliar a ser chamado pelos outros métodos. Se você tiver duas classes diferentes com métodos duplicados, pode dar a uma delas uma referência à outra classe e chamar o método indiretamente por intermédio de uma instância dessa classe. Alternativamente, poderia mover o método duplicado para uma superclasse comum. No entanto, geralmente não é tão fácil remover a duplicação de código porque ela pode estar sutilmente oculta.

Como mencionado anteriormente, há outra forma de duplicação que pode causar problemas: a duplicação entre o software propriamente dito e sua documentação. Tanto a documentação externa quanto a interna devem refletir precisamente o projeto e a implementação do software porque, se ficarem fora de sincronia, a documentação causará confusão, o que pode ser pior do que não haver documentação.

A duplicação de processos também deve ser evitada por razões de eficiência. Por exemplo, seria um desperdício de recursos um método fazer um cálculo complexo e então chamar outro método que o repetisse. Nesse caso, seria melhor o primeiro método passar o resultado do cálculo para o segundo método, para que este evitasse a duplicação.

Para concluir, além de tudo que dissemos até agora sobre duplicação de dados, código, documentação e processos, é preciso lembrar-se de que o surgimento de qualquer forma de duplicação em seu software pode ser um sintoma de um problema maior. Se você encontrar muita duplicação, provavelmente deve repensar os papéis, responsabilidades e relacionamentos entre as classes e componentes envolvidos para ver se pode ser obtido um projeto melhor que elimine ou pelo menos reduza a duplicação.

Interface completa

Até esse ponto do capítulo, centralizamos nossas discussões na determinação da finalidade de cada classe. Isto é, que conceito cada classe deve representar ou que papel uma classe deve desempenhar? As seções anteriores argumentaram que cada classe deve desempenhar um e apenas um papel, para aumentar a coesão e separar as responsabilidades. Além disso, argumentaram que cada responsabilidade deve ser atribuída a exatamente uma classe.

Mas, uma vez que tivemos definido o papel de uma classe, ainda teremos que decidir que comportamentos (métodos) ela deve ter com relação a esse papel. Deveremos incluir apenas o comportamento mínimo necessário? Ou aproveitar e adicionar mais comportamentos à classe na esperança de aumentar sua reutilização? Que métodos devemos incluir para implementar esses comportamentos?

As respostas a essas perguntas variam de uma classe para outra. Algumas classes são muito específicas do aplicativo. Em particular, uma classe apenas com um método **main()** sendo executado quando o aplicativo começa a funcionar não poderá

ser reutilizada em outros aplicativos e, portanto, faria pouco sentido se esforçar muito projetando-a para reutilização. Porém, atenção e energia consideráveis devem ser dedicadas ao projeto de classes que farão parte de uma biblioteca de que muitos outros aplicativos dependerão. No resto desta seção, abordaremos esse tipo de classe.

Para classes que quisermos que tenham o maior nível de reutilização possível, há uma nova diretriz: *forneça à classe uma interface completa*. Com “completa” queremos dizer que a classe deve ter o conjunto completo de comportamentos apropriados para poder executar qualquer ação necessária relacionada ao papel que desempenha. Se, por exemplo, tivermos um componente de GUI com um método **setSelected()** sem parâmetros que o realcem, ele deve ter um método **setUnselected()** sem parâmetros para remover o realce. Seria ainda melhor se tivéssemos um método **setSelected()** com um parâmetro **boolean b** que realçasse o componente se **b** fosse **true** e removesse o realce se **b** fosse **false**. Para ficar completo, esse componente também deveria ter uma função **boolean isSelected()** para informar se ele está selecionado atualmente.

Qual deve ser o nível de preenchimento do conjunto de comportamentos? Isto é, quantos métodos públicos devem ser incluídos? Em um extremo teríamos a criação do número mínimo de métodos *essenciais*, isto é, os métodos que qualquer classe que faça uma coisa bem deve ter. Normalmente as implementações desses métodos estão intimamente ligadas à representação dos dados armazenados na classe. Informalmente, podemos considerar um método essencial aquele que não pode ser implementado com chamadas a outros métodos da classe. Chamamos todos os métodos não essenciais de métodos de *conveniência*. No outro extremo de uma classe que tem apenas com métodos essenciais teríamos uma classe com um grande número de métodos de conveniência além dos essenciais.

Por exemplo, considere uma classe **Rectangle** giratória com um método **rotate()** que use o número de graus de rotação como parâmetro. Suponhamos que um número significativo das rotações esperadas fossem rotações de 90 graus no sentido anti-horário. Então, um método **rotateLeft()** poderia ser incluído na classe **Rectangle** por conveniência. O método **rotateLeft()** não é, no entanto, essencial, já que pode ser implementado por uma chamada a **rotate(90)**.

Para concluir, o projetista tem que tomar cuidado para não assustar os usuários adicionando métodos demais a uma classe na tentativa de assegurar que ela esteja completa. Em outras palavras, não tente prever todos os usos possíveis que serão dados à sua classe. É melhor incluir um conjunto básico de métodos essenciais e o conjunto mais apropriado de métodos de conveniência e permitir que os usuários construam seus próprios métodos de conveniência ou estendam a classe com subclasse se precisarem de um conjunto de métodos maior. A chave é assegurar que o conjunto básico seja suficientemente grande para que os usuários possam criar qualquer método de conveniência de que precisarem.

Projete pensando em mudanças

Grande parte de nossa discussão até aqui teve a ver com a reutilização e a legibilidade do software. Por exemplo, queremos nomes que revelem a finalidade para que nosso código seja legível. Queremos que nossas classes tenham coesão e uma interface completa para que possam ser facilmente reutilizadas. No entanto, há outras questões igualmente importantes a se considerar no projeto de classes. São as questões de facilidade de manutenção, modificação e extensão.

Há muitas razões que tornam necessárias modificações em classes existentes, inclusive corrigir falhas, fazer otimizações e adicionar novo comportamento. A parte difícil dessas alterações é fazê-las de uma maneira que não introduza novas falhas ou torne o código “frágil”, ou seja, um código que é danificado facilmente quando novas modificações são tentadas.

Para ter um sistema de fácil manutenção, modificação e extensão, você precisa projetá-lo pensando na possibilidade de futuras modificações. A nova diretriz é: *projete suas classes de modo que elas possam lidar com mudanças*.

É mais fácil falar do que seguir essa regra. Como você pode projetar pensando em mudanças se elas podem ocorrer de várias formas? Na verdade, você não pode prever todos os tipos de mudança, mas seguir algumas diretrizes ao projetar seu código melhorará muito a capacidade das classes lidarem com a mudança quando ela ocorrer.

Uma dessas diretrizes é conhecida como *Princípio Aberto-Fechado*. Ele diz: *você deve projetar software de um modo que seja fácil estendê-lo adicionando novas classes e estendendo e reutilizando classes existentes sem precisar modificá-las*. Observe que “estender” aqui não significa apenas criar subclasses. Em vez disso, estamos sendo mais abrangentes e nos referindo a estender um sistema adicionando novos recursos ou um novo comportamento.

O benefício de seguir esse princípio deve ficar claro. Quando o software precisa ser alterado, como sempre ocorre, uma das abordagens é alterar as classes existentes. Por exemplo, você pode alterar o software adicionando novos métodos ou modificando métodos já existentes nas classes. Infelizmente, essa abordagem também costuma exigir a alteração de outras classes que dependam das classes alteradas, que por sua vez podem requerer a alteração de classes que dependam dessas classes, e assim por diante. Além de todas essas alterações em cascata serem demoradas, elas podem introduzir novos erros no código. É claro que é vantajoso reduzir a modificação de um código funcional existente e estender esse código adicionando novas classes que incorporem as alterações, não mexendo nas classes antigas.

Como projetar classes sem violar o Princípio Aberto-Fechado? Considere uma tomada elétrica de sua casa que atualmente forneça energia para um relógio elétrico que você comprou na Acme dez anos atrás. Há pelo menos três maneiras de vermos a tomada, com analogias ao projeto orientado a objetos:

1. Se pensarmos na tomada somente como uma tomada para relógios Acme (isto é, se reservarmos essa tomada apenas para relógios da marca do relógio atual), se a Acme parar de produzir relógios, a tomada perderá a utilidade quando seu relógio estragar. A tomada, nesse caso, é análoga a uma variável **AcmeClock** que só pudesse referenciar (ligar) relógios de tipo **AcmeClock**.
2. Se pensarmos na tomada como uma tomada para relógios elétricos, quando nosso relógio atual estragar ou ficar obsoleto, poderemos comprar um novo relógio elétrico de qualquer marca ou estilo e ligá-lo na tomada. A tomada, nesse caso, é análoga a uma variável de tipo **ElectricClock** (uma classe ou interface abstrata) que pudesse referenciar objetos de qualquer subclasse (marca) de **ElectricClock**. Esse modo de ver a tomada é muito mais útil do que o primeiro, mas ainda é restrito porque só relógios elétricos podem usar a tomada.
3. Se pensarmos na tomada como uma tomada para utensílios elétricos, podemos conectar nela qualquer utensílio, e não apenas um relógio. Os utensílios

não precisam ter alguma relação a não ser um plugue que se encaixe na tomada (a interface elétrica). Esse modo de ver é como se tivéssemos uma interface **ElectricAppliance** e uma variável de mesmo tipo que pudesse conter qualquer objeto de qualquer classe que implementasse essa interface.

Está claro que a terceira perspectiva fornece muito mais flexibilidade aos usuários das classes. Nesse caso, é muito mais fácil alterar o valor da variável quando necessário. A ideia é definir suas variáveis e valores com o tipo mais amplo para que possam ser usados no maior intervalo de valores possível. O tipo mais amplo possível em Java é o de interface.

Portanto, temos outra diretriz: *codifique para interfaces e não para classes*. Isto é, sempre que possível, escreva seu código de modo que os objetos sejam referenciados pelas interfaces que eles implementam e não pela classe concreta a qual pertencem.

Uma maneira de ver a codificação para interfaces, ou, para ser mais amplo, o projeto de classes que lide com mudanças, é pensar nisso como um modo de tornar fácil voltar atrás em qualquer decisão de projeto ou implementação que você tiver tornado. Por exemplo, se você programar seu código em hardware para referenciar um objeto de uma classe específica, será difícil desistir desse projeto e usar outra classe. Por outro lado, se tivesse escrito seu código de modo que o objeto fosse armazenado em uma variável de um tipo de interface e só os métodos da interface fossem usados, seria fácil substituir o objeto por um de outra classe que implementasse a interface. Pouco ou nenhum código teria que ser modificado.

Observe que muito do que discutimos nesta seção pode ser considerado uma redução da conexão entre partes do programa. Por exemplo, ao codificar para interfaces e não para uma classe concreta, você reduz a interdependência entre as classes concretas. O conceito de reduzir as conexões é chamado de “redução da vinculação”.

Também podemos considerar a interdependência em uma escala maior, como entre grupos de classes em vez de apenas entre duas classes. Por exemplo, suponhamos que decidíssemos ser preciso substituir um grupo de classes que funcionam em conjunto para executar uma tarefa por outro grupo que execute a mesma tarefa. Se as classes fossem projetadas de modo que cada classe tivesse um nível mínimo de vinculação com as outras classes, seria mais fácil fazer a mudança.

Duas técnicas que podem ajudá-lo a projetar prevendo mudanças são o encapsulamento e a ocultação de informações. Como vimos no Capítulo 1, *encapsular* é agrupar itens relacionados e isolá-los ou protegê-los do acesso externo. Embora o encapsulamento possa ser obtido na maioria das linguagens de programação, linguagens orientadas a objetos como Java fornecem um mecanismo natural, a classe, para encapsular dados e os métodos que operam com eles. Java também fornece um mecanismo de pacote para o encapsulamento de grupos de classes.

Como os itens encapsulados podem ser protegidos? Uma técnica útil é a *ocultação de informações*, ou seja, manter as informações ocultas de outras pessoas. Especificamente, é útil manter a implementação de uma classe ou grupo de classes oculta dos usuários da classe ou das classes o máximo possível. Se as informações de uma classe forem expostas, os clientes podem ficar dependentes dessa exposição, o que resultaria em uma vinculação maior entre essa classe e as classes dos clientes. Uma maneira de ocultar informações já foi mencionada aqui: manter todas as variáveis de instância privadas. Essa privacidade é importante para assegurar que os objetos de

uma classe estejam sempre em um estado bem-formado. Também é necessária se, no futuro, quisermos alterar a implementação de uma classe sem ser preciso que outras classes mudem. Se outras classes tiverem acesso a essas variáveis de instância, forneça métodos *get* e *set* para cada uma delas em vez de tornar as variáveis públicas. Uma vantagem do uso desses métodos em vez de tornar as variáveis públicas é que podemos incluir um código no método *set* que assegure que o objeto permaneça em um estado bem-formado. Por exemplo, poderíamos incluir uma verificação de intervalo. Além disso, esses métodos podem ser sobrepostos por subclasses, o que dá aos desenvolvedores mais flexibilidade para criar classes que atendam suas necessidades. Por outro lado, uma variável de instância pública não pode ser sobreposta em subclasses.

Resumindo, variáveis de instância privadas e, se necessário, métodos *get* e *set* públicos, são quase sempre preferíveis a variáveis de instância públicas.

Quanto da implementação de uma classe devemos encapsular e ocultar? A regra geral é encapsular e ocultar o máximo possível. Por exemplo, torne privados todos os métodos que forem usados apenas internamente na classe como métodos auxiliares de outros métodos. É muito mais fácil expor algo posteriormente que não precise mais ficar oculto do que ocultar algo que já tenha sido exposto. A ocultação de um item que já foi exposto pode requerer mudanças globais no sistema.

Em resumo, uma das coisas mais importantes que você pode fazer para tornar seu sistema elegante é projetar pensando em mudanças. O Princípio Aberto-Fechado declara mais especificamente o que devemos perseguir. Codificar para interfaces, reduzir a vinculação e usar o encapsulamento e a ocultação de informações são diretrizes que ajudam a alcançar esse objetivo.

Lei de Demeter

Há um último tópico que queremos abordar nesta seção. Ele está relacionado ao padrão Expert, à vinculação entre classes e à facilidade com que seu software pode lidar com mudanças. Esse tópico é a *Lei de Demeter*.

Para entender a Lei de Demeter, considere o general de um exército que estivesse estabelecendo uma base de operações no campo. Uma das muitas tarefas que têm de ser executadas é cavar trincheiras. Aqui está uma maneira de a tarefa ser executada: o general poderia chamar um de seus coronéis e pedir a ele que chamasse um major; ele pediria então ao major que chamasse um capitão; depois, pediria ao capitão que chamasse um sargento; o general pediria então ao sargento para chamar um soldado; só aí pediria que o soldado cavasse algumas trincheiras.

Essa abordagem é bem absurda, não? Mas podemos facilmente ver o equivalente em código:

```
general.getColonel(c).getMajor(m).getCaptain(c).
    getSergeant(s).getPrivate(p).digFoxhole();
```

Espero que você tenha percebido que, da mesma forma que não é apropriado o general percorrer todas essas etapas para cavar uma trincheira, é inapropriado o código percorrer essa cadeia de mensagens para uma tarefa ser executada.

Considere uma cadeia semelhante, porém mais realista, que poderia aparecer em um aplicativo ATM, em que o programa lesse, em uma placa ATM, o nome de um banco **b**, o número da filial **r**, o nome do cliente **c** e o número da conta **a**, e tentasse, começando com algum objeto **CentralControl** associado, obter o saldo do cliente nesta conta:

```
| Balance balance = centralControl.getBank(b).getBranch(r).  
|   getCustomer(c).getAccount(a).getBalance();
```

Detectou os problemas desse código? Um deles é que agora vinculou rigidamente as classes **CentralControl**, **Bank**, **Branch**, **Customer**, **Account** e **Balance** do aplicativo ATM. Qualquer alteração na estrutura de uma das últimas classes poderia afetar o código de todas as classes anteriores. Igualmente importante é o fato de os métodos *getX* darem ao usuário acesso aos objetos **Bank**, **Branch**, **Customer**, **Account** e **Balance**; portanto, ele poderia manipulá-los (isto é, chamar métodos neles) de maneiras que podem não ser as pretendidas para alguém que não pertença a um subgrupo privilegiado.

Como essas situações podem ser tratadas mais adequadamente? Voltemos ao general do exército e consideremos o que ele faria realmente. O mais provável é que não se preocupasse com detalhes como cavar trincheiras. Mas, caso se preocupasse, certamente chamaria um de seus oficiais e diria: “Não quero saber como será feito, mas arrume alguém para cavar trincheiras”.

Da mesma forma, uma maneira possível de tratar a solicitação de um saldo de conta seria fazer algo assim:

```
| Balance balance = centralControl.getBalance(b, r, c, a);
```

Nessa versão, é incumbência da classe **CentralControl** delegar ou encaminhar as responsabilidades para outras classes conforme necessário para obter o saldo desejado. Por exemplo, **CentralControl** poderia encontrar o banco de nome **b**, passar para ele os valores de **r**, **c** e **a** e pedir o saldo. O banco daria prosseguimento de maneira semelhante. Como resultado, as classes estariam menos rigidamente vinculadas. O aplicativo principal não precisa se preocupar com a existência de objetos **Customer** ou **Account** e pode deixar esses detalhes para **CentralControl** ou para outras classes com as quais **CentralControl** se comunicar.

O encadeamento de métodos visto nesses exemplos é uma violação da *Lei de Demeter*, que diz: *só chame métodos em objetos que você mesmo criar ou aos quais tiver acesso direto*. Informalmente, a lei diz: *não fale com estranhos*. Em particular, não fale com (isto é, chame métodos em) objetos que sejam retornados por métodos chamados em outros objetos.

Aqui estão mais alguns exemplos em que vemos essa questão:

1. Empreiteiros em geral, quando constroem uma casa ou uma estrutura maior, costumam contratar subempreiteiros. Os subempreiteiros, por sua vez, podem contratar outros subempreiteiros para ajudá-los. O empreiteiro principal não se preocupa com o segundo nível de subempreiteiros. Isto é, espera que o primeiro nível execute as tarefas para as quais foi contratado sem se preocupar com as subcontratações ou com o modo como a tarefa será concluída.
2. Suponhamos que você estivesse acessando o banco de dados de um banco e precisasse encontrar um cliente com um nome específico. Poderia (a) solicitar ao banco de dados uma lista de todos os clientes e então percorrê-la procurando o nome desejado, (b) solicitar ao banco de dados uma lista de todos os clientes e pedir à lista que encontrasse o cliente desejado para você ou (c) fornecer ao banco de dados o nome e lhe pedir que fizesse uma busca para encontrar o cliente. A versão (c) é a melhor a partir da perspectiva da Lei de Demeter.

Também é importante mencionar aqui que, na verdade, a Lei de Demeter deveria se chamar “Diretriz” em vez de “Lei”. Ela não é algo absoluto que deva sempre ser seguido. Caso contrário, colocaria uma carga tremenda sobre os objetos aos quais temos acesso direto, porque todas as solicitações possíveis de qualquer outro objeto teriam que ser tratadas por intermédio deles. Como nem sempre sabemos que tipos de solicitações serão feitas (lembre-se, “mudanças acontecem”), é difícil prever todas as futuras solicitações.

Verificação do progresso

1. O que diz o padrão Expert?
2. O princípio *Evite a duplicação* só se aplica a dados. Verdadeiro ou falso?
3. O que diz o princípio Aberto-Fechado?
4. O que a Lei de Demeter diz informalmente?

HERANÇA VERSUS DELEGAÇÃO

No projeto de um sistema grande, é muito importante considerar o papel desempenhado pela herança, um dos recursos mais significativos da programação orientada a objetos. A herança pode aumentar muito a reutilização de classes e também reduzir a duplicação de código. No entanto, se usada inapropriadamente, pode diminuir bastante a qualidade do projeto. Nesta seção, abordaremos a questão de como usar a herança corretamente.

Diagramas de classes UML

Para ajudar em nossa discussão do uso apropriado da herança, pode ajudar termos um diagrama mostrando os relacionamentos entre as classes de um projeto. Logo, antes de começar a discussão, introduziremos os diagramas de classes UML. A UML (que significa “Unified Modeling Language”) é uma notação ou linguagem padrão para a diagramação da estrutura de um sistema de software. Ela independe da linguagem de programação usada na implementação do sistema. O *diagrama de classes* UML é um entre os mais de uma dúzia de diagramas da notação UML. Ele exibe classes e interfaces e os relacionamentos entre elas de uma maneira independente da linguagem. Tal diagrama fornece uma visão estática de classes, interfaces e seus relacionamentos, em vez de uma visão dinâmica das interações entre objetos dessas classes.

Uma classe é representada por uma caixa dividida em três seções. A seção do topo fornece o nome da classe. A seção do meio fornece os atributos ou propriedades mantidos pelos objetos da classe. Essas propriedades são abstrações dos dados ou do estado de um objeto e geralmente são implementadas com o uso de variáveis de

Respostas:

1. O objeto que contém os dados necessários à execução de uma tarefa deve ser o que a executará.
2. Falso.
3. Projete software que seja aberto à extensão, mas fechado à modificação.
4. Não fale com estranhos. Fale apenas com seus vizinhos imediatos.

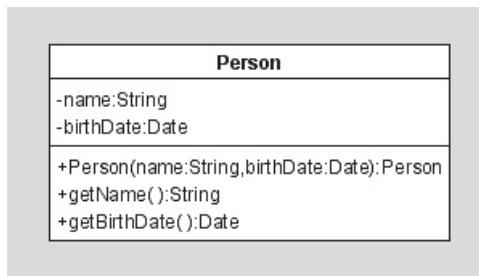


Figura 16-1 O diagrama de uma classe **Person**.

instância. A seção inferior fornece as operações de uma classe, que em Java correspondem aos métodos e construtores.

Por exemplo, consulte a Figura 16.1 para ver o diagrama de uma classe **Person**. O diagrama exibe que essa classe tem um atributo **name** que é um string e um atributo **birthDate** de tipo **Date**. Em Java, esses dois atributos poderiam ser implementados com a inclusão de duas variáveis de instância na classe **Person**: uma variável chamada **name** de tipo **String** e uma variável chamada **birthDate** de tipo **Date**. O diagrama também mostra que essa classe tem um construtor com dois parâmetros, um método **getName()** sem parâmetros que retorna um **String** e um método **getBirthDate()** sem parâmetros que retorna um **Date**.

Várias outras coisas podem ser observadas nesse diagrama:

1. O símbolo “–” na frente dos atributos indica que eles são privados. Da mesma forma, o símbolo “+” na frente das operações indica que elas são públicas.
2. Os tipos dos atributos, parâmetros e tipos de retorno das operações são fornecidos após os nomes do elemento e separados deles por dois pontos.
3. A implementação desses atributos e operações não é incluída na caixa porque geralmente um diagrama UML não se preocupa com um nível tão baixo de detalhes.

Qualquer uma das partes de uma classe, exceto seu nome, pode ser omitida do diagrama. Para evitar desorganização e confusão, é fortemente recomendado que o diagrama deixe de fora todos os detalhes que não sejam relevantes à discussão. Portanto, é importante lembrar que, se as operações de uma classe não foram mostradas, por exemplo, isso não quer dizer que ela não tenha operações; quer dizer que as operações não são relevantes nesse diagrama.

Os diagramas de classes UML não exibem apenas as classes e interfaces, mas também os relacionamentos entre elas. A notação UML para as subclasses é mostrada na Figura 16.2. A seta que aponta da subclasse para a superclasse se chama relacionamento de *generalização* em UML.

Uma *associação* é um relacionamento estrutural entre duas classes. Se objetos de uma classe tiverem uma referência a objetos de outra classe, ou se você precisar navegar de objetos de uma classe para objetos de outra, poderia representar a conexão entre as classes com uma associação, que é desenhada em diagramas de classes UML como uma linha entre as caixas das classes. Consulte a Figura 16.3 para ver um exemplo. Nesse diagrama, a associação é implementada por um atributo **x** de **A** de tipo **B**.

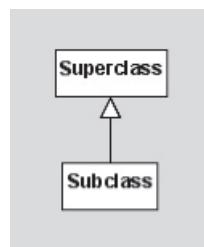


Figura 16-2 A notação UML para a subclasse e a superclasse.

As linhas das associações podem ter muitos complementos opcionais, como números em uma ou nas duas extremidades, indicando multiplicidade (o número de objetos dessa extremidade da associação). Também podem ter uma seta em uma extremidade, como mostrado na figura, para indicar naveabilidade ou reconhecimento unidirecional. De objetos da parte de trás da seta, você pode “reconhecer” ou “chegar a” objetos da classe da ponta da seta. Uma linha reta sem setas pode indicar naveabilidade bidirecional ou que a direção da naveabilidade não é importante e, portanto, foi deixada de fora.

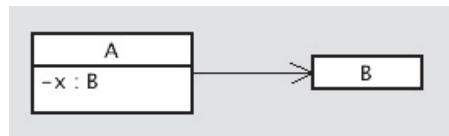


Figura 16-3 Uma associação entre A e B.

Pergunte ao especialista

P O que fazem todos os outros diagramas UML?

R A UML 2.2 tem 14 diagramas, que podem ser divididos em duas categorias gerais: diagramas estruturais e diagramas comportamentais. O diagrama de classes se enquadra no primeiro grupo. Não faz parte do escopo deste livro a discussão de todos os diagramas. Em vez disso, mencionaremos apenas três deles aqui. O diagrama de *casos de uso* é um diagrama comportamental útil na representação e gerenciamento de todos os usos a que o software será aplicado. Por exemplo, um aplicativo gerenciador de música digital pode ser usado na reprodução e importação de música, na criação e edição de listas de reprodução e na cópia de música em um mp3 player. Todos esses usos seriam representados em um diagrama de caso de uso. O diagrama UML de *sequência* é um diagrama comportamental usado na exibição das interações entre os objetos ao longo de uma linha de tempo. O diagrama UML de *transição de estados* é um diagrama comportamental que representa como o estado de um objeto ou sistema muda com o tempo. Por exemplo, considere o software de um relógio digital de baixo custo. Inicialmente, ele pode exibir a hora atual. Mas, após pressionarmos os botões corretos, exibirá um cronômetro ou timer. Agora o software do relógio estará em um novo estado. Se o relógio estiver no modo cronômetro e pressionarmos o botão correto, o tempo começará a correr, o que é mais um estado do software. O diagrama de transição de estados mostra como todos esses estados estão relacionados e que eventos precisam ocorrer para passarmos de um estado para outro.

Possibilidade de reutilização do código

Agora retornemos à discussão do uso apropriado da herança. Um dos benefícios reais da herança é a reutilização de código. Mas somente a reutilização de código é razão suficiente para o uso da herança?

Consideremos um exemplo. Suponhamos que seu projeto incluísse uma classe **Dog** e uma classe **Person**, as duas tendo uma variável de instância **name** e um método **getName()**. Para evitar a duplicação desse campo e do método, você poderia remover a variável de instância e o método da classe **Person** e torná-la subclasse da classe **Dog**. Consulte a Figura 16.4 para ver um diagrama UML exibindo esse relacionamento.

Infelizmente, em Java, as subclasses herdam todos os elementos de suas superclasses. Logo, a classe **Person** também herdaría os métodos **bark()** e **getLastRabiesShotDate()**, que são um pouco inapropriados para objetos **Person**.

Alternativamente, você poderia inserir a variável de instância **name** e o método **getName()** na classe **Person** e tornar a classe **Dog** subclasse de **Person**. É claro que essa solução é igualmente inapropriada.

Em situações como essas, é melhor reutilizar apenas parte do código. Se uma subclasse pudesse escolher seletivamente que código herdar ou não, um argumento a favor do uso da herança teria mais mérito.

Qual seria uma maneira melhor de evitar a duplicação de código nas classes **Dog** e **Person**? Uma maneira seria criar uma classe **NamedObject** com o campo **name** e o método **getName()** e tornar tanto **Dog** quanto **Person** subclasses de **NamedObject**. Essa abordagem funcionará bem se não tivermos outras superclasses de **Dog** e **Person**. (Lembre-se, em Java, cada classe pode ter no máximo uma superclasse.)

Resumindo, poder herdar código que de outra forma teria que ser duplicado é um recurso útil das linguagens orientadas a objetos. Mas apenas a reutilização do código raramente justifica a herança.

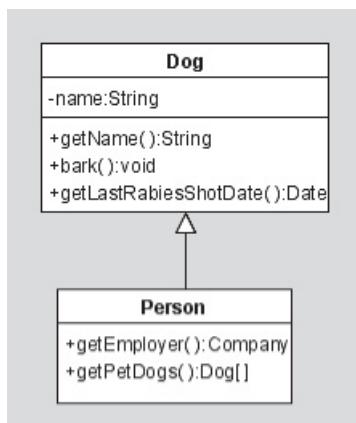


Figura 16-4 Mau uso da herança.

O relacionamento É-um

O problema do uso da herança para as classes **Dog** e **Person** na seção anterior é que cães não são pessoas e vice-versa. Isto é, não há um relacionamento *é-um* entre os conceitos que as classes modelam. A combinação de reutilização de código e o relacionamento *é-um* entre as classes é razão suficiente para o uso de subclasses?

Consideremos outro exemplo. Suponhamos que um projetista de softwares precisasse modelar formas geométricas e, portanto, criasse uma classe **Square** e uma classe **Rectangle**. A herança deve ser usada entre essas duas classes? Esse é um caso óbvio em que, do ponto de vista geométrico, todo quadrado “é um” retângulo. Além disso, certamente há boas oportunidades de haver reutilização de código entre eles. Por exemplo, as implementações de métodos que envolvam a movimentação ou busca da área ou perímetro dos retângulos e quadrados provavelmente serão idênticas nas duas classes. Logo, parece natural tornar a classe **Square** subclasse da classe **Rectangle**. Mas essa é uma decisão adequada?

Considere a implementação Java a seguir das classes **Square** e **Rectangle** usando herança. Observe como há alta reutilização de código – a classe **Square** não precisa implementar nada além de um construtor.

```
public class Rectangle {

    private int x, y, width, height;

    public Rectangle(int x, int y, int w, int h) {
        this.x = x; this.y = y; width = w; height = h;
    }

    public int getWidth() { return width; }

    public int getHeight() { return height; }

    public int getArea() { return width * height; }

    public int getPerimeter() { return 2 * (width + height); }

    public void setTopLeft(int newx, int newy) { x = newx; y = newy; }

    public void setSize(int w, int h) { width = w; height = h; }
}

public class Square extends Rectangle
{
    public Square(int x, int y, int side) {
        super(x, y, side, side);
    }
}
```

No entanto, como você deve ter notado, há um problema sério nesse projeto. Devido ao fato de as subclasses herdarem todos os métodos de suas superclasses, agora a classe **Square** herda um método **setSize()** que tem dois parâmetros. Uma chamada a esse método pode tornar desiguais a largura e a altura de um quadrado, um resultado

bastante indesejável. Um método **setSize()** para quadrados deveria usar apenas um parâmetro.

O que podemos fazer nessa situação? Antes de descartar a herança, tentemos corrigir o problema por outros meios. Uma maneira de fazê-lo é anular os efeitos negativos do método **setSize()** herdado sobrepondo-o na subclasse. Por exemplo, poderíamos adicionar o método a seguir à classe **Square**

```
public void setSize(int w, int h) { width = w; height = w; }
```

que ignora o parâmetro **h** e usa **w** tanto para a altura quanto para a largura. Esse método permite que os usuários modifiquem o tamanho do quadrado, mas só de uma maneira que preserve suas propriedades.

Infelizmente, essa solução não é muito boa. Para vermos o porquê, suponhamos que o usuário adicionasse o método abaixo ao seu programa:

```
public int stretchAndFetch(Rectangle r, int dx) {
    r.setSize(r.getWidth() + dx, r.getHeight());
    return r.getHeight();
}
```

Quando esse método for executado, o usuário espera que um retângulo seja estendido horizontalmente ao longo do espaço **dx**. Mas se, sem conhecimento do usuário, um objeto **Square** for passado como primeiro argumento, a altura também será ajustada de acordo com **dx** e a nova altura será retornada pelo método. O usuário ficará surpreso ou confuso ao ver essa nova altura porque esperava que só houvesse prolongamento horizontal. Um código não é considerado elegante se o usuário fica surpreso ou confuso com seu comportamento.

Nosso problema aqui é que a subclasse não tem um comportamento coerente com o comportamento de sua superclasse. O comportamento coerente é necessário para termos um código elegante. Essa diretriz também pode ser expressa em termos de perplexidade: *se um cliente achar que tem uma referência a um objeto de tipo A, mas na verdade tiver uma referência a um objeto de subtipo B, não devem surgir surpresas quando ele chamar métodos no objeto*. A diretriz se chama *princípio da menor perplexidade*.

Por exemplo, é quase certo que um cliente usando um objeto **Square** como valor de **r** no método **stretchAndFetch()** acima fique surpreso com seu comportamento, indicando um problema na hierarquia de herança.

Quais são os efeitos de seguirmos o princípio da menor perplexidade? A resposta depende do que consideramos “surpresa”. Há duas definições dessa palavra que poderíamos usar nesse contexto. No caso mais restritivo, uma surpresa é qualquer diferença nos comportamentos dos objetos de duas classes. Se usarmos essa definição, então, pelo princípio da menor perplexidade, não poderemos alterar nenhum dos comportamentos herdados na subclasse, ou seja, não poderemos sobrepor métodos da superclasse. Logo, as subclasses só podem adicionar um novo comportamento em vez de modificar um existente. O novo comportamento não será visível quando objetos da subclasse estiverem sendo tratados como se fossem do tipo da superclasse. Um nome mais adequado para essa versão restritiva do princípio seria *princípio da não perplexidade*.

Uma definição menos restritiva de surpresa seria dizer que uma surpresa é qualquer diferença no comportamento *documentado* de objetos das duas classes. Se a documentação externa do método **stretchAndFetch()** informar que só a largura é pro-

longada, o comportamento desse método quando um quadrado for o argumento violará as diretrizes. No entanto, se a documentação não fizer menção sobre o comportamento do método com relação à altura do retângulo, o comportamento com quadrados não violará a diretriz. Nesse caso, é perfeitamente aceitável o quadrado ser alongado tanto horizontal quanto verticalmente e o princípio da menor perplexidade será obedecido.

Você deve ter notado outro aspecto do exemplo do quadrado/retângulo. O relacionamento *é-um* original entre quadrados e retângulos geométricos, como considerado pelos matemáticos, só ocorre porque, em matemática, todas as formas geométricas são fixas, ou imutáveis. Se um matemático falar no alongamento de um retângulo, com certeza estará falando na criação de um novo retângulo e não na modificação do existente. Logo, se tornássemos a classe **Rectangle** e a classe **Square** imutáveis, elas se aproximariam mais da ideia que os matemáticos fazem de retângulos e quadrados. Na verdade, se tornássemos as classes imutáveis removendo os métodos **setTopLeft()** e **setSize()**, seria perfeitamente aceitável tornar a classe **Square** subclasse da classe **Rectangle**.

Comportamento semelhante

Como mencionado na seção anterior, o problema do relacionamento *é-um* entre as classes **Square** e **Rectangle** originais era o fato de os comportamentos de **Square** e **Rectangle** não coincidirem. Ou seja, um **Square** não será um **Rectangle** se um **Rectangle** puder modificar sua largura separadamente de sua altura. Outra maneira de dizer isso seria que nem todos os métodos públicos da classe **Rectangle** eram apropriados para a classe **Square**. Mas o que aconteceria se tivéssemos interfaces públicas semelhantes com comportamentos parecidos entre duas de nossas classes, além da reutilização de código e de um relacionamento *é-um*? Agora temos uma razão suficiente para usar subclasses?

Considere um exemplo clássico usado com frequência na introdução do conceito de herança, as classes **Student** e **Person**. É claro que um aluno “é uma” pessoa (contanto que esteja frequentando uma escola de ensino fundamental ou médio ou uma faculdade, por exemplo, e não um curso de adestramento). Um aluno, como a pessoa, tem todas as propriedades e comportamentos de uma pessoa, como nome, endereço e data de nascimento, e, além disso, tem outros atributos, como a escola em que está matriculado atualmente, as notas conquistadas, a média de pontos obtida e o cronograma de aulas. O aluno e a pessoa certamente têm comportamentos em comum e, portanto, há oportunidade de reutilização de código.

Seria natural você argumentar que a classe **Student** deve ser subclass de **Person**. Mas deve mesmo? Suponhamos que incluíssemos essa herança em nosso projeto e considerássemos uma universidade maior que usasse o projeto para armazenar seus registros de cada aluno. Além disso, suponhamos que a universidade armazenasse seus registros de funcionários em objetos **Employee** (onde, se seguirmos a mesma linha de raciocínio, **Employee** é outra subclasse de **Person**). Consulte a Figura 16.5.

Agora, suponhamos que um dos alunos se formasse, começasse a trabalhar na universidade e, portanto, se tornasse um funcionário. Ou suponhamos que o aluno se tornasse um funcionário da universidade enquanto ainda fosse aluno. O que a universidade deve fazer para atualizar seus registros nesse caso?

No caso de um aluno se formando e se tornando funcionário, a universidade poderia simplesmente substituir o objeto **Student** por um objeto **Employee**. No entanto, essa abordagem tem dois problemas em potencial. Se houver muitas referências ao

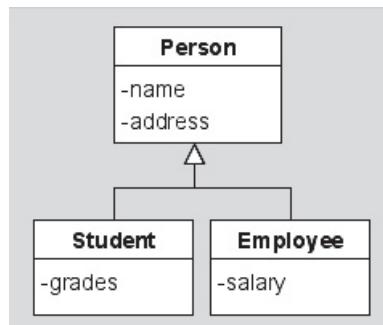


Figura 16.5 Uma hierarquia de herança entre **Person**, **Student** e **Employee**.

objeto **Student** nos registros da universidade, todas essas referências precisarão ser atualizadas para referenciar o novo objeto **Employee**. Além disso, pode haver dados preservados no objeto **Student**, como o histórico do aluno, que não foi incluído no objeto **Employee**.

Outra alternativa seria a universidade criar dois objetos distintos para representar a pessoa: um objeto **Employee** ativo e um objeto **Student** inativo que pudesse ser arquivado. Porém, essa abordagem é deselegante porque duplica todos os dados comuns de **Person** nos dois objetos.

Seria fácil resolver os problemas se fosse possível alterar um objeto dinamicamente para que pertencesse a uma classe diferente (e, portanto, transformar o objeto **Student** em um objeto **Employee**). Mas essa alteração não é possível em Java.

Uma abordagem melhor evitaria a herança e consideraria o aluno e o funcionário como *papéis* sendo desempenhados pela pessoa. Normalmente os papéis são aspectos temporários de um objeto. Em nosso exemplo, uma pessoa não é sempre um aluno ou um funcionário e, portanto, esses papéis são temporários. Devido à natureza temporária dos papéis e à natureza mais permanente da pessoa, a herança não é uma maneira adequada de associá-los.

Um projeto mais adequado usa a *delegação* (também chamada de *referência*), em que um objeto **Student** tem um objeto **Person** subjacente que ele referencia. Nesse caso, poderíamos ter exatamente um objeto **Person** para cada pessoa real, que seria independente de todos os papéis desempenhados atualmente por essa pessoa.

Por exemplo, a classe **Person** poderia ser definida assim:

```

public class Person {
    private String name;
    private String address;

    public String getAddress() { return address; }

    //...outros métodos e dados...
}
  
```

Então, à medida que novos papéis fossem desempenhados, objetos representando-os, como **Student** e **Employee**, poderiam ser criados incluindo uma referência ao objeto **Person** responsável pelo desempenho dos papéis.

Por exemplo, poderíamos definir a classe **Student** dessa forma:

```
public class Student {
    private Person me;
    private AcademicRecord myRecord;

    public String getAddress() { return me.getAddress(); }

    public float getGPA() {
        //...calcula a partir do registro acadêmico...
    }

    //...outros métodos e dados...
}
```

Observe como qualquer comportamento específico de **Person** requerido de um aluno é executado com o encaminhamento da solicitação ao objeto **Person**. Por exemplo, se forem solicitados endereços aos objetos **Student**, esses objetos delegarão a tarefa para os objetos **Person** subjacentes. O objeto **Student** trata de tarefas específicas dos alunos, como o cálculo da média das notas (GPA, grade point average) e o gerenciamento de cronogramas de cursos, e o objeto **Person** trata de todas as tarefas pessoais.

Da mesma forma, a classe **Employee** poderia ter uma referência à pessoa que está sendo empregada. Consulte a Figura 16.6 para ver um diagrama dessas alternativas usando as classes **Student**, **Employee** e **Person**.

Com o uso da delegação de responsabilidades, ficou fácil acompanhar a mudança de papéis de uma pessoa. Inicialmente, quando a pessoa se torna um aluno, tanto um objeto **Student** quanto um objeto **Person** são criados, e o objeto **Student** recebe uma referência ao objeto **Person**. Se e quando o aluno se tornar um funcionário, um novo objeto **Employee** será criado referenciando o mesmo objeto **Person**. Nessa situação, é perfeitamente aceitável termos objetos **Student** e **Employee** referenciando o mesmo objeto **Person**. Quando a pessoa não desempenhar mais um dos papéis (por exemplo, se o aluno se formar ou deixar de trabalhar para a universidade), esse papel poderá ser excluído ou arquivado e os outros papéis permanecerão ativos. Portanto, podemos considerar que o objeto **Person** existirá permanentemente, mas os papéis da pessoa podem ir e vir. Além disso, não há duplicação de dados.

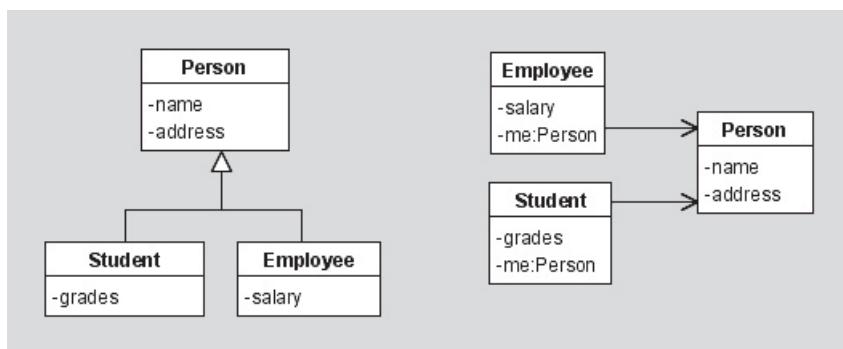


Figura 16-6 Herança (esquerda) versus delegação (direita).

A diretriz a seguir resume essa discussão: *se a classe B modelar um papel desempenhado pela classe A, principalmente um papel temporário, B não deve ser subclasse de A. Em vez disso, os objetos da classe B devem ter referências a objetos da classe A.*

Polimorfismo

E se tivermos uma situação em que há reutilização de código, relacionamento é-um e comportamentos públicos semelhantes entre as classes, e precisarmos do polimorfismo? Agora temos uma razão suficiente para usar subclasses? Provavelmente.

Por exemplo, suponhamos que você quisesse criar uma nova classe **B** para ser usada como argumento de um método cujo parâmetro fosse de tipo **A**, em que **A** é uma classe. Nesse caso, não teria escolha a não ser tornar **B** subclasse de **A**.

Essa situação ocorre com mais frequência do que parece. Por exemplo, qualquer método para o qual não importe o tipo de objeto passado como argumento terá um parâmetro de tipo **Object**. Como todas as outras classes estendem automaticamente **Object**, o polimorfismo está sendo usado aqui.

Normalmente classes abstratas também envolvem polimorfismo. Se você quiser utilizar os recursos desse tipo de classe, deve estendê-la.

Observe que, em um sistema bem projetado em que a diretriz *codifique para interfaces* tiver sido seguida, o maior número possível de variáveis e parâmetros terá um tipo de interface em vez de um tipo de classe, o que reduz a necessidade de subclasses. Mas nem sempre é possível ou apropriado usar interfaces em todos os locais. Por exemplo, o pacote Swing de Java, discutido na próxima parte deste livro, é uma estrutura de GUI em que grande parte do difícil trabalho de lidar com botões, janelas e menus já foi implementada para o usuário em vários pacotes. Muitas das classes estendem outras classes dos pacotes para lhes dar a funcionalidade necessária e facilitar o máximo possível para o usuário a construção de um sistema a partir dos componentes fornecidos.

Custos da herança

A essa altura, você já deve ter percebido que a herança, mesmo parecendo um recurso excelente à primeira vista, não deve ser usada de qualquer maneira se quisermos um sistema bem projetado. Há outros problemas a serem considerados no uso da herança.

Um problema da herança, principalmente em uma árvore de herança longa com muitas gerações, é que o código do método de uma classe mais baixa da árvore pode estar espalhado entre todos os seus ancestrais de níveis mais altos, o que dificulta para o leitor do código seguir o fluxo de execução. Por exemplo, suponhamos que alguém estivesse lendo um código e visse um método **getName()** ser chamado em um objeto. Se a classe do objeto não implementar **getName()**, então, para encontrar a implementação desse método, o leitor terá que examinar a superclasse imediata do objeto. Se essa classe não implementar **getName()**, uma busca adicional subindo a hierarquia de herança terá que ser feita. Para complicar, **getName()** poderia chamar outro método, digamos **getPerson()**, no mesmo objeto. Não é preciso que haja um relacionamento maior entre os locais de **getName()** e **getPerson()** na árvore de herança e, portanto, novamente o leitor terá que começar a busca na classe do objeto e subir a árvore para encontrar a implementação de **getPerson()**. Tudo piora se o leitor não souber qual é a classe do objeto e só souber que ele pertence a uma entre várias

subclasses de uma determinada classe. Nesses casos, é extremamente difícil, quando não impossível, descobrir o corpo de método que será executado em um determinado momento e a que classe ele pertence.

Outro problema da herança é que todas as subclasses ficam fortemente vinculadas às suas superclasses. Essa vinculação vem do fato de que, para garantirmos um comportamento específico em uma subclasse, ela tem que conhecer partes significativas da implementação dos métodos das superclasses. No entanto, os detalhes desse problema não pertencem ao escopo deste texto.

Devido a esses problemas e a todas as outras razões mencionadas nesta seção, o projetista de software deve sempre considerar cuidadosamente todas as opções ao definir os relacionamentos entre as classes. Por exemplo, há situações em que duas classes podem ser associadas via uma interface comum, em outras devem ser conectadas por uma associação como a delegação e há momentos em que a herança é a melhor opção.

TENTE ISTO 16-1 Uma fila pura implementada com o uso da delegação

`PureQueue.java`
`PureQueueDemo.java`

Este projeto demonstra como usar a delegação em vez da herança para implementar uma fila pura.

Como já vimos neste livro, uma pilha é uma lista em que os elementos só podem ser acessados na ordem “último a entrar, primeiro a sair” (LIFO, last-in, first-out). Uma *fila* é uma estrutura de dados semelhante exceto por seus elementos só poderem ser acessados na ordem “primeiro a entrar, primeiro a sair” (FIFO, first-in, first-out). As filas são usadas em vários locais, por exemplo, nos balcões de check-in dos aeroportos e nos caixas dos supermercados. Elas são usadas em computadores de diversas maneiras, como no gerenciamento de processos. Se houver muitos processos competindo por uma CPU, eles serão alinhados em uma fila. Normalmente a operação de adicionar um elemento ao fim da fila é chamada de *enfileirar* e a de remover e obter o elemento do início da fila de *retirar da fila*.

Suponhamos que estivéssemos trabalhando em uma instalação industrial e precisássemos de uma fila. Além disso, queremos que a fila seja “pura”, ou seja, ela só deve dar suporte às operações de fila que acabamos de descrever. Isso impedirá que os elementos sejam acessados fora de ordem. Em nossa fila pura, queremos que esses métodos se chamem `enqueue()` e `dequeue()`. Também queremos que nossa fila tenha um método `size()` que retorne o número de elementos da fila e um método `peek()` que obtenha o elemento do início da fila, mas sem removê-lo. Por conveniência, uma sobreposição de `toString()` é desejada. Como podemos criar essa fila? Há pelo menos três maneiras de proceder.

Uma é implementar uma classe **PureQueue** “a partir do zero”, por exemplo, usando um array para conter os dados, semelhante à maneira como **SimpleStack** foi implementada em vários exemplos Tente isto de capítulos anteriores. Por ra-

zões educacionais, foi útil criar essa classe de pilha porque ela demonstrou muitos aspectos-chave de Java. No entanto, um princípio importante do desenvolvimento de software no mundo real é *não reinvente a roda*. Em outras palavras, se houver uma classe que já faça o que você quer, não crie uma nova para fazê-lo. Em vez disso, reutilize a classe existente. Por acaso já há uma classe **ArrayDeque** no pacote **java.util** da biblioteca Java que faz tudo que queremos que nossa classe **PureQueue** faça e muito mais. Se seguirmos o princípio mencionado acima, faz mais sentido usar essa classe em vez de implementar nossa classe **PureQueue** a partir do zero.

Logo, uma segunda maneira de proceder é criar uma classe **PureQueue** que herde os elementos de **java.util.ArrayDeque**. No entanto, a classe **ArrayDeque** não é uma fila FIFO pura. Por exemplo, ela tem métodos como **push()** e **pop()** que permitem que seja usada como uma pilha. Se **PureQueue** fosse subclasse de **java.util.ArrayDeque**, herdaria mais do que apenas os cinco métodos desejados. Além disso, a classe **ArrayDeque** chama os métodos **enqueue** e **dequeue** de “offer” e “poll”, que não são os nomes que queremos usar.

Uma terceira alternativa é usar a delegação. Isto é, criar uma classe **PureQueue** que mantenha uma referência a um **ArrayDeque** do pacote **java.util**. Dessa forma, a classe **PureQueue** poderá ter apenas o conjunto de métodos desejado, mas os implementará delegando todo o trabalho para a classe **ArrayDeque** referenciada por ela. Usaremos essa terceira abordagem no projeto.

PASSO A PASSO

- Crie um arquivo chamado **PureQueue.java** e digite o código a seguir:

```
import java.util.ArrayDeque;

class PureQueue<E> {
    private ArrayDeque<E> data;

    PureQueue() {
        data = new ArrayDeque<E>();
    }

    void enqueue(E o) { data.offer(o); }

    E dequeue() { return data.poll(); }

    E peek() { return data.peek(); }

    int size() { return data.size(); }

    public String toString() {
        return data.toString();
    }
}
```

Observe que a classe tem uma variável de instância **data** de tipo **java.util.ArrayDeque**. Observe também como é fácil implementar todos os métodos. Todo o trabalho está sendo feito pelo objeto **data**. Usando a delegação dessa forma, criamos

uma classe apenas com os métodos que queremos e conseguimos implementá-los com muito facilidade.

2. Para testar essa classe, crie um arquivo chamado **PureQueueDemo.java** e digite o código abaixo:

```
class PureQueueDemo {

    public static void main(String[] args) {
        PureQueue<String> q = new PureQueue<String>();
        System.out.println(q);
        q.enqueue("3");
        q.enqueue("abc");
        System.out.println(q);
        q.dequeue();
        System.out.println(q);
    }
}
```

3. Agora compile os dois arquivos e execute o método **main()** da classe **PureQueueDemo**. Você deve ver a saída a seguir:

```
[]  
[3, abc]  
[abc]
```

Verificação do progresso

1. O que diz o princípio da menor perplexidade?
2. Se **A** e **B** forem classes, um método tiver um parâmetro de tipo **A** e você puder usar um objeto de tipo **B** como parâmetro, então, **B** deve estender **A**. Verdadeiro ou falso?
3. Uma alternativa ao uso da herança que devemos considerar é a _____.

PADRÕES DE PROJETO

Já discutimos as vantagens e desvantagens de muitos projetos de software e mostramos algumas diretrizes e regras a serem seguidas que podem ajudá-lo a desenvolver bons projetos. Nesta seção, introduziremos mais ferramentas que o ajudarão a criar bons projetos de software. Essas ferramentas, que podem ser consideradas como algumas das melhores práticas da indústria de software para a resolução de problemas dentro de certos limites, são chamadas de *padrões de projeto*. A catalogação e orga-

Respostas:

1. Se um cliente achar que tem uma referência a um objeto de tipo **A**, mas na verdade tiver uma referência a um objeto de tipo **B**, não devem surgir surpresas quando ele chamar métodos no objeto.
2. Verdadeiro.
3. delegação

nização desses padrões avançaram desde o fim dos anos 1980. Nomes foram dados aos padrões para que os desenvolvedores de software tenham um vocabulário ao falar sobre projetos em um nível mais alto de abstração, o que de outra forma não seria possível. Por exemplo, agora os projetistas das equipes podem dizer coisas como “Usaremos o padrão Decorator aqui para tratar todas as opções”.

Nesta seção, introduziremos dois dos padrões de projeto mais simples para que você possa ter uma ideia do que são esses padrões.

Padrão Adapter

Um dos padrões mais simples, porém muito útil, é o padrão *Adapter*. Um exemplo do padrão Adapter não relacionado a software será dado primeiro. Nos Estados Unidos, os plugues elétricos da maioria dos equipamentos têm dois pinos chatos e um terceiro pino opcional redondo que cabem nas tomadas elétricas convencionais americanas. No entanto, em outros países, as tomadas são projetadas para equipamentos com plugues que têm dois ou três pinos redondos ou pinos chatos orientados em direções diferentes ou em posições diferentes em relação uns aos outros. A voltagem e a frequência da corrente elétrica também pode diferir em outros países. Neles, a interface elétrica não coincide com a interface dos equipamentos americanos. Logo, como podemos usar equipamentos americanos no exterior? A solução é comprar plugues adaptadores, com fendas em um lado, em que os equipamentos americanos se encaixem, e pinos no outro, que caibam nas tomadas dos demais países. Em outras palavras, a solução é não alterar os equipamentos ou as tomadas, mas criar um adaptador que permita que os equipamentos as usem. Agora vejamos como esse padrão aparece naturalmente no projeto de software.

Suponhamos que você estivesse trabalhando em uma equipe que construiu um grande *front end* (por exemplo, uma interface Web) para um dos principais aplicativos de sua empresa. O *back end* do aplicativo é outro sistema grande que contém todos os dados (talvez em um banco de dados) e as ferramentas e métodos para sua busca e tratamento. O *front end* foi projetado para usar a interface do *back end*; isto é, foi projetado para chamar os métodos do *back end* e tratar seus dados.

Agora suponhamos que outra empresa gostasse tanto de seu *front end* que quisesse usá-lo com o *back end* dela. Mas e se o *back end* da outra empresa tiver uma interface diferente da do *back end* para o qual seu *front end* foi projetado? Isto é, suponhamos que o *back end* da outra empresa tivesse um conjunto de métodos totalmente diferente para a busca e tratamento dos dados. O que ela deve fazer para seu *front end* funcionar com o *back end* dela? O problema que você tem é o de interfaces incompatíveis e, portanto, seu *front end* não pode se comunicar com o *back end* da outra empresa.

Uma maneira de a outra empresa resolver o problema seria modificar o *front end* de modo que ele use a interface de seu *back end*. Alternativamente, eles poderiam resolver o problema modificando seu *back end* para que use a interface esperada pelo *front end*. Uma terceira maneira seria fazer algumas das alterações no *front end* e algumas no *back end*. No entanto, você corre o risco de introduzir novas falhas se alterar o código funcional existente. Como a outra empresa pode fazer o *front end* e o *back end* se comunicarem com alterações mínimas no código atual?

Uma maneira elegante é fornecer uma nova classe que aja como um “adaptador” para ajustar uma interface à outra. Os objetos dessa classe ficarão entre o *front end* e o *back end*. O *front end* se comunicará com a classe adaptadora em vez de com

o *back end*, e o adaptador se comunicará com o *back end*. Mais precisamente, o adaptador fornecerá a interface que o *front end* espera, mas, na verdade, obterá os dados no *back end* e os transformará nos dados esperados pelo *front end*.

O uso de uma classe adaptadora, como nesse exemplo, para ajustar uma classe existente a uma interface diferente, é o que pretende o padrão Adapter.

Para mostrar um exemplo de classe adaptadora, deixemos de lado os *front end* e *back end* grandes. Em vez deles, consideremos a interface e as classes simples a seguir:

```
class Printer {
    void printRect(RectI r) {
        System.out.println(r.getWidth() + ", " + r.getHeight());
    }
}

interface RectI {
    int getWidth();
    int getHeight();
}

class SimpleRect implements RectI {
    int width, height;

    SimpleRect(int w, int h) {
        width = w; height = h;
    }

    public int getWidth() { return width; }
    public int getHeight() { return height; }
}

class NonConformingRect {
    int top, left, bottom, right;

    NonConformingRect(int a, int b, int c, int d) {
        top = a; left = b; bottom = c; right = d;
    }

    public int getTop() { return top; }
    public int getLeft() { return left; }
    public int getBottom() { return bottom; }
    public int getRight() { return right; }
}
```

A classe **Printer** é nosso “*front end*”. Seu método **printRect()** recebe, como argumento, um objeto que implementa a interface **RectI**. A classe **SimpleRect** implementa essa interface e, portanto, os objetos dessa classe podem ser passados como argumentos para o método **printRect()**. Ela é nosso “*back end*”.

Agora suponhamos que quiséssemos passar objetos **NonConformingRect** para **printRect()**. Essa classe não implementa a interface **RectI**, logo, instâncias não podem ser passadas como argumento para **printRect()**. Para pode passar **NonConfor-**

mingRects, você poderia alterar a classe **Printer** ou a interface **RectI**, mas alterar código funcional existente não é a maneira de tratar essa situação. Em vez disso, usaremos a classe adaptadora a seguir:

```
class RectAdapter implements RectI {  
    NonConformingRect ncRect;  
  
    RectAdapter(NonConformingRect r) {  
        ncRect = r;  
    }  
  
    public int getWidth() { return ncRect.getRight() - ncRect.getLeft(); }  
    public int getHeight() { return ncRect.getBottom() - ncRect.getTop(); }  
}
```

Observe que essa classe implementa a interface **RectI** e, portanto, pode ser passada para **printRect()**. Observe também que ela obtém seus dados em um **NonConformingRect**.

Quando você usar essa classe adaptadora, *nenhuma* alteração terá que ser feita nas classes existentes. Aqui está um programa completo que demonstra todas essas classes em ação:

```
class Printer {  
    void printRect(RectI r) {  
        System.out.println(r.getWidth() + ", " + r.getHeight());  
    }  
}  
  
interface RectI {  
    int getWidth();  
    int getHeight();  
}  
  
class SimpleRect implements RectI {  
    int width, height;  
  
    SimpleRect(int w, int h) {  
        width = w; height = h;  
    }  
  
    public int getWidth() { return width; }  
    public int getHeight() { return height; }  
}  
  
class NonConformingRect {  
    int top, left, bottom, right;  
  
    NonConformingRect(int a, int b, int c, int d) {  
        top = a; left = b; bottom = c; right = d;  
    }  
}
```

```

public int getTop() { return top; }
public int getLeft() { return left; }
public int getBottom() { return bottom; }
public int getRight() { return right; }
}

class RectAdapter implements RectI {
    NonConformingRect ncRect;

    RectAdapter(NonConformingRect r) {
        ncRect = r;
    }

    public int getWidth() { return ncRect.getRight() - ncRect.getLeft(); }
    public int getHeight() { return ncRect.getBottom() - ncRect.getTop(); }
}

class AdapterDemo {
    public static void main(String[] args) {
        Printer prntr = new Printer();
        prntr.printRect(new SimpleRect(3, 4));
        prntr.printRect(new RectAdapter(new NonConformingRect(1, 2, 3, 4)));
    }
}

```

Observe como, na classe principal, o objeto **NonConformingRect** é encapsulado dentro de um objeto **RectAdapter** antes de ser passado para o método **printRect()**. Quando esse programa for executado, você obterá a saída abaixo:

```

3, 4
2, 2

```

Padrão Observer

Consideremos um programa que um banco poderia usar para controlar contas de poupança. Suponhamos que ele tivesse a classe **SavingsAccount** a seguir para armazenar informações sobre uma conta:

```

class SavingsAccount {
    private String owner;
    private int acctNumber;
    private int balance; // em dólares americanos

    SavingsAccount(String o, int a) {
        owner = o;
        acctNumber = a;
        balance = 0;
    }

    // quantia positiva significa depósito
    // quantia negativa significa retirada
    void changeBalance(int amount) {

```

```
    balance += amount;
}

int getBalance() { return balance; }
int getAcctNumber() { return acctNumber; }
String getOwner() { return owner; }
}
```

Suponhamos também que muitos locais acessassem as contas de poupança, como bancos eletrônicos, as diversas filiais do banco e lojas varejistas que aceitassem um cartão de débito da conta.

Agora suponhamos que o banco decidisse instituir um novo recurso de segurança que marcasse todas as retiradas acima de 1.000 dólares para que fosse verificado se houve fraude. Como esse recurso pode ser implementado?

Uma maneira seria modificar os códigos de todos os locais em que transações ocorressem para que marcassem os saques grandes. Essa abordagem requer alterações demais no código existente. Uma maneira melhor seria a classe **SavingsAccount** cuidar da marcação. Tudo que teríamos de fazer é modificar o método **changeBalance()** para procurar retiradas grandes e, se encontradas, verificar se houve fraude. Em um projeto assim, a classe poderia usar a seguinte implementação do método **changeBalance()**:

```
void changeBalance(int amount) {
    balance += amount;
    if(amount < -1000)
        checkForFraud(amount);
}

void checkForFraud(int amount) {
    System.out.println("Checking for fraudulent " +
                       "withdrawal of amount: " + amount);
}
```

Observe que só foi preciso alterar uma classe porque a classe **SavingsAccount** cuida das fraudes usando seu método **checkForFraud()**.

No entanto, essa versão não é ideal porque agora a conta de poupança deixou de ser coesa. Ela tanto armazena informações quanto procura fraudes. Em vez disso, uma classe **FraudHandler** separada deve cuidar da procura por fraudes:

```
class FraudHandler {
    void checkForFraud(int amount, SavingsAccount acct) {
        System.out.println("Checking for fraudulent withdrawal" +
                           " of amount: " + amount + " from" +
                           " account " + acct.getAcctNumber());
    }
}
```

Com o uso dessa nova classe, o método **changeBalance()** ficaria assim:

```
void changeBalance(int amount) {
    balance += amount;
    if(amount < -1000)
```

```

    fraudHandler.checkForFraud(amount, this);
}

```

Nessa versão, a classe **SavingsAccount** tem uma variável de instância **fraudHandler** de tipo **FraudHandler** para a qual delega todo o tratamento de fraudes. Agora a conta de poupança ficou mais coesa. É um pouco deselegante ela ter de manter uma referência a um tratador de fraudes, já que gostaríamos de reduzir a vinculação entre as classes, mas podemos aceitar essa situação. Observe que a divisão de trabalho tem outra vantagem: a classe **FraudHandler** pode ser reutilizada se o banco quiser verificar fraudes em outros tipos de conta.

Suponhamos agora que o banco decidisse arrecadar mais cobrando dos clientes alguma taxa sempre que retirassem dinheiro de suas contas de poupança (afinal, não estarão economizando muito se fizerem retiradas, certo?). Como esse novo recurso pode ser implementado?

Como antes, poderíamos modificar o código de todos os locais em que transações ocorressem, mas isso envolveria a alteração de muitos códigos. Além disso, como antes, poderíamos fazer a conta de poupança cobrar a taxa, mas reduziríamos a coesão da classe **SavingsAccount** fazendo-a cuidar de taxas e manter a conta. Uma solução melhor seria uma classe **FeeHandler** reutilizável separada cuidar da taxa, como **FraudHandler** lida com fraudes. Se a classe **SavingsAccount** tiver uma variável de instância **feeHandler** de tipo **FeeHandler**, o método **changeBalance()** poderá ser reescrito assim:

```

void changeBalance(int amount) {
    balance += amount;
    if(amount < -1000)
        fraudHandler.checkForFraud(amount, this);
    if(amount < 0)
        feeHandler.handleWithdrawalFee(this);
}

```

Esse projeto separa satisfatoriamente as responsabilidades entre várias classes. No entanto, agora a classe **SavingsAccount** ficou um pouco mais deselegante por ter uma referência a um tratador de taxas.

Mas e se outras ações tiverem de ser adicionadas? Por exemplo, um programa de recompensas para pessoas que mantiverem um saldo de pelo menos 10.000 dólares todo mês ou uma taxa de saque a descoberto se o saldo ficar abaixo de 0? Se implementarmos esses novos recursos de maneira semelhante à dos recursos de detecção de fraude e taxa de retirada, nossa classe **SavingsAccount** terá de ser alterada novamente e ficará ainda mais vinculada a outras classes. Além disso, suponhamos que o banco decidisse que 1.000 dólares não é o valor apropriado para a verificação de fraude. Nesse caso, a classe **SavingsAccount** precisará ser alterada mais uma vez. O que podemos fazer para reduzir tanto as alterações de código quanto a vinculação?

Vejamos um resumo da situação. A classe **SavingsAccount** precisa notificar outras classes quando um evento especial ocorre, a saber, uma quantia sendo retirada ou depositada. Como pode fazê-lo sem estar fortemente vinculada a essas classes?

Nossa solução é usar o padrão *Observer*. Nesse padrão de projeto, há *observadores* e há *assuntos* a serem observados. Os assuntos também são chamados de

“publicadores” ou “transmissores”, e os observadores de “assinantes”, ou “ouvintes”. Os publicadores mantêm uma lista de assinantes e, sempre que há algo a publicar, eles notificam todos os assinantes. Nesse padrão, um publicador pode ter qualquer número de assinantes. (Para um objeto se tornar um assinante, precisa solicitar ao publicador que o adicione à sua lista de assinantes.)

Em nosso exemplo, a conta de poupança é o publicador e os tratadores de fraudes e taxas são os assinantes. Para usarmos o padrão Observer aqui, o publicador (a conta de poupança) precisa manter uma lista de assinantes e notificá-los sempre que o saldo mudar. Ele poderia manter uma lista de assinantes usando um array, por exemplo. Mas qual deve ser o tipo do array? A resposta é usar uma interface. Isto é, defina uma nova interface **BalanceChangeHandler**

```
interface BalanceChangeHandler {  
    void balanceChanged(int change, SavingsAccount acct);  
}
```

e então adicione uma nova variável de instância à classe **SavingsAccount** que refencie um array de **BalanceChangeHandlers**.

Todos os tratadores, como os tratadores de taxas e detecção de fraudes, precisam implementar essa interface e então se registrar na conta de poupança. Ao se registrar, eles serão adicionados ao array de assinantes. Assim, sempre que o saldo mudar, a conta de poupança notificará todos os assinantes registrados chamando seus métodos **balanceChanged()**. Nesses métodos, eles poderão tratar a mudança da maneira que quiserem.

Aqui está o código completo, inclusive com um programa de demonstração curto para testar as classes.

```
class SavingsAccount {  
    String owner;  
    int acctNumber;  
    int balance; // em dólares americanos  
    BalanceChangeHandler[] subscribers;  
    int numSubscribers;  
  
    SavingsAccount(String o, int a) {  
        owner = o;  
        acctNumber = a;  
        balance = 0;  
        subscribers = new BalanceChangeHandler[20];  
        // No máximo 20 assinantes são permitidos  
        numSubscribers = 0;  
    }  
  
    void addHandler(BalanceChangeHandler h) {  
        if(numSubscribers < 20) {  
            subscribers[numSubscribers] = h;  
            numSubscribers++;  
        }  
    }  
  
    // quantia positiva significa depósito
```

```
// quantia negativa significa retirada
void changeBalance(int amount) {
    if(amount != 0) {

        // Altera o saldo
        balance += amount;

        // Notifica todos os assinantes sobre a mudança
        for(int i = 0; i < numSubscribers; i++)
            subscribers[i].balanceChanged(amount, this);
    }
}

int getBalance() { return balance; }
int getAcctNumber() { return acctNumber; }
String getOwner() { return owner; }
}

class FeeHandler implements BalanceChangeHandler {

    public void balanceChanged(int change, SavingsAccount acct) {
        System.out.println("Deducting a fee from account " +
                           acct.getAcctNumber());
    }
}

class FraudHandler implements BalanceChangeHandler {

    public void balanceChanged(int change, SavingsAccount acct) {
        System.out.println("Checking for fraudulent withdrawal" +
                           " of amount: " + change + " from" +
                           " account " + acct.getAcctNumber());
    }
}

interface BalanceChangeHandler {
    void balanceChanged(int change, SavingsAccount acct);
}

class SavingsAccountDemo {
    public static void main(String[] args) {
        SavingsAccount acct = new SavingsAccount("Sam", 1234);
        FeeHandler feeHandler = new FeeHandler();
        FraudHandler fraudHandler = new FraudHandler();

        acct.addHandler(feeHandler);
        acct.addHandler(fraudHandler);

        acct.changeBalance(0); // nada ocorre
        acct.changeBalance(10);
    }
}
```

```
    acct.changeBalance(-10);  
}  
}
```

Quando o programa de demonstração for executado, a saída abaixo aparecerá:

```
Deducting a fee from account 1234  
Checking for fraudulent withdrawal of amount: 10 from account 1234  
Deducting a fee from account 1234  
Checking for fraudulent withdrawal of amount: -10 from account 1234
```

Vejamos um resumo de todas as vantagens do uso do padrão Observer. Em primeiro lugar, você pode registrar muitos tratadores diferentes na conta de poupança sem ter de alterar o código da classe **SavingsAccount**. Em segundo lugar, a vinculação é reduzida entre a conta de poupança e os tratadores. Isto é, a conta de poupança não precisa saber nada sobre os tratadores a não ser que implementam a interface **BalanceChangeHandler**, que permite que a conta chame os métodos **balanceChanged()** dos tratadores.

É preciso mencionar que o pacote **java.util** fornece uma estrutura que dá suporte ao padrão Observer. Essa estrutura é discutida no Capítulo 24.

Esses dois padrões (Adapter e Observer) são apenas exemplos dos vários padrões de projeto que fornecem soluções ideais para problemas comuns de software. À medida que você aprender mais sobre desenvolvimento de software, vai querer conhecer melhor esses outros padrões para poder dominar a linguagem de padrões de projeto.

Verificação do progresso

1. As melhores práticas da indústria de software para a resolução de problemas com certos limites são chamadas de _____.
2. O uso de uma classe para adaptarmos uma classe existente a uma interface diferente é o padrão _____.
3. No padrão Observer, há dois tipos de classes envolvidas: os publicadores e _____.

EXERCÍCIOS

1. Para métodos, classes e variáveis, você deve usar nomes _____.
2. Todos os princípios discutidos neste capítulo são regras rigorosas que devem ser seguidas para termos um software elegante. Verdadeiro ou falso?
3. Um bom programador cria código autodocumentado e, portanto, nunca precisa adicionar comentários internos. Verdadeiro ou falso?

Respostas:

1. padrões de projeto
2. Adapter
3. os assinantes

4. Geralmente é melhor inserir todos os métodos importantes em uma classe e tornar a maioria das outras classes simples classes secundárias. Verdadeiro ou falso?
5. Uma documentação desatualizada é melhor do que nenhuma documentação. Verdadeiro ou falso?
6. Um método com encadeamento de métodos na forma **a.getB().getC().getD().doSomething()** não segue a _____.
7. Suponhamos que você tivesse uma classe **Person** com variáveis de instância contendo o nome, a data de nascimento, o endereço, o nome do cônjuge, o nome dos filhos e a ocupação da pessoa. De acordo com o padrão Expert, essa classe deve ter métodos que lidem com qualquer tratamento desses dados de que outras classes precisem. No entanto, um número ilimitado de tratamentos de um objeto **Person** pode ser executado. Quantos métodos a classe **Person** deve ter para estar correta?
8. Normalmente uma classe tem tanto variáveis de instância quanto métodos. Considere dois casos extremos: uma classe **A** que tem muitas variáveis de instância e nenhum método e uma classe **B** que tem vários métodos e nenhuma variável de instância. Qual dos princípios discutidos neste capítulo é mais provável que essas classes violem?
9. Quando tentamos falar com um executivo de uma empresa, temos pelo menos três opções:
 - A. Ligar e aguardar até ele estar livre.
 - B. Continuar ligando em intervalos de minutos até ele estar livre.
 - C. Deixar uma mensagem pedindo ao executivo que retorne quando estiver livre.
 Qual dessas opções chega mais perto do padrão Observer?
10. Cite uma falha na classe a seguir que a torna deselegante:

```
class NamedObject {
    private String name;
    NamedObject(String n) { name = n; }
    void setName(String n) { name = n; }
}
```

11. Considere o método a seguir de uma classe **B**. Suponhamos que a classe **A** tivesse os métodos **load()**, **sort()** e **store()**.

```
void loadSortAndStore(A a) {
    a.load();
    a.sort();
    a.store();
}
```

Esse método é elegante? Por quê? Se não for, explique o que deve ser feito para torná-lo elegante.

12. Se uma classe tiver uma variável de instância inteira **x**, é melhor tornar **x** publicamente acessível para que outros objetos possam lê-la e gravá-la ou tornar **x** privada e adicionar os método **getX()** e **setX()** para a leitura e gravação de **x**?

13. Suponhamos que você tivesse objetos **Person** e objetos **Location** e precisasse registrar o local de nascimento de cada pessoa. Há pelo menos três maneiras de organizar essa informação:
- Cada objeto **Person** teria uma variável de instância de tipo **Location** que armazenaria o local de nascimento.
 - Cada **Location** manteria um conjunto de referências a objetos **Person** correspondente às pessoas nascidas nesse local.
 - Um terceiro objeto manteria uma tabela de objetos **Person** e seus objetos **Location** com o local de nascimento.
- Discuta as vantagens e desvantagens de cada um desses três projetos.
14. Para implementar uma classe **Rectangle** que armazene sua posição e tamanho, você poderia usar as quatro variáveis de instância inteiras **x**, **y**, **width** e **height** em que (x,y) é o canto superior esquerdo ou poderia usar as oito variáveis **x1**, **y1**, **x2**, **y2**, **x3**, **y3**, **x4** e **y4** para armazenar as coordenadas (x,y) dos quatro cantos do retângulo. Qual dessas duas implementações é melhor e por quê?
15. Suponhamos que você estivesse criando um método **getChildren()** em uma classe **Person** que retornasse um array de **Persons**. Em seguida, entre outras coisas, você precisa lidar com o caso no qual a pessoa em que o método foi chamado não tem filhos. O que seu método deve retornar nesse caso? Ele deve retornar **null** ou um array de tamanho 0?
16. Um projetista de classes está projetando uma classe **Person** que ele deseja que seja tão popular e reutilizável quanto a classe **String** e, portanto, está tentando aumentar a capacidade de reutilização com a inclusão do maior número de métodos possível. O construtor e os métodos que ele adicionou estão listados abaixo. Quais dos métodos não deveriam ser incluídos em uma classe **Person** bem projetada?
- Person(int birthDate, String name, String address)** // construtor
 - String getName()**
 - void setName(String name)**
 - String toString() // retorna o nome da pessoa**
 - int getSSN() // retorna o número de inscrição na previdência**
 - String getSSN() // retorna o número de inscrição na previdência como um String**
 - Person[] siblings() // retorna um array com os irmãos da pessoa**
 - Race getRace()**
 - Person getBoss() // retorna a pessoa que empregou a pessoa em questão**
 - boolean isDogOwner()**
 - boolean isUSCitizen()**
 - String getAddress()**
 - int numberOfCarsOwned()**
 - int getBirthdate()**
 - void setBirthDate(int date)**
17. Suponhamos que lhe pedissem para implementar uma classe cujos objetos precissem armazenar 20 dados e os dados fossem de vários tipos. Algumas pessoas poderiam argumentar que é melhor armazenar cada dado em uma variável de instância separada. Outras que é um absurdo haver uma variável de instância para cada dado e, portanto, seria melhor armazenar todos os dados na mesma variável

de instância, como em um array **data** de tipo **Object**. Por exemplo, um objeto **Person** poderia armazenar o primeiro nome da pessoa no local **data[0]**, o sobrenome em **data[1]**, o endereço em **data[2]**, a idade em **data[3]** e assim por diante. Dê sua opinião sobre essas duas abordagens. Discuta a abordagem que usaria.

18. Crie um exemplo do padrão Adapter não relacionado a software que não seja o exemplo de tomadas elétricas mencionado neste capítulo.
19. Considere as classes **ParttimeEmployee** e **FulltimeEmployee**. Um funcionário de meio expediente tem menos benefícios do que um de tempo integral e, portanto, a classe **FulltimeEmployee** precisa de mais campos do que **ParttimeEmployee**. Logo, para evitar duplicação, faria sentido tornarmos a classe **FulltimeEmployee** subclasse de **ParttimeEmployee** de modo que ela pudesse herdar as variáveis de instância da superclasse. Essa abordagem é elegante?
20. Cite o que não é elegante na classe a seguir. *Dica:* tem relação com as invariantes de classe.

```
/* Os objetos dessa classe armazenam um double aleatório entre 0 e
1.*/
public class RandomDoubleStore {
    double value = 0.0;

    public RandomDoubleStore() { }

    public void initialize() {
        value = Math.random();
    }
}
```

21. O texto mencionou que todo método deve retornar um valor ou modificar o estado de um ou mais objetos, mas não ambas as coisas. Também mencionou que o método **pop()** tradicional de uma pilha não segue essa convenção, já que ele tanto remove o valor do topo da pilha quanto o retorna. Se você fosse forçado a alterar a implementação de **pop()** em uma pilha para que ela seguisse a convenção, qual seria a maneira mais simples de fazê-lo?
22. Suponhamos que um objeto **Person** tivesse um método **getHeight()** que retornasse a altura da pessoa. Considere o método a seguir:

```
int getAverageHeight(Person p1, Person p2, Person p3) {
    int p1Height = p1.getHeight();
    int p2Height = p2.getHeight();
    int p3Height = p3.getHeight();

    return (p1Height + p2Height + p3Height) / 3;
}
```

Como você pode ver, ele calcula e retorna a altura média das três pessoas. Por que esse método não é muito elegante? Sugira uma alteração para torná-lo mais elegante.

23. Suponhamos que houvesse uma interface **Nameable** que definisse um método:

```
interface Nameable {  
    String getName();  
}
```

e que houvesse uma classe **Person** que implementasse a interface **Nameable**. Para concluir, suponhamos que, em outra classe, houvesse o método abaixo:

```
| String nameOf(Person p) { return p.getName(); }
```

Qual é a maneira mais simples de tornar o método **nameOf()** mais útil?

24. Suponhamos que uma pessoa precisasse modelar o conceito de balões e, portanto, querendo usar a herança para gerar um projeto elegante, ela criasse uma classe **RubberObject** e uma subclasse **Balloon**. Você consegue ver algum problema nesse projeto?
25. Uma pessoa poderia dizer que “Fofo é um gato” e “Um gato é um mamífero” e, portanto, **Fofo** deve ser subclasse de **Gato** e **Gato** deve ser subclasse de **Mamífero**, porque temos dois relacionamentos *é-um*. O que está errado nesse raciocínio?
26. Suponhamos que uma classe **NamedObject** tivesse um construtor que usasse, como seu único parâmetro, um string que fosse o nome inicial do objeto. Ela também tem um método **setName()** que usa um string como parâmetro para a alteração do nome do objeto e um método **getName()** que retorna o nome. Agora suponhamos que você quisesse criar uma classe **ImmutableNamedObject** cujo nome inicial não pudesse ser alterado. Uma maneira de implementar essa classe é torná-la subclasse de **NamedObject** e, na subclasse, sobrepor o método **setName()** para que ele não faça coisa alguma. O que acha da elegância dessa abordagem?
27. Suponhamos que uma classe **RentalCarCompany** tivesse três variáveis de instância inteiras, **numInUse**, **numAvail** e **total**, que referenciassem o número de carros que estão alugados atualmente, o número de carros atualmente disponíveis para serem alugados e o número total de carros (a soma dos outros dois valores).
 - A. Por que essa classe não é elegante?
 - B. Suponhamos que ela tivesse o método a seguir para a atualização do número de carros, por exemplo, quando alguns carros ficassem danificados, fossem comprados ou vendidos:

```
void updateNumCars(int newInUse, int newAvail, int newTotal) {  
    numInUse = newInUse; numAvail = newAvail; total = newTotal;  
}
```

O que há de deselegante nesse método?

28. Há duas maneiras de informações serem transferidas de uma origem a um destino. No modelo *pull*, o destino vai até a origem e obtém as informações (o destino “puxa” as informações da origem até ele próprio). No modelo *push*, a origem envia as informações ao destino (a origem “empurra” as informações de si própria até o destino). Que modelo tem maior proximidade com o padrão Observer?

PARTE II

Introdução à programação de GUIs com Swing

A Parte II introduz o kit de ferramentas de Swing. Swing é a abordagem moderna de Java para a criação de interfaces gráficas de usuário (GUIs). Esta parte começa apresentando a filosofia de projeto de Swing e vários conceitos básicos. Os capítulos subsequentes examinam diversos controles Swing, o sistema de menus, a atualização da tela e a criação de applets baseados em Swing.

***Nota:** Uma descrição completa das classes, interfaces e métodos Swing pode ser encontrada na documentação do JDK, que é disponibilizada on-line pela Oracle.*

Aspectos básicos de Swing

PRINCIPAIS HABILIDADES E CONCEITOS

- Saber as origens de Swing
- Entender a importância dos componentes leves e de uma aparência adaptável
- Entender a arquitetura MVC
- Ter uma visão geral dos componentes e contêineres de Swing
- Criar, compilar e executar um aplicativo simples de Swing
- Saber os aspectos básicos do tratamento de eventos
- Entender o papel dos gerenciadores de leiaute
- Usar o componente **JButton**
- Usar o componente **JTextField**

Com exceção dos exemplos de applets mostrados no Capítulo 15, todos os programas da Parte I se basearam no console, ou seja, não fazem uso de uma interface gráfica de usuário (GUI). Apesar de os programas de console serem ótimos no ensino dos aspectos básicos de Java, muitos aplicativos do mundo real são baseados em GUI. Portanto, este capítulo e os quatro próximos examinarão o kit de ferramentas da GUI Java moderna: Swing.

Swing é uma coleção de classes e interfaces que oferece um rico conjunto de componentes visuais, como botões de ação, campos de texto, barras de rolagem, caixas de seleção, árvores, tabelas e menus, que podem ser personalizados para atender a qualquer necessidade. Com o uso de Swing, seu aplicativo terá uma interface de usuário moderna baseada em controles de GUI. Ou seja, você pode criar programas com aparência e comportamento como os de outros aplicativos de GUI que conhece.

O objetivo deste capítulo é introduzir Swing, inclusive sua história, conceitos básicos, filosofia de projeto e principais recursos. Como você verá, Swing é composto por muitos elementos relacionados. Essa correlação o torna uma estrutura poderosa, porém otimizada, e exige o aprendizado de como esses elementos funcionam em conjunto. Este capítulo também introduz um recurso Java importante: o evento. Entre outros usos, o evento é o mecanismo pelo qual o programa é notificado sobre inte-

rações do usuário. O capítulo termina com a introdução a dois controles comuns de Swing, o botão e o campo de texto, usando-os para demonstrar os elementos básicos da programação com Swing.

ORIGENS E FILOSOFIA DE PROJETO DE SWING

Swing não existia nos primórdios da linguagem Java. Ele foi uma resposta às deficiências presentes no subsistema de GUI original da linguagem: o Abstract Window Toolkit (AWT). O AWT define um conjunto básico de componentes que dá suporte a uma interface gráfica usável, mas limitada. Uma das razões da natureza limitada do AWT é o fato de seus componentes dependerem de janelas nativas específicas da plataforma, ou *pares*. Por isso, são considerados *pesados*.

O uso de pares nativos gera vários problemas. Em primeiro lugar, devido às diferenças entre os sistemas operacionais, um componente pode aparecer, ou até mesmo agir, diferentemente em plataformas distintas. Em segundo lugar, a aparência de cada componente era fixa (porque é definida pela plataforma) e não podia ser (facilmente) alterada. Em terceiro lugar, o uso de componentes pesados gerava algumas restrições incômodas. Por exemplo, um componente pesado é sempre opaco.

Não muito tempo após o lançamento original de Java, ficou claro que as limitações e restrições presentes no AWT eram graves o suficiente para que uma abordagem melhor fosse necessária. A solução foi Swing. Introduzido em 1997, Swing foi incluído como parte de Java Foundation Classes (JFC). Inicialmente, ele estava disponível para uso com Java 1.1 como uma biblioteca separada. No entanto, a partir de Java 1.2, Swing (e o resto do JFC) foi totalmente integrado à biblioteca Java.

Swing resolve as limitações associadas aos componentes do AWT com o uso de dois recursos-chave: *componentes leves* e *aparência adaptável*. Embora sejam amplamente transparentes para o programador, esses dois recursos formam a base da filosofia de projeto Swing e são a razão de grande parte de seu poder e flexibilidade. Examinemos cada um.

Com muito poucas exceções, os componentes Swing são *leves*, ou seja, o componente é escrito totalmente em Java. Em vez de ter seu próprio par nativo, ele usa a janela fornecida por um predecessor pesado. Os componentes leves apresentam algumas vantagens importantes, inclusive eficiência e flexibilidade. Por exemplo, um componente leve pode ser transparente, o que permite a criação de formas não retangulares. Além disso, uma vez que os componentes leves não são transformados em pares específicos da plataforma, a aparência de cada componente é determinada por Swing. Assim, cada componente pode funcionar de maneira coerente em qualquer plataforma.

Como cada componente de Swing é gerado por código Java e não por pares específicos da plataforma, é possível separar a aparência e a lógica de um componente, e é isso que Swing faz. A separação da aparência fornece uma vantagem significativa: permite alterar a maneira como o componente é gerado sem afetar nenhum de seus outros aspectos. Em outras palavras, é possível “conectar” uma

nova aparência a qualquer componente sem criar nenhum efeito colateral no código que faz uso dele.

Java fornece aparências, como metal e Motif, que estão disponíveis para todos os usuários de Swing. A aparência metal também é chamada de *aparência Java*. Ela é uma aparência independente da plataforma que está disponível em todos os ambientes de execução Java. Também é a aparência padrão. Portanto, os exemplos deste livro usam a aparência padrão Java (metal).

Swing pode fornecer uma aparência adaptável porque usa uma versão modificada da arquitetura de componentes clássica *modelo-visão-controlador* (*MVC*, *model-view-controller*). Na terminologia MVC, *modelo* são as informações de estado associadas ao componente. Por exemplo, no caso de uma caixa de seleção, o modelo contém um campo que indica se a caixa está marcada ou desmarcada. *Visão* determina como o componente será exibido na tela, incluindo qualquer aspecto que seja afetado pelo estado atual do modelo. *Controlador* determina como o componente reagirá ao usuário. Por exemplo, quando o usuário clicar em uma caixa de seleção, o controlador reagirá alterando o modelo para refletir a escolha (marcada ou desmarcada). Isso resultará na atualização da exibição. Com a separação do componente em um modelo, um modo de exibição e um controlador, a implementação específica de um pode ser alterada sem afetar os outros dois. Por exemplo, diferentes implementações da visão podem gerar o mesmo componente de maneiras distintas sem afetar o modelo ou o controlador.

Embora a arquitetura MVC e os princípios existentes por trás dela sejam conceitualmente sólidos, o alto nível de separação entre a visão e o controlador não seria benéfico para os componentes de Swing. Em vez disso, Swing usa uma versão modificada do MVC que combina a visão e o controlador na mesma entidade lógica chamada *delegação de UI*. Por essa razão, a abordagem de Swing é chamada de arquitetura *modelo-delegação* ou arquitetura *de modelo separável*. Logo, embora a arquitetura de componentes de Swing seja baseada no MVC, ela não usa a implementação clássica. Mesmo que você não trabalhe diretamente com modelos ou delegação de UI neste capítulo, eles estarão presentes em segundo plano.

No decorrer deste capítulo, você verá que, mesmo com Swing incorporando muitos conceitos de projeto sofisticados, ele é fácil de usar. Na verdade, alguém poderia dizer que a facilidade de uso de Swing é a vantagem mais importante. Resumindo, Swing torna gerenciável a quase sempre difícil tarefa de desenvolver a interface de usuário do programa. Isso permitirá que você se concentre na GUI propriamente dita, em vez de nos detalhes da implementação.

Pergunte ao especialista

P Você diz que Swing define a aparência da GUI Java moderna. Isso quer dizer que Swing substitui o AWT?

R Não, Swing não substitui o AWT. Swing se baseia nos fundamentos fornecidos pelo AWT. Logo, o AWT ainda é uma parte crucial de Java. Swing também usa o mesmo mecanismo de tratamento de eventos do AWT. Embora não seja preciso conhecer bem o AWT para usar Swing, as partes do AWT requeridas por Swing serão descritas à medida que forem necessárias em toda a Parte II.

Verificação do progresso

1. Um componente leve é escrito em código de máquina altamente otimizado. Verdadeiro ou falso?
2. Que recurso permite que a aparência de um componente Swing seja alterada?
3. Swing usa uma implementação padrão da arquitetura MVC?

COMPONENTES E CONTÊINERES

Uma GUI de Swing é composta por dois itens principais: *componentes* e *contêineres*. No entanto, essa distinção é em grande parte conceitual, porque todos os contêineres também são componentes. A diferença entre os dois pode ser detectada em sua finalidade: com o significado dado usualmente ao termo, um *componente* é um controle visual independente, como um botão de ação ou um controle deslizante, e um contêiner contém um grupo de componentes. Logo, um contêiner é um tipo especial de componente que é projetado para conter outros componentes. Todas as GUIs de Swing terão pelo menos um contêiner. Já que os contêineres são componentes, também podem conter outros contêineres. Isso permite que Swing defina a chamada *hierarquia de contenção*, no topo da qual deve haver um *contêiner de nível superior*.

Examinemos mais detalhadamente os componentes e contêineres.

Componentes

Em geral, os componentes de Swing são derivados da classe **JComponent**. (As únicas exceções são os quatro contêineres de nível superior, descritos na próxima seção.) **JComponent** fornece a funcionalidade que é comum a todos os componentes, como por exemplo, dar suporte à aparência adaptável. **JComponent** também herda as classes **Container** e **Component** do AWT. Logo, um componente de Swing se baseia em e é compatível com um componente do AWT.

Todos os componentes de Swing são representados por classes definidas dentro do pacote **javax.swing**. Estas eram as definidas pelos componentes (inclusive as usadas como contêineres) quando o texto foi escrito:

Respostas:

1. Falso.
2. Aparência adaptável.
3. Não, Swing usa uma abordagem modificada chamada Modelo-Delegação ou Modelo Separável.

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	JTextField	JTextPane
JToggleButton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Observe que todas as classes de componentes começam com a letra **J**. Por exemplo, a classe de rótulo é **JLabel**, a classe de botão de ação é **JButton** e a classe de barra de rolagem é **JScrollBar**. Várias delas serão examinadas nesta parte do livro.

Contêineres

Swing define dois tipos de contêineres. Primeiro, temos os contêineres de nível superior: **JFrame**, **JApplet**, **JWindow** e **JDialog**. Esses contêineres não herdam **JComponent**, no entanto, herdam as classes **Component** e **Container** do AWT. Diferentemente de outros componentes de Swing, que são leves, os contêineres de nível superior são pesados. Isso os torna um caso especial na biblioteca de componentes de Swing.

Como o nome sugere, um contêiner de nível superior deve estar no topo de uma hierarquia de contenção. Ele não pode estar dentro de outro contêiner. Além disso, toda hierarquia de contenção deve começar com um contêiner de nível superior. O mais usado para aplicativos é **JFrame**; o usado para applets é **JApplet**.

O segundo tipo de contêiner com suporte no Swing é o contêiner leve. Os contêineres leves *herdam JComponent*. Exemplos de contêineres leves são **JPanel** e **JRootPane**. Geralmente, os contêineres leves são usados para organizar e gerenciar coletivamente grupos de componentes relacionados, porque um contêiner leve pode estar contido dentro de outro contêiner. Logo, você pode usar contêineres leves para criar subgrupos de controles relacionados dentro de um contêiner externo.

Painéis do contêiner de nível superior

Cada contêiner de nível superior define um conjunto de *painéis*. No topo da hierarquia, temos uma instância de **JRootPane**, um contêiner leve cuja finalidade é a de gerenciar os outros painéis. Ele também ajuda a gerenciar a barra de menus opcional. Os painéis que compõem o painel raiz se chamam *painel de vidro*, *painel de conteúdo* e *painel em camadas*.

O painel de vidro é o painel de nível superior. Ele fica acima de todos os outros painéis e os cobre totalmente. Permite o gerenciamento de eventos do mouse que afetem o contêiner inteiro (e não um controle individual) ou a geração de algo acima

de outro componente, por exemplo. Na maioria dos casos, não precisamos usá-lo diretamente. O painel em camadas permite que os componentes recebam um valor de profundidade. Esse valor determina como os componentes serão sobrepostos. (Logo, o painel em camadas nos permite especificar uma ordem Z (de profundidade) para um componente, embora geralmente isso não seja necessário.) O painel em camadas contém o painel de conteúdo e a barra de menus (opcional). Apesar de o painel de vidro e os painéis em camadas fazerem parte da operação de um contêiner de nível superior e desempenharem papéis importantes, muito do que fornecem ocorre em segundo plano. Não os usaremos diretamente.

O painel com o qual seu aplicativo interagirá mais é o painel de conteúdo, porque é a ele que você adicionará componentes visuais. Em outras palavras, quando você adicionar um componente, como um botão, a um contêiner de nível superior, o adicionará ao painel de conteúdo. Portanto, o painel de conteúdo contém os componentes com os quais o usuário interage.

Verificação do progresso

1. Todos os componentes de Swing devem ser armazenados em um _____.
2. Com apenas algumas exceções, os componentes de Swing são derivados de **JComponent**. Verdadeiro ou falso?
3. Além do painel raiz, que outros painéis todos os contêineres de nível superior têm?
4. Os contêineres de nível superior são um caso especial porque são pesados, e não leves. Verdadeiro ou falso?

GERENCIADORES DE LEIAUTE

Antes de você começar a escrever um programa Swing, há mais uma coisa que precisa conhecer: o gerenciador de leiaute. O gerenciador de leiaute controla a posição dos componentes dentro de um contêiner. Em outras palavras, um gerenciador de leiaute determina o local dos controles dentro de um contêiner. Java oferece vários gerenciadores de leiaute. Muitos são fornecidos pelo AWT (dentro de **java.awt**), mas Swing adiciona alguns por conta própria. Todos os gerenciadores de leiaute são instâncias de uma classe que implementa a interface **LayoutManager**. (Algumas também implementam a interface **LayoutManager2**.) Aqui está uma lista de alguns dos gerenciadores de leiaute disponíveis para o programador de Swing:

Respostas:

1. contêiner
2. Verdadeiro.
3. O painel de vidro, o painel de conteúdo e o painel em camadas.
4. Verdadeiro.

FlowLayout	Leiaute simples que posiciona os componentes da esquerda para a direita e de cima para baixo. (Posiciona os componentes da direita para a esquerda em algumas configurações regionais.)
BorderLayout	Posiciona os componentes no centro ou nas bordas do contêiner. É o leiaute padrão para um painel de conteúdo.
GridLayout	Dispõe os componentes dentro de uma grade.
GridBagLayout	Dispõe componentes de tamanhos diferentes dentro de uma grade flexível.
BoxLayout	Dispõe os componentes vertical ou horizontalmente dentro de uma caixa.
SpringLayout	Dispõe os componentes de acordo com um conjunto de restrições.

Este capítulo só usa dois gerenciadores de leiaute – **BorderLayout** e **FlowLayout**. Ambos são muito fáceis de usar e serão descritos aqui. Outros gerenciadores de leiaute serão descritos quando necessário.

BorderLayout é o gerenciador de leiaute padrão para o painel de conteúdo. Ele implementa um estilo de leiaute que define cinco locais onde um componente pode ser adicionado. O primeiro é o centro, os outros quatro são os lados (isto é, bordas), que se chamam norte, sul, leste e oeste. Por padrão, quando você adicionar um componente ao painel de conteúdo, o adicionará ao centro. Para adicionar um componente em uma das outras regiões, especifique seu nome.

Embora o leiaute de borda seja útil em algumas situações, com frequência um gerenciador de leiaute mais flexível é necessário. Um dos mais simples é **FlowLayout**. Um leiaute de fluxo dispõe os componentes em uma linha de cada vez, de cima para baixo. Quando uma linha fica cheia, o leiaute avança para a próxima linha. Esse esquema fornece pouco controle sobre a inserção dos componentes, mas é muito fácil de usar. No entanto, cuidado ao redimensionar o quadro, porque a posição dos componentes mudará.

PRIMEIRO PROGRAMA SWING SIMPLES

Os programas Swing diferem dos programas de console mostrados anteriormente neste livro e também diferem dos applets baseados no AWT mostrados no Capítulo 15. Além de os programas Swing usarem o conjunto de componentes de Swing para realizar a interação com o usuário, eles têm requisitos especiais relacionados ao uso de threads. A melhor maneira de entendermos a estrutura de um programa Swing é trabalhar com um exemplo. Há dois tipos de programas Java em que Swing costuma ser usado. O primeiro é o aplicativo desktop e o segundo é o applet. Esta seção mostrará como criar um aplicativo Swing. A criação de um applet Swing será descrita no Capítulo 21.

Embora curto, o programa a seguir mostra uma maneira de criar um aplicativo Swing. No processo, demonstra vários recursos-chave. Ele usa dois componentes de Swing: **JFrame** e **JLabel**. **JFrame** é o contêiner de nível superior normalmente usado em aplicativos Swing. **JLabel** é o componente de Swing que cria um rótulo, que é um componente que exibe informações. O rótulo é o componente mais simples de

Swing porque é passivo, isto é, não responde a entradas do usuário, ele apenas exibe saídas. O programa usa um contêiner **JFrame** para armazenar uma instância de **JLabel**. O rótulo exibe uma mensagem de texto curta.

```
// Um programa Swing simples.

import javax.swing.*; ← Os programas Swing devem importar javax.swing.

class SwingDemo {

    SwingDemo() { ← Cria um contêiner de nível superior.

        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("A Simple Swing Application"); ←

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(275, 100); ← Define as dimensões do contêiner.

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); ← Encerra o programa quando o usuário clica na caixa Fechar.

        // Cria um rótulo baseado em texto.
        JLabel jlab = new JLabel(" Swing defines the modern Java GUI."); ←

        // Adiciona o rótulo ao painel de conteúdo.
        jfrm.add(jlab); ← Cria um JLabel. Adiciona o rótulo ao painel de conteúdo.

        // Exibe o quadro.
        jfrm.setVisible(true); ← Torna o quadro visível.
    }

    public static void main(String[] args) {
        // Cria o quadro na thread de despacho de evento.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SwingDemo(); ← Cria a GUI na thread de despacho de evento.
            }
        });
    }
}
```

Os programas Swing são compilados e executados da mesma forma que outros aplicativos Java. Logo, para compilar esse programa, você pode usar a linha de comando a seguir:

```
| javac SwingDemo.java
```

Para executar o programa, use esta linha de comando:

```
| java SwingDemo
```

Quando o programa for executado, produzirá a janela mostrada na Figura 17-1.

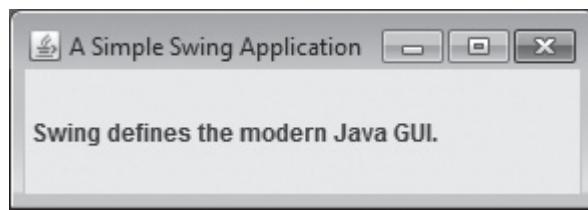


Figura 17-1 Janela produzida pelo programa **SwingDemo**.

Primeiro exemplo de Swing linha a linha

O programa **SwingDemo** ilustra vários conceitos-chave de Swing. Por isso o examinaremos com cuidado, linha a linha. O programa começa importando o pacote a seguir:

```
| import javax.swing.*;
```

O pacote **javax.swing** contém os componentes e modelos definidos por Swing. Por exemplo, ele define classes que implementam rótulos, botões, controles de edição e menus. Esse pacote será incluído em todos os programas que usarem Swing.

Em seguida, o programa declara a classe **SwingDemo** e um construtor para ela. É no construtor que grande parte da ação do programa ocorre. Ele começa criando um **JFrame**, usando a linha de código a seguir:

```
| JFrame jfrm = new JFrame("A Simple Swing Application.");
```

Esse código cria um contêiner chamado **jfrm** que define uma janela retangular com uma barra de título, botões fechar, minimizar, maximizar e restaurar e um menu de sistema. Portanto, cria uma janela de nível superior padrão. O título da janela é passado para o construtor.

A janela é então dimensionada com o uso da seguinte instrução:

```
| jfrm.setSize(275, 100);
```

O método **setSize()** define as dimensões da janela, que são especificadas em pixels. Sua forma geral é mostrada aqui:

```
void setSize(int largura, int altura)
```

Nesse exemplo, a largura da janela é configurada com 275 e a altura com 100.

Por padrão, quando uma janela de nível superior é fechada (como quando o usuário clica na caixa Fechar), ela é removida da tela, mas o aplicativo não é encerrado. Embora esse comportamento padrão seja útil em algumas situações, não é o que a maioria dos aplicativos precisa. Em vez disso, geralmente queremos que o aplicativo inteiro seja encerrado quando sua janela de nível superior é fechada. Há algumas maneiras de fazê-lo. A mais fácil é chamar **setDefaultCloseOperation()**, como faz o programa:

```
| jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Após essa chamada ser executada, o fechamento da janela fará o aplicativo inteiro ser encerrado. A forma geral de **setDefaultCloseOperation()** é mostrada aqui:

```
void setDefaultCloseOperation(int o que fazer)
```

O valor passado em *o que fazer* determina o que ocorrerá quando a janela for fechada. Há outras opções além de **JFrame.EXIT_ON_CLOSE**. Elas são mostradas abaixo:

DISPOSE_ON_CLOSE	HIDE_ON_CLOSE	DO NOTHING_ON_CLOSE
------------------	---------------	---------------------

Seus nomes refletem suas ações. Essas constantes foram declaradas em **Window-Constants**, uma interface declarada no pacote **javax.swing** que é implementada por **JFrame**.

A próxima linha de código cria um componente **JLabel**:

```
| JLabel jlab = new JLabel(" Swing defines the modern Java GUI.");
```

JLabel é o componente de Swing mais fácil de usar, porque não aceita entradas do usuário, apenas exibe informações, que podem ser compostas por texto, ícone ou uma combinação dos dois. O rótulo criado pelo programa contém só texto, que é passado para seu construtor.

Agora temos uma linha de código que adiciona o rótulo ao painel de conteúdo do quadro:

```
| jfrm.add(jlab);
```

Como explicado anteriormente, todos os contêineres de nível superior têm um painel de conteúdo em que os componentes são armazenados. Logo, para adicionar um componente a um quadro, você deve adicioná-lo ao seu painel de conteúdo. Isso é feito com uma chamada a **add()** na referência **JFrame** (**jfrm**, nesse caso). O método **add()** tem várias versões. A forma geral da usada pelo programa é mostrada aqui:

Component add(Component *comp*)

Por padrão, o painel de conteúdo associado a um **JFrame** usa layout de borda. Essa versão de **add()** adiciona o componente (nesse caso, um rótulo) ao centro. Outras versões permitem a especificação de uma das regiões de borda. Quando um componente é adicionado ao centro, seu tamanho é ajustado automaticamente para caber na região central.

A última instrução do construtor de **SwingDemo** faz a janela ficar visível.

```
| jfrm.setVisible(true);
```

O método **setVisible()** tem esta forma geral:

void setVisible(boolean *flag*)

Se *flag* for igual a **true**, a janela será exibida, caso contrário, ela será ocultada. Por padrão, **JFrame** é invisível, logo, devemos chamar **setVisible(true)** para exibi-lo.

Dentro de **main()**, um objeto **SwingDemo** é criado, o que exibe a janela e o rótulo. Observe que o construtor de **SwingDemo** é chamado com as linhas de código a seguir:

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new SwingDemo();
    }
});
```

Essa sequência faz um objeto **SwingDemo** ser criado na *thread de despacho do evento* e não na thread principal do aplicativo. Vejamos o porquê. Em geral, os programas Swing são acionados por eventos; por exemplo, quando um usuário interage com um componente, um evento é gerado. O evento é passado para o aplicativo com uma chamada a um tratador de eventos que o aplicativo define. No entanto, o tratador é executado na thread de despacho do evento fornecida pelo Swing e não na thread principal do aplicativo. Logo, embora os tratadores de eventos sejam definidos pelo programa, eles são chamados em uma thread que não foi criada por ele. Para evitarmos problemas (como duas threads diferentes tentando atualizar o mesmo componente ao mesmo tempo), todos os componentes de GUI de Swing devem ser criados e atualizados a partir da thread de despacho do evento e não da thread principal do aplicativo, mas **main()** é executado na thread principal. Portanto, **main()** não pode instanciar diretamente um objeto **SwingDemo**. Em vez disso, deve criar um objeto **Runnable** para ser executado na thread de despacho do evento e fazer esse objeto criar a GUI.

Para permitir que o código da GUI seja criado no thread de despacho de evento, você deve usar dois métodos definidos pela classe **SwingUtilities**. Os métodos são **invokeLater()** e **invokeAndWait()**. Eles são mostrados abaixo:

```
static void invokeLater(Runnable obj)
static void invokeAndWait(Runnable obj)
    throws InterruptedException, InvocationTargetException
```

Aqui, *obj* é um objeto **Runnable** que terá seu método **run()** chamado pela thread de despacho de evento. A diferença entre os dois métodos é que **invokeLater()** retorna imediatamente e **invokeAndWait()** espera até *obj.run()* retornar. Você pode usar esses métodos para chamar um método que construa a GUI de seu aplicativo Swing ou sempre que precisar modificar o estado da GUI a partir de código não executado pela thread de despacho de evento. Normalmente, vai querer usar **invokeLater()**, como faz o programa anterior. No entanto, quando estiver construindo a GUI inicial de um applet Swing, pode querer usar **invokeAndWait()**.

Para concluir, observe a sintaxe que cria o novo **Runnable**. Ela cria uma classe interna anônima que faz um objeto **Runnable** não nomeado ser gerado. Examinaremos com mais detalhes as classes internas anônimas posteriormente neste capítulo.

Pergunte ao especialista

P O que é o método é **getContentPane()**? Preciso usá-lo?

R Essa pergunta nos leva a uma questão histórica importante. Antes do JDK 5, ao adicionar um componente ao painel de conteúdo, não podíamos chamar o método **add()** diretamente em uma instância de **JFrame**. Em vez disso, tínhamos que chamar **add()** explicitamente no painel de conteúdo do objeto **JFrame**. O painel de conteúdo pode ser obtido com uma chamada a **getContentPane()** em uma instância de **JFrame**. O método **getContentPane()** é mostrado abaixo:

```
Container getContentPane()
```

Pergunte ao especialista (continuação)

Ele retorna uma referência **Container** do painel de conteúdo. O método **add()** era então chamado nessa referência para adicionar um componente ao painel. Logo, no passado, você teria que usar a instrução a seguir para adicionar **jlab** a **jfrm**:

```
| jfrm.getContentPane().add(jlab); // estilo antigo
```

Aqui, primeiro **getContentPane()** obtém uma referência ao painel de conteúdo e então **add()** adiciona o componente ao contêiner vinculado a esse painel. Esse mesmo procedimento também era requerido em chamadas a **remove()** para a remoção de um componente e a **setLayout()** para a definição do gerenciador de leiaute do painel de conteúdo. Você verá chamadas explícitas a **getContentPane()** com frequência em códigos anteriores à versão 5.0. Atualmente, não é mais necessário usar **getContentPane()**, você pode chamar apenas **add()**, **remove()** e **setLayout()** diretamente em **JFrame**, porque esses métodos foram alterados para operarem automaticamente com o painel de conteúdo.

Verificação do progresso

1. Que classe de Swing cria um rótulo?
 2. Que pacote deve ser incluído em todos os programas Swing?
 3. Qualquer código que crie ou modifique a GUI deve ser executado na thread
-

Pergunte ao especialista

P Anteriormente, você disse que é possível adicionar um componente a outras regiões de um leiaute de borda com o uso de uma versão sobrecarregada de **add()**. Pode explicar?

R Como explicado, **BorderLayout** implementa um estilo de leiaute que define cinco locais aos quais um componente pode ser adicionado. O primeiro é o centro, os outros quatro são os lados (isto é, bordas), que se chamam norte, sul, leste e oeste. Por padrão, quando você adicionar um componente ao painel de conteúdo, o adicionará ao centro. Para especificar um dos outros locais, use a seguinte forma de **add()**:

```
void add(Component comp, Object loc)
```

Nessa versão, *comp* é o componente a ser adicionado e *loc* especifica o local de inserção. O valor de *loc* pode ser um dos listados a seguir:

BorderLayout.CENTER	BorderLayout.EAST	BorderLayout.NORTH
BorderLayout.SOUTH	BorderLayout.WEST	

Respostas:

1. **JLabel**
2. **javax.swing**
3. de despacho de evento

Em geral, **BorderLayout** é mais útil na criação de um **JFrame** contendo um componente centralizado (que pode ser um grupo de componentes contido dentro de um dos contêineres leves de Swing) com um componente de cabeçalho e/ou rodapé associado. Em outras situações, um dos outros gerenciadores de layout Java, como **FlowLayout**, será mais apropriado.

TRATAMENTO DE EVENTOS

O exemplo anterior mostrou a forma básica de um programa Swing, mas deixou de fora uma parte importante: o tratamento de eventos. Já que **JLabel** não recebe entrada do usuário, não gera eventos, logo, não foi necessário tratá-los. No entanto, os outros controles de Swing *respondem* a entradas do usuário, e os eventos gerados por essas interações têm que ser tratados. Por exemplo, um evento é gerado quando o usuário clica em um botão, pressiona uma tecla no teclado ou seleciona um item em uma lista. Eventos também são gerados de maneiras não diretamente relacionadas à entrada do usuário. Por exemplo, um evento é gerado quando o tempo de um timer expira. Seja qual for o caso, o tratamento de eventos é uma parte importante de qualquer aplicativo baseado em Swing.

O mecanismo de tratamento de eventos usado por Swing se baseia em uma abordagem chamada *modelo de delegação de eventos*. Seu conceito é muito simples: uma *fonte* gera um evento e o envia para um ou mais *ouvintes*. Nesse esquema, o ouvinte apenas espera até receber um evento. Ao recebê-lo, ele processa o evento e, então, retorna. A vantagem desse projeto é que a lógica que processa eventos fica claramente separada da lógica da interface de usuário que os gera. Um elemento da interface de usuário pode “delegar” o processamento de um evento para um código separado. No modelo de delegação de eventos, os ouvintes devem se registrar em uma fonte para receber a notificação de um evento. Se esse mecanismo está lhe parecendo conhecido, é porque deveria mesmo parecer. Ele é, essencialmente, o padrão **Observer** descrito no Capítulo 16.

Examinemos os eventos, as fontes e os ouvintes com um pouco mais de detalhes.

Eventos

No modelo de delegação, o evento é um objeto que descreve uma mudança de estado em uma fonte. Ele pode ser gerado como consequência de uma pessoa estar em interação com o controle de uma interface gráfica de usuário ou ser criado sob controle do programa. A superclasse de todos os eventos é **java.util.EventObject**. Muitos eventos são declarados em **java.awt.event**. Outros são encontrados em **javax.swing.event**.

Fontes de eventos

Uma fonte de evento é um objeto que gera um evento. Quando uma fonte gera um evento, ela deve enviá-lo para todos os ouvintes registrados. Portanto, para um ouvinte receber um evento, deve se registrar na fonte desse evento. Os ouvintes se registram em uma fonte chamando um método **addTipoListener()** no objeto de fonte do evento. Cada tipo de evento tem seu próprio método de registro. Esta é a forma geral:

```
public void addTipoListener(TipoListener el)
```

Aqui, *Tipo* é o nome do evento e *el* é uma referência ao ouvinte do evento. Por exemplo, o método que registra um ouvinte que recebe eventos de pressionamento de teclas no teclado se chama **addKeyListener()**. O método que registra um ouvinte de movimento do mouse se chama **addMouseMotionListener()**. Quando ocorre um evento, todos os ouvintes registrados são notificados.

A fonte também deve fornecer um método que permita ao ouvinte cancelar o interesse em um tipo específico de evento. Esta é a forma geral do método:

```
public void removeTipoListener(TipoListener el)
```

Aqui, *Tipo* é o nome do evento e *el* é uma referência ao ouvinte do evento. Por exemplo, para remover um ouvinte de pressionamento de teclas, você chamaria **removeKeyListener()**.

Os métodos que adicionam ou removem ouvintes são fornecidos pela fonte que gera os eventos. Por exemplo, a classe **JButton**, que dá suporte a botões de ação, fornece métodos para a inclusão e a remoção de um *ouvinte de ação*, que é notificado quando o botão é pressionado.

Ouvintes de eventos

Um ouvinte é um objeto que é notificado quando um evento ocorre. Ele tem dois requisitos principais. Em primeiro lugar, precisa ter sido registrado em uma ou mais fontes para receber notificações sobre um tipo de evento específico. Em segundo lugar, deve implementar métodos para o recebimento e o processamento desse evento.

Os métodos que recebem e processam eventos são definidos em um conjunto de interfaces. As que usaremos ficam em **java.awt.event** e **javax.swing.event**. Por exemplo, a interface **ActionListener** define um método que recebe notificação quando uma ação, como um clique no botão, ocorre. Qualquer objeto pode receber e processar esse evento se fornecer uma implementação da interface **ActionListener**.

Há um princípio geral importante que deve ser mencionado agora. Um tratador de eventos deve executar sua tarefa rapidamente e retornar. Na maioria dos casos, ele não deve executar uma operação longa porque isso retardaria o aplicativo inteiro, que deixaria de responder. Quando uma operação demorada é necessária, geralmente uma thread separada é criada para esse fim.

Classes de eventos e interfaces de ouvintes

As classes que representam eventos formam a base do mecanismo Java de tratamento de eventos. Na raiz da hierarquia Java de classes de eventos está **EventObject**, que faz parte de **java.util**. Ela é a superclasse de todos os eventos. A classe **AWT-Event**, declarada no pacote **java.awt**, é subclasse de **EventObject**. Ela é (direta ou indiretamente) a superclasse de todos os eventos baseados no AWT usados pelo modelo de delegação de eventos. Swing usa os eventos do AWT e adiciona vários eventos próprios.

A Tabela 17-1 mostra um resumo das classes de eventos e interfaces de ouvintes definidas em **java.awt.event**, e a Tabela 17-2 mostra um resumo das definidas em **javax.swing.event**. As classes e interfaces de eventos usadas pelos exemplos serão descritas quando necessário.

Tabela 17-1 Resumo das classes de eventos de `java.awt.event`

Classe de eventos	Descrição	Ouvinte de eventos correspondente
ActionEvent	Gerado quando uma ação ocorre dentro de um controle, como quando um botão é pressionado.	ActionListener
AdjustmentEvent	Gerado quando uma barra de rolagem é manipulada.	AdjustmentListener
FocusEvent	Gerado quando um componente ganha ou perde o foco.	FocusListener
ItemEvent	Gerado quando um item é selecionado, como quando uma caixa de seleção é clicada.	ItemListener
KeyEvent	Gerado quando uma entrada do teclado é recebida.	KeyListener
MouseEvent	Gerado quando o mouse é arrastado ou movido, ou seu botão é clicado, pressionado ou solto; também é gerado quando o mouse entra ou sai de um componente.	MouseListener e MouseMotionListener
MouseWheelEvent	Gerado quando a roda do mouse é movida.	MouseWheelListener
WindowEvent	Gerado quando uma janela é ativada, fechada, desativada, deiconificada, iconificada, aberta ou abandonada.	WindowListener

Tabela 17-2 Resumo das classes de eventos de `javax.swing.event`

Classe de eventos	Descrição	Ouvinte de eventos correspondente
AncestorEvent	Gerado quando o predecessor de um componente tiver sido adicionado, movido ou removido.	AncestorListener
CaretEvent	Gerado quando a posição do circunflexo muda em um componente de texto.	CaretListener
ChangeEvent	Gerado quando um componente muda seu estado.	ChangeListener
HyperlinkEvent	Gerado quando um hiperlink é acessado.	HyperlinkListener
ListDataEvent	Gerado quando o conteúdo de uma lista muda.	ListDataListener
ListSelectionEvent	Gerado quando a seleção em uma lista muda.	ListSelectionListener
MenuEvent	Gerado quando ocorre uma seleção no menu.	MenuListener
TableModelEvent	Gerado quando o modelo de tabela muda.	TableModelListener
TreeExpansionEvent	Gerado quando uma árvore é expandida ou recolhida.	TreeExpansionListener
TreeModelEvent	Gerado quando um modelo de árvore muda.	TreeModelListener
TreeSelectionEvent	Gerado quando um nó de uma árvore é selecionado.	TreeSelectionListener

Classes adaptadoras

Embora não seja difícil implementar a maioria das interfaces de ouvintes de eventos, Java oferece um conjunto de *classes adaptadoras* que fornecem uma implementação vazia de seus métodos. As classes adaptadoras são úteis quando queremos receber e processar apenas alguns dos eventos associados a uma interface de eventos específica. Podemos definir uma nova classe para agir como um ouvinte de eventos estendendo uma das classes adaptadoras e implementar só os métodos em que estivermos interessados. Não ser preciso implementar todos os métodos definidos por uma interface de ouvinte de eventos economiza bastante trabalho e impede que o código fique cheio de métodos vazios. Também é comum a implementação de uma classe adaptadora com o uso de uma classe interna anônima, que pode simplificar ainda mais o código.

Nem todas as interfaces de ouvintes têm classes adaptadoras correspondentes. Por exemplo, não há classe adaptadora para **ActionListener** porque ela define apenas um método. Em geral, há classes adaptadoras para ouvintes que definem dois ou mais métodos. Por exemplo, a classe **MouseMotionListener** tem dois métodos, **mouseDragged()** e **mouseMoved()**. Implementações vazias desses métodos são fornecidas por **MouseMotionAdapter**. Se você estivesse interessado apenas em eventos de arrastar o mouse, poderia simplesmente estender **MouseMotionAdapter** e implementar **mouseDragged()**. A implementação vazia de **mouseMoved()** trataria os eventos de movimentação do mouse para você.

Aqui está um resumo das classes adaptadoras. A maioria está definida em **java.awt.event**, mas **MouseInputAdapter** fica em **javax.swing.event**.

Classe adaptadora	Implementa
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
MouseInputAdapter	MouseListener e MouseMotionListener
WindowAdapter	WindowListener

Verificação do progresso

- O modelo de delegação de eventos é baseado em fontes de eventos e _____.
- Qual é o nome da interface de ouvinte de eventos de ação?
- O que as classes adaptadoras fazem?

Respostas:

- ouvintes de eventos
- ActionListener**
- As classes adaptadoras simplificam a implementação de ouvintes de eventos fornecendo implementações vazias de todos os métodos da interface. Logo, só precisamos implementar os métodos em que estamos interessados.

USANDO UM BOTÃO DE AÇÃO

Um dos controles mais simples de Swing é o botão de ação. Ele também é um dos mais usados. Um botão de ação é uma instância de **JButton**. **JButton** herda a classe abstrata **AbstractButton**, que define a funcionalidade comum a todos os botões. Os botões de ação de Swing podem conter texto, imagem ou ambos. Aqui só usaremos botões de ação baseados em texto, mas outros tipos de botões serão discutidos no Capítulo 18.

JButton fornece vários construtores. O usado aqui é
JButton(String msg)

O parâmetro *msg* especifica o string que será exibido dentro do botão.

Quando um botão de ação é pressionado, ele gera um **ActionEvent**. Logo, **JButton** fornece os métodos a seguir (herdados de **AbstractButton**), que são usados para adicionar ou remover o ouvinte de uma ação:

```
void addActionListener (ActionListener al)
void removeActionListener (ActionListener al)
```

Aqui, *al* especifica um objeto que receberá notificações de eventos. Esse objeto deve ser instância de uma classe que implemente a interface **ActionListener**.

A interface **ActionListener** só define um método: **actionPerformed()**. Ele é mostrado abaixo:

```
void actionPerformed(ActionEvent ae)
```

Esse método é chamado quando um botão é pressionado. Em outras palavras, ele é o tratador de eventos chamado quando ocorre um evento de pressionamento de botão. A implementação de **actionPerformed()** deve responder rapidamente ao evento e retornar. Como regra geral, conforme mencionado, os tratadores de eventos não devem se ocupar de operações longas porque isso retarda o aplicativo inteiro. Se um procedimento demorado precisar ser executado, uma thread separada deve ser criada para esse fim.

Usando o objeto **ActionEvent** passado para **actionPerformed()**, você pode obter várias informações úteis relacionadas ao evento de pressionamento de botão. A usada por este capítulo é o *string do comando de ação* associado ao botão. Por padrão, esse é o string exibido dentro do botão. O comando de ação é obtido com uma chamada ao método **getActionCommand()** no objeto de evento. Esta é sua declaração:

```
String getActionCommand()
```

O comando de ação identifica o botão, logo, quando são usados dois ou mais botões dentro do mesmo aplicativo, o comando de ação fornece uma maneira fácil de determinarmos qual botão foi pressionado.

O programa a seguir demonstra como criar um botão de ação e responder a eventos de pressionamento de botão. A Figura 17-2 mostra como o exemplo aparece na tela.

```
// Demonstra um botão.

import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;  
  
class ButtonDemo implements ActionListener {  
  
    JLabel jlab;  
  
    ButtonDemo() {  
  
        // Cria um contêiner JFrame.  
        JFrame jfrm = new JFrame("A Button Example");  
  
        // Especifica FlowLayout como gerenciador de layout.  
        jfrm.setLayout(new FlowLayout()); ← Observe o uso de FlowLayout.  
  
        // Fornece um tamanho inicial para o quadro.  
        jfrm.setSize(220, 90);  
  
        // Encerra o programa quando o usuário fecha o aplicativo.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Cria dois botões.  
        JButton jbtnFirst = new JButton("First"); ← Cria dois botões de ação.  
        JButton jbtnSecond = new JButton("Second"); ←  
  
        // Adiciona ouvintes de ação.  
        jbtnFirst.addActionListener(this); ← Adiciona ouvintes de  
        jbtnSecond.addActionListener(this); ← ação para os botões.  
  
        // Adiciona os botões ao painel de conteúdo.  
        jfrm.add(jbtnFirst); ← Adiciona os botões ao painel de conteúdo.  
        jfrm.add(jbtnSecond); ←  
  
        // Cria um rótulo baseado em texto.  
        jlab = new JLabel("Press a button.");  
  
        // Adiciona o rótulo ao quadro.  
        jfrm.add(jlab);  
  
        // Exibe o quadro.  
        jfrm.setVisible(true);  
    }  
  
    // Trata eventos de botão.  
    public void actionPerformed(ActionEvent ae) { ← Este é o tratador de  
        if(ae.getActionCommand().equals("First")) ← ActionEvent para  
            jlab.setText("First button was pressed."); ← os botões de ação.  
        else  
            jlab.setText("Second button was pressed. "); ←  
    }  
  
    public static void main(String[] args) {
```

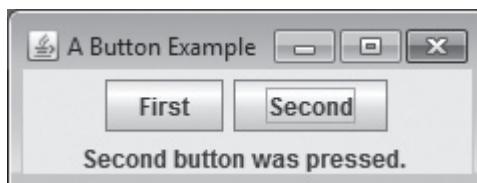


Figura 17-2 Saída do programa de demonstração de JButton.

```
// Cria o quadro na thread de despacho de evento.
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new ButtonDemo();
    }
});
```

Examinemos detalhadamente o que há de novo nesse programa. Primeiro, observe que, além de **javax.swing**, agora ele importa tanto o pacote **java.awt** quanto **java.awt.event**. O pacote **java.awt** é necessário porque contém a classe **FlowLayout**, que dá suporte ao gerenciador de leiaute de fluxo padrão usado na disposição de componentes em um quadro. O pacote **java.awt.event** é necessário porque define a interface **ActionListener** e a classe **ActionEvent**.

Em seguida, a classe **ButtonDemo** é declarada. Observe que ela implementa **ActionListener**, ou seja, objetos **ButtonDemo** podem ser usados para receber eventos de ação. Depois, uma referência **JLabel** é declarada. Essa referência será usada dentro do método **actionPerformed()** para exibir qual botão foi pressionado.

O construtor de **ButtonDemo** começa criando um **JFrame** chamado **jfrm**. Ele então define o gerenciador de leiaute do painel de conteúdo de **jfrm** como **FlowLayout**, como mostrado aqui:

```
| jfrm.setLayout(new FlowLayout());
```

Como explicado anteriormente, por padrão, o painel de conteúdo usa **BorderLayout** como seu gerenciador de leiaute, mas, para muitos aplicativos, **FlowLayout** é mais conveniente. Lembre-se, um leiaute de fluxo dispõe os componentes em uma “linha” de cada vez, de cima para baixo. Quando uma “linha” fica cheia, o leiaute avança para a próxima “linha”. Embora esse esquema forneça pouco controle sobre a inserção de componentes, ele é muito fácil de usar. No entanto, cuidado ao redimensionar o quadro, porque a posição dos componentes mudará.

Após definir o tamanho e a operação de fechamento padrão, **ButtonDemo()** cria dois botões, como mostrado aqui:

```
| JButton jbtnFirst = new JButton("First");
| JButton jbtnSecond = new JButton("Second");
```

O primeiro botão conterá o texto “First” e o segundo conterá “Second”.

Agora, a instância de **ButtonDemo** referenciada via **this** é adicionada ao ouvinte de ação dos botões dessas duas linhas:

```
jbtnFirst.addActionListener(this);  
jbtnSecond.addActionListener(this);
```

Essa abordagem significa que o objeto que criou os botões também receberá notificações quando um botão for pressionado.

Em seguida, os botões são adicionados ao painel de conteúdo de **jfrm**:

```
jfrm.add(jbtnFirst);  
jfrm.add(jbtnSecond);
```

Ou seja, os botões serão exibidos na janela fornecida por **jfrm**.

Sempre que um botão é pressionado, ele gera um evento de ação e todos os ouvintes registrados são notificados com uma chamada ao método **actionPerformed()**. O objeto **ActionEvent** que representa o evento de botão é passado como parâmetro. No caso de **ButtonDemo**, o evento é passado para esta implementação de **actionPerformed()**:

```
public void actionPerformed(ActionEvent ae) {  
    if(ae.getActionCommand().equals("First"))  
        jlab.setText("First button was pressed.");  
    else  
        jlab.setText("Second button was pressed. ");  
}
```

O evento que ocorreu é passado via **ae**. Dentro do método, o comando de ação associado ao botão que gerou o evento é obtido com uma chamada a **getActionCommand()**. (Lembre-se, por padrão, o comando de ação é igual ao texto exibido no botão.) Com base no conteúdo desse string, o texto do rótulo é configurado apropriadamente com o uso de **setText()**. Esse método é definido por **JLabel** como mostrado abaixo:

Void setText(String msg)

Aqui, *msg* especifica o texto que será exibido dentro do rótulo. O método **setText()** nos permite definir o texto que aparecerá dentro de um rótulo depois que ele for criado.

Um último ponto: lembre-se de que **actionPerformed()** é chamado na thread de despacho de evento, como já explicado. Ele deve retornar rapidamente para evitar retardar o aplicativo.

Verificação do progresso

1. Que classe cria um botão de ação de Swing?
2. Por padrão, o string do comando de ação associado a um botão é igual ao texto exibido dentro do botão. Verdadeiro ou falso?

Respostas:

1. **JButton**
2. Verdadeiro.

TENTE ISTO 17-1 Um cronômetro simples

StopWatch.java

Embora só dois controles de Swing tenham sido introduzidos, **JLabel** e **JButton**, você pode usá-los para criar um aplicativo totalmente funcional e útil: um cronômetro. O cronômetro contém dois botões de ação e um rótulo. Os botões de ação se chamam Start e Stop e são usados para iniciar e interromper o cronômetro. O rótulo exibe o tempo decorrido. Embora muito simples, esse projeto mostra a facilidade com que interfaces de GUI podem ser criadas com o uso de Swing.

PASSO A PASSO

1. Comece criando um arquivo chamado **StopWatch.java** e insira o comentário e as instruções **import** a seguir:

```
// Tente isto 17-1: um cronômetro simples.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
```

Observe que **java.util** é importado. Esse pacote é necessário porque contém a classe **Calendar**, que é usada na obtenção da hora atual do sistema.

2. Comece a classe **StopWatch** como mostrado aqui:

```
class StopWatch implements ActionListener {

    JLabel jlab;
    long start; // contém a hora inicial em milissegundos
```

Como o comentário indica, o campo **start** é usado para conter a hora inicial em milissegundos. Esse valor será subtraído da hora final para obtenção do tempo decorrido.

3. Comece o construtor de **StopWatch** com as linhas a seguir:

```
StopWatch() {
    // Cria um contêiner JFrame.
    JFrame jfrm = new JFrame("A Simple Stopwatch");

    // Especifica FlowLayout como gerenciador de leiaute.
    jfrm.setLayout(new FlowLayout());

    // Fornece um tamanho inicial para o quadro.
    jfrm.setSize(230, 90);

    // Encerra o programa quando o usuário fecha o aplicativo.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Essas instruções, como você deve lembrar, são semelhantes às usadas pelos exemplos anteriores.

4. Insira o código abaixo, que cria os botões Start e Stop, adiciona ouvintes de ação para os botões e então adiciona os botões ao painel de conteúdo.

```
// Cria dois botões.
 JButton jbtnStart = new JButton("Start");
 JButton jbtnStop = new JButton("Stop");

// Adiciona ouvintes de ação.
 jbtnStart.addActionListener(this);
 jbtnStop.addActionListener(this);

// Adiciona os botões ao painel de conteúdo.
 jfrm.add(jbtnStart);
 jfrm.add(jbtnStop);
```

5. Crie e adicione um rótulo usando as instruções a seguir:

```
// Cria um rótulo baseado em texto.
 jlab = new JLabel("Press Start to begin timing.");

// Adiciona o rótulo ao quadro.
 jfrm.add(jlab);
```

O rótulo é usado para indicar o status do cronômetro e exibir o tempo decorrido.

6. Termine o construtor de **StopWatch** tornando o quadro visível:

```
// Exibe o quadro.
 jfrm.setVisible(true);
}
```

7. Adicione o método **actionPerformed()** mostrado aqui:

```
// Trata eventos de botão.
public void actionPerformed(ActionEvent ae) {
    // obtém a hora atual do sistema
    Calendar cal = Calendar.getInstance();

    if(ae.getActionCommand().equals("Start")) {
        // Armazena a hora inicial.
        start = cal.getTimeInMillis();
        jlab.setText("Stopwatch is Running...");
    }
    else
        // Calcula o tempo decorrido.
        jlab.setText("Elapsed time is "
            + (double) (cal.getTimeInMillis() - start)/1000);
}
```

Observe que um objeto **Calendar** chamado **cal** é criado e inicializado com a hora atual do sistema com uma chamada ao método estático **getInstance()** de **Calendar**. Logo, sempre que **actionPerformed()** for chamado, **cal** será inicializado com a hora atual do sistema. (Para obter informações sobre **Calendar**, consulte o Capítulo 24.)

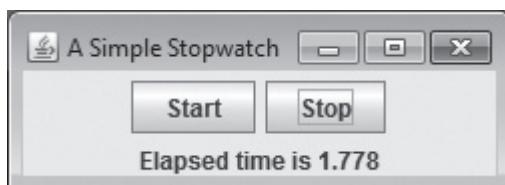
Lembre-se, por padrão, o comando de ação de um botão é o texto exibido por ele. Portanto, o comando de ação do botão Start é “Start”. Quando o botão Start é pressionado, a hora atual (em milissegundos) é obtida (com uma chamada a **getTimeInMillis()** em **cal**) e armazenada no campo **start**. Quando o botão Stop é pressionado, a hora atual é obtida e a hora inicial é subtraída dela, gerando o tempo decorrido. Esse valor é convertido para **double** e dividido por 1.000. Isso converte o tempo decorrido em segundos.

8. Termine adicionando este método **main()**:

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

    // Cria o quadro na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new StopWatch();
        }
    });
}
```

9. Aqui está todo o código reunido em um programa completo. Um exemplo da saída é mostrado abaixo:



```
// Tente isto 17-1: um cronômetro simples.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

class StopWatch implements ActionListener {

    JLabel jlab;
    long start; // contém a hora inicial em milissegundos

    StopWatch() {
```

```
// Cria um contêiner JFrame.  
JFrame jfrm = new JFrame("A Simple Stopwatch");  
  
// Especifica FlowLayout como gerenciador de leiaute.  
jfrm.setLayout(new FlowLayout());  
  
// Fornece um tamanho inicial para o quadro.  
jfrm.setSize(230, 90);  
  
// Encerra o programa quando o usuário fecha o aplicativo.  
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
// Cria dois botões.  
JButton jbtnStart = new JButton("Start");  
JButton jbtnStop = new JButton("Stop");  
  
// Adiciona ouvintes de ação.  
jbtnStart.addActionListener(this);  
jbtnStop.addActionListener(this);  
  
// Adiciona os botões ao painel de conteúdo.  
jfrm.add(jbtnStart);  
jfrm.add(jbtnStop);  
  
// Cria um rótulo baseado em texto.  
jlab = new JLabel("Press Start to begin timing.");  
  
// Adiciona o rótulo ao quadro.  
jfrm.add(jlab);  
  
// Exibe o quadro.  
jfrm.setVisible(true);  
}  
  
// Trata eventos de botão.  
public void actionPerformed(ActionEvent ae) {  
    // obtém a hora atual do sistema  
    Calendar cal = Calendar.getInstance();  
  
    if(ae.getActionCommand().equals("Start")) {  
        // Armazena a hora inicial.  
        start = cal.getTimeInMillis();  
        jlab.setText("Stopwatch is Running...");  
    }  
    else  
        // Calcula o tempo decorrido.  
        jlab.setText("Elapsed time is "  
            + (double) (cal.getTimeInMillis() - start)/1000);  
}  
  
public static void main(String[] args) {
```

```
// Cria o quadro na thread de despacho de evento.
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new StopWatch();
    }
});
}
```

INTRODUÇÃO AO JTextField

Outro controle muito usado é **JTextField**. Ele permite que o usuário insira uma linha de texto. **JTextField** herda a classe abstrata **JTextComponent**, que é a superclasse de todos os componentes de texto, e fornece uma maneira conveniente de obtermos entradas do usuário baseadas em texto.

JTextField define vários construtores. O que usaremos é mostrado abaixo:

`JTextField(int cols)`

Aqui, *cols* especifica a largura do campo de texto em colunas. É importante entender que é possível inserir um string maior do que o número de colunas. O tamanho físico do campo de texto na tela é que terá *cols* colunas de largura.

Quando um usuário pressiona ENTER ao digitar em um campo de texto, um **ActionEvent** é gerado. Portanto, **JTextField** fornece os métodos **addActionListener()** e **removeActionListener()**. Para tratar eventos de ação, você deve implementar o método **actionPerformed()** definido pela interface **ActionListener**. O processo é semelhante ao tratamento de eventos de ação gerados por um botão, como descrito anteriormente.

Para obter o string exibido atualmente no campo de texto, chame o método **getText()** na instância de **JTextField**. Ele é declarado assim:

`String getText()`

Você pode configurar o texto de **JTextField** chamando **setText()**, como mostrado abaixo:

`void setText(String texto)`

Aqui, *texto* é o string que será inserido no campo de texto.

O programa a seguir demonstra **JTextField**. Ele cria um campo de texto com 10 colunas de largura. Sempre que você pressionar ENTER enquanto estiver no campo de texto, o conteúdo atual será exibido via um **JLabel**. Seu modo de operar já deve estar claro. Um exemplo da saída é mostrado na Figura 17-3

```
// Demonstra um campo de texto.

import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;  
  
class JTextFieldDemo implements ActionListener {  
  
    JTextField jtf;  
    JLabel jlab;  
  
    JTextFieldDemo() {  
  
        // Cria um contêiner JFrame.  
        JFrame jfrm = new JFrame("A Text Field Example");  
  
        // Especifica FlowLayout como gerenciador de leiaute.  
        jfrm.setLayout(new FlowLayout());  
  
        // Fornece um tamanho inicial para o quadro.  
        jfrm.setSize(240, 90);  
  
        // Encerra o programa quando o usuário fecha o aplicativo.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Cria uma instância de campo de texto.  
        jtf = new JTextField(10); ← Cria um campo de texto com 10 colunas.  
  
        // Adiciona um ouvinte de ação para o campo de texto.  
        jtf.addActionListener(this); ← Define o ouvinte de ação para o campo de texto.  
  
        // Adiciona o campo de texto ao painel de conteúdo.  
        jfrm.add(jtf);  
  
        // Cria um rótulo vazio baseado em texto.  
        jlab = new JLabel("");  
  
        // Adiciona o rótulo ao quadro.  
        jfrm.add(jlab);  
  
        // Exibe o quadro.  
        jfrm.setVisible(true);  
    }  
  
    // Trata eventos de ação.  
    public void actionPerformed(ActionEvent ae) { ← Um evento de ação é  
        // Obtém o texto atual e o exibe em um rótulo.  
        jlab.setText("Current contents: " + jtf.getText());  
    }  
  
    public static void main(String[] args) {
```

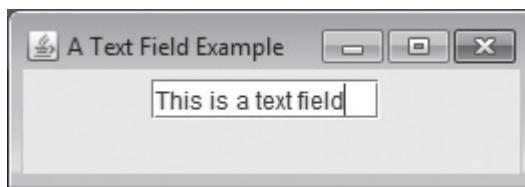


Figura 17-3 Exemplo de saída do programa **JTextField**.

```
// Cria o quadro no thread de despacho de evento.
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new JTextFieldDemo();
    }
});
```

Você deve reconhecer grande parte do programa, mas observe esta linha do método **actionPerformed()**:

```
| jlab.setText("Current contents: " + jtf.getText());
```

Como explicado, quando o usuário pressiona ENTER, um **ActionEvent** é gerado e enviado para todos os ouvintes de ação registrados, por intermédio do método **actionPerformed()**. Em **TextFieldDemo**, esse método apenas obtém o texto contido atualmente no campo chamando **getText()** em **jtf**. Em seguida, exibe o texto usando o rótulo referenciado por **jlab**.

Assim como **JButton**, **JTextField** tem um string de comando de ação associado. Por padrão, o comando de ação é o conteúdo atual do campo de texto, no entanto, você pode configurá-lo com um comando de ação de sua escolha chamando o método **setActionCommand()**, mostrado aqui:

```
void setActionCommand(String cmd)
```

O string passado em **cmd** é o novo comando de ação. O texto do campo não é afetado. Uma vez que você configurar o string do comando de ação, ele permanecerá o mesmo, não importando o que foi inserido no campo de texto. Você pode querer configurar explicitamente o comando de ação para fornecer uma maneira de o campo de texto ser reconhecido como fonte de um evento de ação. Isso será útil quando outro controle do mesmo quadro também gerar eventos de ação e você quiser usar o mesmo tratador para processar os dois eventos. A configuração do comando de ação proporciona uma maneira de diferenciá-los.

Por exemplo, no programa a seguir, são usados dois campos de texto que podem ser reconhecidos por seu comando de ação.

```
// Usa dois campos de texto.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



```

jlab.setText("ENTER pressed in tf1: "
+ jtf1.getText());
else
    jlab.setText("ENTER pressed in jtf2: "
+ jtf2.getText());
}

public static void main(String[] args) {

    // Cria o quadro na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new TwoTFDemo();
        }
    });
}
}

```

Um exemplo da saída é mostrado na Figura 17-4.

Esse programa cria dois **JTextFields**: **jtf1** e **jtf2**. Observe que os comandos de ação associados a **jtf1** e **jtf2** são definidos com o uso destas linhas de código:

```

jtf1.setActionCommand("One");
jtf2.setActionCommand("Two");

```

Em seguida, dentro de **actionPerformed()**, observe como o comando de ação é usado na determinação do campo de texto que gerou o evento de ação:

```

public void actionPerformed(ActionEvent ae) {

    if(ae.getActionCommand().equals("One"))
        jlab.setText("ENTER pressed in tf1: "
                    + jtf1.getText());
    else
        jlab.setText("ENTER pressed in jtf2: "
                    + jtf2.getText());
}

```

Já que os comandos de ação de cada campo de texto foram configurados com um string conhecido, esse string pode ser usado na determinação do campo de texto que gerou o evento.

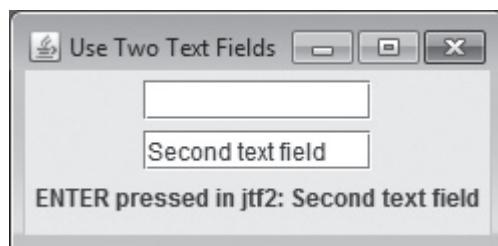


Figura 17-4 Exemplo de saída do programa dos dois campos de texto.

Pergunte ao especialista

P Você explicou que o comando de ação associado a um campo de texto pode ser configurado com uma chamada a `setActionCommand()`. Posso usar esse método para configurar o comando de ação associado a um botão de ação?

R Sim. Como você sabe, por padrão, o comando de ação associado a um botão de ação é o nome do botão. Para configurar o comando de ação com um valor diferente, você pode usar o método `setActionCommand()`. Funciona da mesma forma para `JButton` e `JTextField`.

Verificação do progresso

1. No uso de `JTextField`, se o texto inserido pelo usuário for maior do que o campo de texto, ele será truncado. Verdadeiro ou falso?
2. Quando o usuário pressiona ENTER dentro de um `JTextField`, que evento é gerado?
3. Por que é uma boa ideia configurar explicitamente o comando de ação de um `JTextField`?

TENTE ISTO 17-2 Crie uma máquina de codificação simples

Coder.java

Este projeto usa **JLabel**, **JButton** e **JTextField** para criar uma máquina de codificação simples. A máquina funciona implementando uma codificação substituta em que cada caractere de um string é substituído pelo próximo caractere. Por exemplo, A passa a ser B, B passa a ser C, e assim por diante. É claro que uma abordagem tão simples é fácil de burlar, mas ela fornece um meio interessante de ilustrar as maneiras como os botões de ação e campos de texto podem interagir. Também mostra como eventos gerados por componentes diferentes (nesse caso, um campo de texto e um botão) podem ser mapeados para o mesmo tratador de eventos.

Respostas:

1. Falso. A largura do texto pode exceder a largura do campo.
2. **ActionEvent**
3. Se o comando de ação não for configurado, por padrão ele será o texto contido atualmente no campo de texto. A configuração do comando de ação identifica explicitamente o campo de texto, não importando o que ele contém.

PASSO A PASSO

- Crie um arquivo chamado **Coder.java** e insira o comentário e as instruções **import** a seguir:

```
// Tente isto 17-2: Uma máquina de codificação simples.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

- Comece a criar a classe **Coder**, desta forma:

```
class Coder implements ActionListener {

    JTextField jtfPlaintext;
    JTextField jtfCiphertext;
```

Observe que **Coder** implementa **ActionListener**. Ela também declara campos para dois campos de texto. **jtfPlaintext** conterá a mensagem de texto sem codificação inserida pelo usuário. **jtfCiphertext** conterá a versão codificada da mensagem.

- Inicie o construtor de **Coder**, como mostrado aqui:

```
Coder() {

    // Cria um contêiner JFrame.
    JFrame jfrm = new JFrame("A Simple Code Machine");

    // Especifica FlowLayout como gerenciador de layout.
    jfrm.setLayout(new FlowLayout());

    // Fornece um tamanho inicial para o quadro.
    jfrm.setSize(340, 120);

    // Encerra o programa quando o usuário fecha o aplicativo.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Essa sequência é semelhante à usada por outros exemplos deste capítulo e você já deve conhecê-la.

- Adicione esses dois **JLabels**:

```
// Cria dois rótulos.
JLabel jlabPlaintext = new JLabel(" Plain Text: ");
JLabel jlabCiphertext = new JLabel("Cipher Text: ");
```

- Crie duas instâncias de **JTextField** e atribua-as aos campos **jtfPlaintext** e **jtfCiphertext**.

```
// Cria duas instâncias de campo de texto.
jtfPlaintext = new JTextField(20);
jtfCiphertext = new JTextField(20);
```

- 6.** Defina o comando de ação dos campos de texto e adicione **this** como ouvinte de ação dos dois campos.

```
// Define os comandos de ação dos campos de texto.
jtfPlaintext.setActionCommand("Encode");
jtfCiphertext.setActionCommand("Decode");

// Adiciona ouvintes de ação para os campos de texto.
jtfPlaintext.addActionListener(this);
jtfCiphertext.addActionListener(this);
```

É necessário definir o comando de ação das duas instâncias de **JTextField** por três razões. Em primeiro lugar, ele fornece uma maneira de identificar cada campo de texto. Em segundo lugar, se o comando de ação não for configurado, por padrão o texto atual será usado. Se, por acaso, esse texto for igual ao do comando de ação usado por outro controle, surgirá um conflito. A configuração do comando de ação evita isso. Em terceiro lugar, como veremos em breve, esses dois comandos de ação são iguais aos de dois dos botões de ação. Ou seja, a mesma sequência de código pode ser usada no tratamento de eventos tanto do botão de ação quanto do campo de texto.

- 7.** Adicione os campos de texto e rótulos ao painel de conteúdo.

```
// Adiciona os campos de texto e rótulos ao painel de conteúdo.
jfrm.add(jlabPlaintext);
jfrm.add(jtfPlaintext);
jfrm.add(jlabCiphertext);
jfrm.add(jtfCiphertext);
```

A ordem em que esses componentes são adicionados é importante porque os rótulos descrevem os campos de texto.

- 8.** Crie três botões de ação chamados Encode, Decode e Reset, como mostrado aqui:

```
// Cria instâncias de botões de ação.
 JButton jbtnEncode = new JButton("Encode");
 JButton jbtnDecode = new JButton("Decode");
 JButton jbtnReset = new JButton("Reset");
```

- 9.** Adicione **this** como ouvinte de ação dos botões e então adicione os botões ao painel de conteúdo.

```
// Adiciona ouvintes de ação para os botões.
jbtnEncode.addActionListener(this);
jbtnDecode.addActionListener(this);
jbtnReset.addActionListener(this);

// Adiciona os botões ao painel de conteúdo.
jfrm.add(jbtnEncode);
jfrm.add(jbtnDecode);
jfrm.add(jbtnReset);
```

- 10.** Termine o construtor de **Coder** com uma chamada a **setVisible()**, como mostrado abaixo:

```
// Exibe o quadro.  
jfrm.setVisible(true);  
}
```

- 11.** Comece a codificar o método **actionPerformed()**, desta forma:

```
// Trata eventos de ação.  
public void actionPerformed(ActionEvent ae) {  
  
    // Se o comando de ação for "Encode", codifica o string.  
    if(ae.getActionCommand().equals("Encode")) {  
  
        // Essa variável conterá o string codificado.  
        String encStr = "";  
  
        // Obtém o texto sem codificação.  
        String str = jtfPlaintext.getText();  
  
        // Adiciona 1 a cada caractere.  
        for(int i=0; i<str.length(); i++)  
            encStr += (char)(str.charAt(i) + 1);  
  
        // Insere o texto codificado no campo Cipher Text.  
        jtfCiphertext.setText(encStr.toString());  
    }  
}
```

Essa instrução **if** compara o comando de ação com “Encode”. O comando será “Encode” se o usuário tiver pressionado o botão **jbtnEncode** ou se tiver pressionado ENTER ao inserir texto no campo **jtfPlaintext**. Já que o comando de ação tanto de **jbtnEncode** quanto de **jtfPlaintext** é “Encode”, os dois eventos são tratados pelo mesmo código. Em outras palavras, os eventos gerados por esses dois controles são mapeados para o mesmo tratador porque seus comandos de ação são iguais. Esse tratador codifica o string do campo Plain Text e o insere no campo Cipher Text.

- 12.** Adicione a instrução **else if** a seguir, que determina se o comando de ação é igual a “Decode”:

```
// Se o comando de ação for "Decode", decodifica o string.  
else if(ae.getActionCommand().equals("Decode")) {  
  
    // Essa variável conterá o string decodificado.  
    String decStr = "";  
  
    // Obtém o texto codificado.  
    String str = jtfCiphertext.getText();  
  
    // Subtrai 1 de cada caractere.  
    for(int i=0; i<str.length(); i++)
```

```

    decStr += (char) (str.charAt(i) - 1);

    // Insere o texto decodificado no campo Plain Text.
    jtfPlaintext.setText(decStr.toString());
}

```

Esse código funciona como o tratador de “Encode” exceto por decodificar o string do campo Cipher Text e inseri-lo no campo Plain Text.

13. Termine o método **actionPerformed()** tratando o comando de ação “Reset”, que está vinculado a **jbtnReset**. Como há só três comandos de ação, não há necessidade de procurar “Reset” explicitamente. Se a execução alcançar esse ponto, essa será a única opção.

```

// Só sobrou o comando "Reset".
else {
    jtfPlaintext.setText("");
    jtfCiphertext.setText("");
}
}

```

14. Termine adicionando este método **main()**:

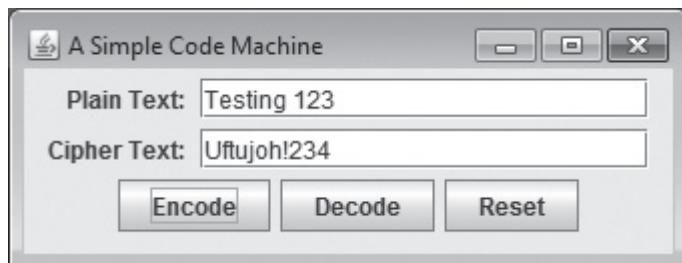
```

public static void main(String[] args) {

    // Cria o quadro na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Coder();
        }
    });
}

```

15. Aqui está todo o código reunido em um programa completo. Um exemplo da saída é mostrado abaixo:



```

// Tente isto 17-2: Uma máquina de codificação simples.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

```
class Coder implements ActionListener {  
  
    JTextField jtfPlaintext;  
    JTextField jtfCiphertext;  
  
    Coder() {  
  
        // Cria um contêiner JFrame.  
        JFrame jfrm = new JFrame("A Simple Code Machine");  
  
        // Especifica FlowLayout como gerenciador de leiaute.  
        jfrm.setLayout(new FlowLayout());  
  
        // Fornece um tamanho inicial para o quadro.  
        jfrm.setSize(340, 130);  
  
        // Encerra o programa quando o usuário fecha o aplicativo.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Cria dois rótulos.  
        JLabel jlabPlaintext = new JLabel("Plain Text: ");  
        JLabel jlabCiphertext = new JLabel("Cipher Text: ");  
  
        // Cria duas instâncias de campo de texto.  
        jtfPlaintext = new JTextField(20);  
        jtfCiphertext = new JTextField(20);  
  
        // Define os comandos de ação dos campos de texto.  
        jtfPlaintext.setActionCommand("Encode");  
        jtfCiphertext.setActionCommand("Decode");  
  
        // Adiciona ouvintes de ação para os campos de texto.  
        jtfPlaintext.addActionListener(this);  
        jtfCiphertext.addActionListener(this);  
  
        // Adiciona os campos de texto e rótulos ao painel de conteúdo.  
        jfrm.add(jlabPlaintext);  
        jfrm.add(jtfPlaintext);  
        jfrm.add(jlabCiphertext);  
        jfrm.add(jtfCiphertext);  
  
        // Cria instâncias de botões de ação.  
        JButton jbtnEncode = new JButton("Encode");  
        JButton jbtnDecode = new JButton("Decode");  
        JButton jbtnReset = new JButton("Reset");  
  
        // Adiciona ouvintes de ação para os botões.  
        jbtnEncode.addActionListener(this);  
        jbtnDecode.addActionListener(this);  
        jbtnReset.addActionListener(this);
```

```
// Adiciona os botões ao painel de conteúdo.  
jfrm.add(jbtnEncode);  
jfrm.add(jbtnDecode);  
jfrm.add(jbtnReset);  
  
// Exibe o quadro.  
jfrm.setVisible(true);  
}  
  
// Trata eventos de ação.  
public void actionPerformed(ActionEvent ae) {  
  
    // Se o comando de ação for "Encode", codifica o string.  
    if(ae.getActionCommand().equals("Encode")) {  
  
        // Essa variável conterá o string codificado.  
        String encStr = "";  
  
        // Obtém o texto sem codificação.  
        String str = jtfPlaintext.getText();  
  
        // Adiciona 1 a cada caractere.  
        for(int i=0; i<str.length(); i++)  
            encStr += (char)(str.charAt(i) + 1);  
  
        // Insere o texto codificado no campo Cipher Text.  
        jtfCiphertext.setText(encStr.toString());  
    }  
  
    // Se o comando de ação for "Decode", decodifica o string.  
    else if(ae.getActionCommand().equals("Decode")) {  
  
        // Essa variável conterá o string decodificado.  
        String decStr = "";  
  
        // Obtém o texto codificado.  
        String str = jtfCiphertext.getText();  
  
        // Subtrai 1 de cada caractere.  
        for(int i=0; i<str.length(); i++)  
            decStr += (char)(str.charAt(i) - 1);  
  
        // Insere o texto decodificado no campo Plain Text.  
        jtfPlaintext.setText(decStr.toString());  
    }  
  
    // Só sobrou o comando "Reset".  
    else {  
        jtfPlaintext.setText("");  
    }  
}
```

```

        jtfCiphertext.setText("");
    }

}

public static void main(String[] args) {

    // Cria o quadro na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Coder();
        }
    });
}
}

```

USE CLASSES INTERNAS ANÔNIMAS PARA TRATAR EVENTOS

Até agora, os programas deste capítulo usaram uma abordagem simples e direta de tratamento de eventos, em que a classe principal do aplicativo implementa ela própria a interface de ouvinte e todos os eventos são enviados para uma instância dessa classe. Embora isso seja perfeitamente aceitável, não é a única maneira de tratar eventos. Duas outras abordagens são de uso comum. Na primeira, podemos implementar classes de ouvinte separadas. Assim, classes diferentes tratariam eventos distintos e ficariam separadas da classe principal do aplicativo. A outra abordagem seria implementar ouvintes com o uso de *classes internas anônimas*.

Você deve lembrar que, no Capítulo 6, vimos que uma classe interna é uma classe não estática declarada dentro de outra classe. As classes internas anônimas são classes internas que não têm um nome. Em vez disso, uma instância da classe é gerada “dinamicamente” quando necessário. Você já usou classes internas anônimas neste capítulo, já que elas foram empregadas na geração de uma chamada ao método `invokeLater()`.

As classes internas anônimas facilitam muito a implementação de alguns tipos de tratadores de eventos. Por exemplo, dado um **JButton** chamado **jbtn**, você poderia implementar um ouvinte de ação para ele da seguinte forma:

```

jbtn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        // Trata o evento de ação aqui.
    }
});

```

Nesse exemplo, é criada uma classe interna anônima que implementa a interface **ActionListener**. Preste atenção na sintaxe: o corpo da classe interna começa após a chave que vem depois de `new ActionListener()`. Observe também que a chamada a `addActionListener()` termina com um parêntese de fechamento e um ponto e vírgula como de praxe. As mesmas sintaxe e abordagem básicas são usadas na criação de uma classe interna anônima para qualquer tratador de eventos. É claro

que, para diferentes eventos, você especificará ouvintes distintos e implementará métodos diferentes.

Uma vantagem do uso de uma classe interna anônima é que o componente que chama os métodos da classe já é conhecido. Por exemplo, no exemplo anterior, não há necessidade de chamar `getActionCommand()` para determinar que componente gerou o evento, porque a implementação de `actionPerformed()` só será chamada por eventos gerados por `jbtn`. Veremos as classes internas anônimas em ação em exemplos subsequentes.

EXERCÍCIOS

1. A maioria dos componentes do AWT é convertida em pares nativos. Por que isso é um problema e como Swing o corrige?
2. A maioria dos componentes de Swing é escrita em código 100% Java. Verdadeiro ou falso?
3. Quais são os quatro contêineres pesados de nível superior?
4. Qual é o contêiner de nível superior mais usado para um aplicativo?
5. **JFrame** contém vários painéis. A que painel os componentes são adicionados?
6. Um ouvinte de eventos deve _____ em uma fonte para receber notificações de eventos.
7. Para receber um evento de ação, uma classe deve implementar que interface?
8. No uso de um **JButton** ou **JTextField**, que método deve ser chamado para configurar o comando de ação?
9. Cite três gerenciadores de leiaute.
10. O exemplo do cronômetro da seção Tente isto 17-1 usa dois botões, um para iniciar o cronômetro e o outro para interrompê-lo. No entanto, é possível usar apenas um botão, que se alternaria entre as atividades de iniciar e interromper o cronômetro. Uma maneira de fazer isso é redefinir o texto do botão após cada pressionamento, alternando-o entre Start e Stop. Já que, por padrão, esse texto também é o comando de ação associado ao botão, você pode usar o mesmo botão para dois fins diferentes. Sua missão é reescrever o projeto da seção Tente isto 17-1 para que implemente essa abordagem.

Para resolver esse problema, você usará outro método de **JButton: setText()**. Esse método define o texto de um botão. Ele é mostrado abaixo:

```
void setText(String msg)
```

Aqui, *msg* especifica o texto que será exibido dentro do botão. Esse método permite a definição do texto de um botão durante a execução de um programa.

11. Modifique o exemplo **SwingDemo** deste capítulo para que o **JFrame** exiba cinco mensagens, uma ao longo de cada borda e uma no meio. Use o **Border-**

Layout padrão. Você pode usar as cinco mensagens que quiser. Ajuste o tamanho da janela e adicione um espaçamento apropriado para as cinco mensagens de modo que elas fiquem bem centralizadas.

12. Modifique o exemplo **ButtonDemo** discutido neste capítulo para que a classe **ButtonDemo** não seja um **ActionListener** e, em vez disso, objetos separados sejam usados como os **ActionListeners** registrados nos botões. Os dois objetos de ouvintes devem ser instâncias da mesma classe implementadora de **ActionListener**.
13. Modifique o exemplo **ButtonDemo** discutido neste capítulo para que a classe **ButtonDemo** não seja um **ActionListener** e, em vez disso, duas classes internas anônimas sejam usadas como os **ActionListeners** registrados nos botões. Cada botão deve ter seu próprio ouvinte.
14. Crie um aplicativo Swing que exiba uma janela contendo quatro botões e um rótulo. O primeiro botão deve apresentar “Clique aqui” e os outros três “Não aqui”. Se um dos botões “Não aqui” for clicado, o rótulo exibirá a mensagem “Errado. Tente novamente”. Se o botão “Clique aqui” for clicado, o rótulo exibirá “Bom trabalho. Repita.”. Além disso, sempre que o botão “Clique aqui” for clicado, um novo botão será selecionado aleatoriamente para receber o texto “Clique aqui” e os outros três botões receberão o texto “Não aqui”. O aplicativo é encerrado quando o usuário clica na caixa Fechar da janela. Use o método **setText()** da classe **JButton** discutido no Exercício 10 e o método **Math.random()** que retorna um valor **double** aleatório entre 0 e 1.
15. Crie um programa que exiba um **JFrame** com um rótulo, duas caixas de texto e um botão. O rótulo deve ter o texto “Digite a mesma coisa nas duas caixas”. Quando o usuário clicar no botão, os conteúdos das duas caixas de texto serão comparados. Se forem diferentes, o rótulo exibirá a mensagem “Tente novamente. Digite a mesma coisa nas duas caixas”. Se forem iguais, um novo **JFrame** aparecerá com a mensagem “Correto!”. A caixa Fechar do novo **JFrame** deve fechar apenas a nova janela e não o programa.
16. Verdadeiro ou falso:
 - A. Se dois ou mais ouvintes de ação se registrarem no mesmo botão, só um será notificado de um clique no botão.
 - B. Um **JTextField** gera um **ActionEvent** sempre que o usuário clica nele.
 - C. Um **JLabel** gera um **ActionEvent** sempre que o usuário clica nele.
17. Suponhamos que um ouvinte de ação se registrasse em três ou quatro controles de GUI, incluindo pelo menos um botão e um campo de texto. Quando seu método **actionPerformed()** for chamado, como o ouvinte saberá qual dos controles gerou o **ActionEvent**?
18. Quais são as principais vantagens de usar uma classe interna anônima como ouvinte de ação de um ou mais controles?

19. Implemente uma classe com um método **main()** que faça o seguinte:
- Crie uma classe interna anônima que implemente a interface **Nameable** abaixo.
 - Crie uma instância **obj** da classe interna anônima.
 - Chame **obj.setName()**.
 - Chame **obj.getName()** e exiba o resultado.

```
public interface Nameable {  
    // define o nome deste objeto  
    void setName(String name);  
  
    // retorna o nome deste objeto  
    String getName();
```

Examinando os controles de Swing

PRINCIPAIS HABILIDADES E CONCEITOS

- JLabel
- ImageIcon
- JButton
- JToggleButton
- JCheckBox
- JRadioButton
- JTextField
- JScrollPane
- JList
- JComboBox
- JTree
- JTable

A GUI moderna é preenchida com um extenso grupo de controles. Aqui, o termo *controles* se refere aos componentes que interagem diretamente com o usuário, como botões, campos de texto e listas suspensas. Para dar suporte à GUI moderna, o Swing fornece um rico conjunto de controles. Usando-os, você pode construir aplicativos com a aparência contemporânea que conhece. Embora uma descrição detalhada de cada controle não faça parte do escopo do livro, este capítulo descreverá um resumo representativo. Nossa objetivo é introduzir vários controles populares e descrever as técnicas básicas que seu uso requer. Uma vez que você entender como esses componentes funcionam, poderá usar a documentação de APIs Java para conhecer melhor seus recursos e outros controles do Swing.

As classes de componentes do Swing descritas neste capítulo são mostradas abaixo:

JButton	JCheckBox	JComboBox
JLabel	JList	JRadioButton
JScrollPane	JTable	JTextField
JToggleButton	JTree	

Todos esses componentes são leves, ou seja, são todos derivados de **JComponent**.

Também discutiremos as classes **ButtonGroup**, que encapsula um conjunto de botões mutuamente exclusivo, e **ImageIcon**, que encapsula uma imagem gráfica. As duas são definidas pelo Swing e ficam no pacote **javax.swing**.

JLabel E ImageIcon

JLabel é o componente mais fácil de usar do Swing. Ele cria um rótulo e foi introduzido no capítulo anterior. Aqui, examinaremos **JLabel** com um pouco mais de detalhes. Como você sabe, **JLabel** é um componente passivo, ou seja, não responde a entradas do usuário. No capítulo anterior, foi usado para exibir uma mensagem de texto. No entanto, também pode ser usado para exibir um ícone (uma imagem gráfica) ou tanto um ícone quanto um texto. Você também pode especificar o alinhamento do conteúdo do rótulo. Por exemplo, abaixo temos mais três construtores de **JLabel**:

```
JLabel(String str, int alin)  
JLabel(Icon icon, int alin)  
JLabel(String str, Icon icon, int alin)
```

Aqui, *str* especifica o texto do rótulo e *icon* especifica a imagem gráfica. O alinhamento horizontal dentro das dimensões do rótulo é especificado por *alin*. Ele deve ter um dos valores a seguir: **LEFT**, **RIGHT**, **CENTER**, **LEADING** ou **TRAILING**. (**LEADING** e **TRAILING** especificam a borda anterior ou posterior, o que é útil na internacionalização de um rótulo para idiomas lidos da esquerda para a direita e da direita para a esquerda.) Essas constantes estão definidas na interface **SwingConstants**, junto com várias outras usadas pelas classes do Swing.

Observe que os ícones são especificados por objetos de tipo **Icon**, que é uma interface definida pelo Swing. A maneira mais fácil de obter um ícone é usar a classe **ImageIcon**. **ImageIcon** implementa **Icon** e encapsula uma imagem. Logo, um objeto de tipo **ImageIcon** pode ser passado como argumento para o parâmetro **Icon** do construtor de **JLabel**. Há várias maneiras de fornecer a imagem, inclusive lendo-a em um arquivo ou baixando-a de um URL. O construtor de **ImageIcon** usado pelo exemplo desta seção é:

```
ImageIcon(String nomearquivo)
```

Ele obtém a imagem no arquivo passado para *nomearquivo*. Normalmente arquivos que contêm imagens usam extensões como **gif**, **jpeg** ou **png**. Essas extensões especificam diferentes formatos de imagem, mas todos podem ser carregados por **ImageIcon**.

O programa a seguir ilustra como criar e exibir rótulos contendo tanto um ícone quanto um string e usar o alinhamento especificado. O programa cria três rótulos. Cada rótulo indica um estado de um semáforo: avançar, alerta e parar. Eles são instâncias de **ImageIcon** e são criados pela carga dos arquivos **Go.gif**, **Caution.gif** e **Stop.gif**, que contêm as imagens. O texto indica o estado do semáforo. Neste exemplo, os conteúdos de cada rótulo são alinhados diferentemente, indo da esquerda para a direita, mas, se quiser, teste outros alinhamentos. Outra coisa: observe que o **BorderLayout** padrão do painel de conteúdo é usado. Lembre-se, esse **BorderLayout** define

cinco locais: centro, norte, sul, leste e oeste. No exemplo, o primeiro rótulo entra no local norte, o segundo usa o local central e o terceiro usa o local sul. **BorderLayout** torna mais fácil mostrar o alinhamento dos rótulos.

```
// Demonstra JLabel e ImageIcon.
// Este exemplo exibe os três estados de um semáforo.

import javax.swing.*;
import java.awt.*;

class JLabelDemo {

    JLabelDemo() {

        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("JLabel and ImageIcon Example");

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(320, 280);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria um ícone e um rótulo para Go.
        ImageIcon goIcon = new ImageIcon("Go.gif"); ← Cria um ícone a partir de
        JLabel jlabGo = new JLabel(" Go ", goIcon, SwingConstants.LEFT); ← uma imagem armazenada
                                                                em um arquivo.

        // Cria um ícone e um rótulo para Caution.
        ImageIcon cautionIcon = new ImageIcon("Caution.gif");
        JLabel jlabCaution = new JLabel(" Caution ", cautionIcon,
                                       SwingConstants.CENTER); ← Cria um rótulo contendo
                                                                um ícone e texto.

        // Cria um ícone e um rótulo para Stop.
        ImageIcon stopIcon = new ImageIcon("Stop.gif");
        JLabel jlabStop = new JLabel(" Stop ", stopIcon,
                                   SwingConstants.RIGHT);

        // Adiciona os rótulos ao painel de conteúdo.
        jfrm.add(jlabGo, BorderLayout.NORTH);
        jfrm.add(jlabCaution, BorderLayout.CENTER);
        jfrm.add(jlabStop, BorderLayout.SOUTH);

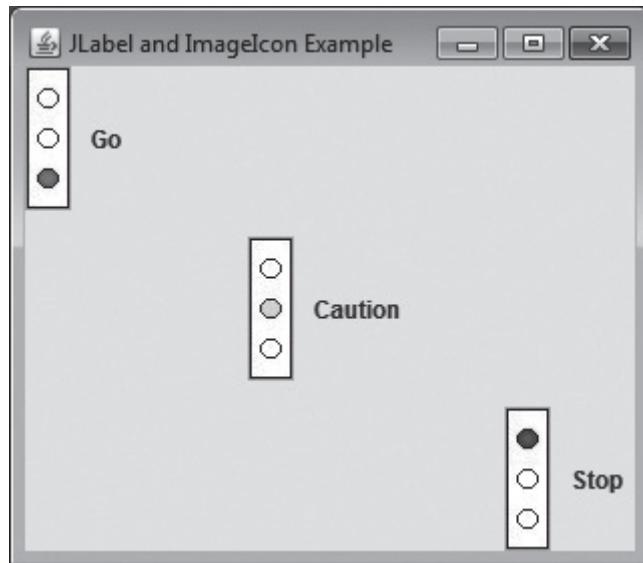
        // Exibe o quadro.
        jfrm.setVisible(true);
    }

    public static void main(String[] args) {
        // Cria a GUI na thread de despacho de evento.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {

```

```
        new JLabelDemo();  
    }  
});  
}  
}
```

A saída do exemplo de rótulo é mostrada abaixo:



Após um rótulo ter sido criado, você pode alterar o texto ou o ícone que será exibido usando os métodos `setText()` ou `setIcon()`. O método `setText()` foi descrito no Capítulo 17. O método `setIcon()` é mostrado aqui:

```
void setIcon(Icon novoIcone)
```

Aqui, *novoÍcone* passa a ser a nova imagem exibida dentro do rótulo.

Apesar de sua relativa simplicidade, **JLabel** dá suporte a muitos outros recursos que você deve explorar. Por exemplo, você pode definir um mnemônico de teclado, definir o alinhamento vertical e ver o texto e/ou o ícone do rótulo. **JLabel** tem mais recursos do que se supõe à primeira vista.

Pergunte ao especialista

P Como posso criar um rótulo que exiba várias linhas de texto?

R Para exibir várias linhas em um rótulo, você pode usar HTML como texto. Para fazer isso, deve começar o texto com a sequência <html>. Quando o fizer, o texto será formatado automaticamente como descrito pela marcação. Além de outros benefícios, o uso

de HTML permite a criação de rótulos que se estendam por duas ou mais linhas. Por exemplo, esse código cria um rótulo que exibe duas linhas de texto, com o string “Top” acima do string “Bottom”.

```
JLabel jlabhtml = new JLabel("<html>Top<br>Bottom");
```

Na verdade, a HTML pode ser usada em outros componentes além dos rótulos. Por exemplo, você pode usá-la em um botão. Em geral, quando um componente exibe texto, esse texto pode ser HTML.

Verificação do progresso

1. Um **JLabel** pode incluir uma imagem gráfica?
2. O conteúdo de um **JLabel** deve ser sempre centralizado. Verdadeiro ou falso?
3. Que classe é normalmente usada para encapsular um ícone?

OS BOTÕES DE SWING

O Capítulo 17 introduziu o botão descrevendo os aspectos básicos da classe **JButton**. No entanto, **JButton** é apenas um dos vários tipos de botões do Swing. Aqui está uma lista das classes de botões que o Swing fornece:

JButton	Um botão de ação padrão
JToggleButton	Um botão com dois estados (ativo/inativo)
JCheckBox	Um botão de seleção padrão
JRadioButton	Um botão de seleção mutuamente exclusivo

Todos os botões são subclasses da classe **AbstractButton**, que estende **JComponent**. **AbstractButton** contém muitos métodos que fornecem a funcionalidade básica comum a todos os botões.

Há três categorias de eventos de botão: eventos de ação, eventos de item e eventos de mudança. Um evento de ação é gerado quando o usuário executa uma ação, como clicar em um botão. Um evento de item é gerado quando um botão é marcado ou desmarcado. Um evento de mudança é acionado quando o estado do botão muda. Lembre-se, os três eventos não são obrigatoriamente significativos para todos os botões ou todas as aplicações dos botões. Por exemplo, **JButton** não gera eventos de item. Neste capítulo, só eventos de ação e de item serão usados. Eles serão descritos agora.

Respostas:

1. Sim.
2. Falso.
3. **ImageIcon**

Tratando eventos de ação

Um evento de ação é gerado quando o usuário clica em um botão. Ele foi introduzido no Capítulo 17. Recapitulemos: um evento de ação é representado pela classe **ActionEvent** e tratado por classes que implementam a interface **ActionListener**. Essa interface especifica apenas um método, **actionPerformed()**, que é mostrado aqui:

```
void actionPerformed(ActionEvent ae)
```

A ação é recebida em *ae*.

O objeto **ActionEvent** passado para **actionPerformed()** dá acesso a várias informações. Talvez o mais importante seja que ele permite identificar que componente gerou o evento. Há duas maneiras de identificar o componente: pelo string do comando de ação (como mostrado no Capítulo 17) ou pela referência de objeto. Podemos obter o string do comando de ação do componente que gerou o evento chamando **getActionCommand()** no objeto **ActionEvent**. O comando de ação identifica o botão. Também podemos definir o comando de ação chamando **setActionCommand()**. Lembre-se de que esses métodos são declarados como mostrado abaixo:

```
String getActionCommand()  
void setActionCommand(String cmd)
```

O string passado em *cmd* passa a ser o novo comando de ação. O texto do botão não é afetado.

A segunda maneira de identificar o componente que gerou um evento de ação é obtendo uma referência a ele chamando **getSource()** no objeto **ActionEvent**. Esse método é definido por **EventObject**, a superclasse de todos os objetos de evento. Ele é mostrado aqui:

```
Object getSource()
```

Uma vantagem de se usar **getSource()** é que, se você tiver que agir diretamente sobre o componente, pode fazer isso por intermédio de sua referência. Não usaremos essa abordagem nos exemplos deste capítulo, mas é uma opção que pode ser útil em seu código.

Tratando eventos de item

Um evento de item ocorre quando um item, como um botão de alternância, caixa de seleção ou botão de rádio, é selecionado. Os eventos de item são representados pela classe **ItemEvent** e tratados por classes que implementam a interface **ItemListener**. Essa interface especifica apenas um método, **itemStateChanged()**, que é mostrado a seguir:

```
void itemStateChanged(ItemEvent ie)
```

O evento de item é recebido em *ie*.

Para obter uma referência ao item que mudou, chame **getItem()** no objeto **ItemEvent**. Esse método é mostrado abaixo:

```
Object getItem()
```

A referência retornada deve ser convertida para a classe de componente que está sendo tratada, como **JCheckBox** ou **JRadioButton**.

Quando um evento de item ocorre, o componente fica em um entre dois estados: marcado ou desmarcado. A classe **ItemEvent** define as constantes **static final int** a seguir que representam esses dois estados:

SELECTED	DESELECTED
----------	------------

Para obter o novo estado, chame o método **getStateChange()** definido por **ItemEvent**. Ele é mostrado aqui:

```
int getStateChange()
```

O método retorna **ItemEvent.SELECTED** ou **ItemEvent.DESELECTED**. Você também pode determinar o estado marcado/desmarcado de um botão chamando **isSelected()**, que é definido por **AbstractButton**. Sua forma é:

```
boolean isSelected()
```

Ele retorna **true** se o botão for selecionado; caso contrário, retorna **false**.

JButton

A classe **JButton** fornece a funcionalidade de um botão de ação. Você já viu uma forma simples dela no capítulo anterior. Aqui usaremos alguns de seus recursos adicionais.

JButton permite que um ícone, um string ou ambos sejam associados ao botão de ação. Três de seus construtores são mostrados abaixo:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

Aqui, *str* e *icon* são o string e o ícone usados no botão.

Como explicado, quando o botão é pressionado, um **ActionEvent** é gerado. Usando o objeto **ActionEvent** passado para o método **actionPerformed()** do **ActionListener** registrado, você pode obter o string de comando de ação associado ao botão. Por padrão, ele é o string exibido dentro dele. No entanto, você pode configurar o comando de ação chamando **setActionCommand()** no botão. E pode ver o comando de ação chamando **getActionCommand()** no objeto de evento. Assim, quando são usados dois ou mais botões dentro da mesma aplicação, o comando de ação fornece uma maneira fácil de determinar que botão foi pressionado.

No capítulo anterior, você viu o exemplo de um botão baseado em texto. O exemplo a seguir demonstra um botão baseado em ícones. Ele exibe três botões de ação e um rótulo. Cada botão exibe um estado do semáforo, usando as mesmas imagens usadas pelo exemplo anterior. Quando um botão é pressionado, o estado do semáforo é exibido no rótulo. Observe o uso de **setActionCommand()** para definir os comandos de ação dos botões baseados em ícones.

```
// Demonstra JButtons baseados em ícones.  
// Este exemplo exibe ícones de semáforo dentro de botões.  
  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class JButtonDemo implements ActionListener {  
    JLabel jlab;  
  
    JButtonDemo() {  
  
        // Cria um contêiner JFrame.  
        JFrame jfrm = new JFrame("JButton Example");  
  
        // Especifica FlowLayout como gerenciador de leiaute.  
        jfrm.setLayout(new FlowLayout());  
  
        // Fornece um tamanho inicial para o quadro.  
        jfrm.setSize(300, 180);  
  
        // Encerra o programa quando o usuário fecha o aplicativo.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Cria os botões.  
        ImageIcon goIcon = new ImageIcon("Go.gif"); ——— Cria um botão baseado  
        JButton jbtnGo = new JButton(goIcon); ——— em ícones.  
        jbtnGo.setActionCommand("Go");  
        jbtnGo.addActionListener(this);  
  
        ImageIcon cautionIcon = new ImageIcon("Caution.gif");  
        JButton jbtnCaution = new JButton(cautionIcon);  
        jbtnCaution.setActionCommand("Caution");  
        jbtnCaution.addActionListener(this);  
  
        ImageIcon stopIcon = new ImageIcon("Stop.gif");  
        JButton jbtnStop = new JButton(stopIcon);  
        jbtnStop.setActionCommand("Stop");  
        jbtnStop.addActionListener(this);  
  
        // Adiciona os botões ao painel de conteúdo.  
        jfrm.add(jbtnGo);  
        jfrm.add(jbtnCaution);  
        jfrm.add(jbtnStop);  
  
        // Cria e adiciona o rótulo ao painel de conteúdo.  
        jlab = new JLabel("Select a Traffic Light");  
        jfrm.add(jlab);  
  
        // Exibe o quadro.  
        jfrm.setVisible(true);
```

```

}

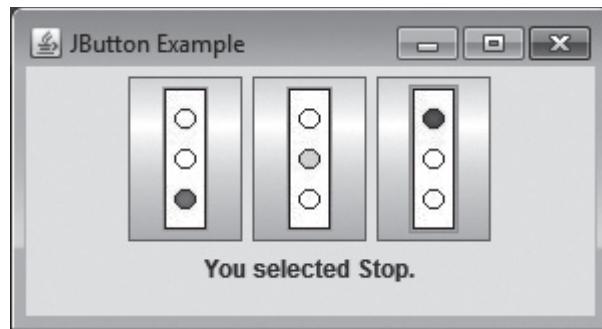
// Trata eventos de botão.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand() + ".");
}

public static void main(String[] args) {

    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JButtonDemo();
        }
    });
}
}

```

A saída do exemplo de botão é mostrada aqui:



Às vezes é útil desativar um botão quando ele não pode ser usado. Para fazer isso, chame `setEnabled()`, que é herdado de `JComponent`. Sua forma geral é mostrada abaixo:

```
void setEnabled(boolean estado)
```

Se o *estado* for **true**, o botão estará ativo. Se for **false**, ele estará inativo. Um botão inativo tem seu texto e/ou ícone exibido em cinza. Se quiser, tente desativar um botão do programa anterior e observe os resultados.

Além do ícone padrão especificado pelo construtor de `JButton`, é possível definir ícones para serem exibidos na ocorrência de certas ações de botão. Especificamente, você pode definir o ícone que será exibido quando o botão estiver inativo, quando for pressionado e quando o mouse for rolado por cima dele. Para definir esses ícones, você usará os métodos a seguir:

```
void setDisableIcon(Icon íconeDesativado)
void setPressedIcon(Icon íconePressionado)
void setRolloverIcon(Icon íconeComrolagemdomouse)
```

Esses métodos são herdados por **JButton** de **AbstractButton**. Uma vez que o ícone for definido, ele será exibido sempre que uma das ações ocorrer. Se quiser, teste-os por conta própria. Sua inclusão pode gerar uma GUI muito dinâmica.

Pergunte ao especialista

P Você acabou de mencionar que um **JButton** pode ser desativado pelo uso de **setEnabled()**. Outros controles podem ser desativados?

R Sim. Como **setEnabled()** é especificado por **JComponent**, é aplicável a outros tipos de controles leves. A possibilidade de ativar/desativar um componente é parte importante do design de GUIs no mundo real.

JToggleButton

Uma variação útil do botão de ação se chama *botão de alternância*. Um botão de alternância tem a mesma aparência de um botão de ação, mas age diferentemente porque tem dois estados: pressionado e solto. Isto é, quando pressionamos um botão de alternância, ele se mantém pressionado em vez de voltar para cima como um botão de ação comum. Quando pressionamos o botão de alternância uma segunda vez, ele é solto (volta a subir). Portanto, sempre que um botão de alternância é pressionado, ele se alterna entre seus dois estados.

Os botões de alternância são objetos da classe **JToggleButton**. Além de criar botões de alternância padrão, **JToggleButton** é superclasse de outros dois componentes do Swing que também representam controles de dois estados. Eles são **JCheckBox** e **JRadioButton**, que serão descritos posteriormente neste capítulo. Logo, **JToggleButton** define a funcionalidade básica de todos os componentes de dois estados.

JToggleButton define vários construtores. O usado pelo exemplo desta seção é mostrado aqui:

```
JToggleButton(String str)
```

Esse construtor cria um botão de alternância que contém o texto passado em *str*. Por padrão, o botão fica na posição ‘desativado’. Outros construtores nos permitem criar um botão de alternância contendo imagens (ou imagens e texto) ou definir seu estado.

Como **JButton**, **JToggleButton** gera um evento de ação sempre que é pressionado. No entanto, diferentemente de **JButton**, ele também gera um evento de item. Esse evento é usado pelos componentes que dão suporte ao conceito de seleção. Quando um **JToggleButton** é pressionado, é feita uma seleção. Quando é solto, a seleção é anulada.

Para tratar eventos de item, você deve implementar a interface **ItemListener**. Sempre que um evento de item é gerado, ele é passado para o método **itemStateChanged()** definido por **ItemListener**. Dentro de **itemStateChanged()**, o método **getItem()** pode ser chamado no objeto **ItemEvent** para obter uma referência à instância de **JToggleButton** que gerou o evento.

Provavelmente a maneira mais fácil de determinar o estado de um botão de alternância seja chamando o método **isSelected()** no botão que gerou o evento, como descrito anteriormente. Lembre-se, ele retorna **true** se o botão estiver pressionado; caso contrário, retorna **false**.

Aqui está um exemplo que usa um botão de alternância. Observe como o ouvinte de itens funciona. Ele apenas chama **isSelected()** para determinar o estado do botão.

```
// Demonstra JToggleButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JToggleButtonDemo {

    JLabel jlab;
    JToggleButton jtbn;

    JToggleButtonDemo() {

        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("JToggleButton Example");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(200, 100);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria um rótulo.
        jlab = new JLabel("Button is off.");

        // Cria um botão de alternância.
        jtbn = new JToggleButton("On/Off"); ← Cria um botão de alternância.

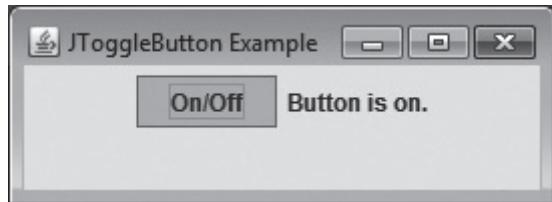
        // Adiciona um ouvinte de itens para o botão de alternância.
        jtbn.addItemListener(new ItemListener() { ← Usa um ItemListener
            public void itemStateChanged(ItemEvent ie) {
                if(jtbn.isSelected()) ← Usa isSelected()
                    jlab.setText("Button is on.");           para determinar o
                else                                estado do botão.
                    jlab.setText("Button is off.");
            }
        });

        // Adiciona o botão de alternância e o rótulo ao painel de conteúdo.
        jfrm.add(jtbn);
        jfrm.add(jlab);

        // Exibe o quadro.
        jfrm.setVisible(true);
    }
}
```

```
public static void main(String[] args) {  
  
    // Cria a GUI na thread de despacho de evento.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new JToggleButtonDemo();  
        }  
    });  
}
```

A saída do exemplo de botão de alternância é mostrada abaixo:



Caixas de seleção

A classe **JCheckBox** fornece a funcionalidade de uma caixa de seleção. Sua superclasse imediata é **JToggleButton**, que dá suporte a botões de dois estados, como acabamos de descrever. **JCheckBox** define vários construtores. O usado aqui é

`JCheckBox(String str)`

Ele cria uma caixa de seleção contendo o texto especificado por *str* como rótulo. Outros construtores permitem a especificação do estado de seleção inicial do botão e a especificação de um ícone.

Quando o usuário marca ou desmarca uma caixa de seleção, um **ItemEvent** é gerado. Você pode obter uma referência ao **JCheckBox** que gerou o evento chamando **getItem()** no **ItemEvent** passado para o método **itemStateChanged()** definido por **ItemListener**. A maneira mais fácil de determinar o estado de seleção da caixa é chamar **isSelected()** na instância de **JCheckBox**.

O exemplo a seguir ilustra as caixas de seleção. Ele exibe quatro caixas de seleção e um rótulo. Quando o usuário clica em uma caixa de seleção, um **ItemEvent** é gerado. O método **getItem()** é chamado dentro de **itemStateChanged()** para obter uma referência ao objeto **JCheckBox** que gerou o evento. Em seguida, uma chamada a **isSelected()** determina se a caixa foi marcada ou desmarcada. O método **getText()** obtém o texto da caixa de seleção e o usa para definir o texto do rótulo.

```
// Demonstra JCheckBox.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class JCheckBoxDemo implements ItemListener {
```

```

JLabel jlabChange;
JLabel jlabSupported;

JCheckBox cbWin;
JCheckBox cbLinux;
JCheckBox cbMac;

JCheckBoxDemo() {
    // Cria um contêiner JFrame.
    JFrame jfrm = new JFrame("JCheckBox Example");

    // Especifica FlowLayout como gerenciador de leiaute.
    jfrm.setLayout(new FlowLayout());

    // Fornece um tamanho inicial para o quadro.
    jfrm.setSize(340, 140);

    // Encerra o programa quando o usuário fecha o aplicativo.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Adiciona caixas de seleção ao painel de conteúdo.
    cbWin = new JCheckBox("Windows"); ←
    cbWin.addItemListener(this);
    jfrm.add(cbWin);

    cbLinux = new JCheckBox("Linux"); ← Cria as caixas de seleção.
    cbLinux.addItemListener(this);
    jfrm.add(cbLinux);

    cbMac = new JCheckBox("Mac OS"); ←
    cbMac.addItemListener(this);
    jfrm.add(cbMac);

    // Cria rótulos.
    jlabChange = new JLabel("Select Supported Operating Systems");
    jfrm.add(jlabChange);

    jlabSupported = new JLabel();
    jfrm.add(jlabSupported);

    // Exibe o quadro.
    jfrm.setVisible(true);
}

// Trata eventos de item das caixas de seleção.
public void itemStateChanged(ItemEvent ie) { ← Escuta eventos de item gerados
    JCheckBox cb = (JCheckBox)ie.getItem(); pelas caixas de seleção.

    if(cb.isSelected()) ← Determina se a caixa
        jlabChange.setText(cb.getText() + " has been selected");
        de seleção foi marcada
        ou desmarcada.
}

```

```
else
    jlabChange.setText(cb.getText() + " has been cleared");

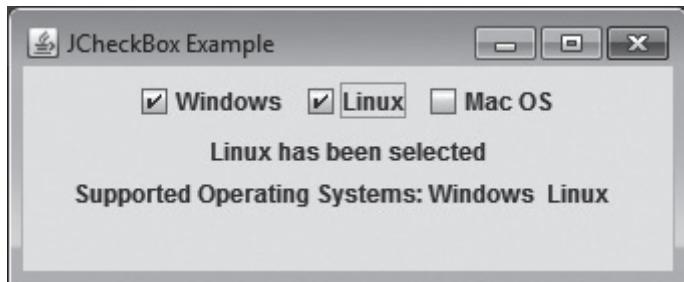
// Constrói um string que indica todas as seleções.
String supported = "Supported Operating Systems: ";
if(cbWin.isSelected()) supported += "Windows ";
if(cbLinux.isSelected()) supported += "Linux ";
if(cbMac.isSelected()) supported += "Mac OS";

jlabSupported.setText(supported);
}

public static void main(String[] args) {

    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JCheckBoxDemo();
        }
    });
}
```

A saída desse exemplo é mostrada aqui:



Botões de rádio

Os botões de rádio são um grupo de botões mutuamente exclusivos, em que somente um botão de cada vez pode ser pressionado. Eles são suportados pela classe **JRadioButton**, que estende **JToggleButton**. **JRadioButton** fornece vários construtores. O usado no exemplo é mostrado abaixo:

```
JRadioButton(String str)
```

Aqui, *str* é o rótulo do botão. Outros construtores permitem a especificação do estado de seleção inicial do botão e a especificação de um ícone.

Para sua natureza mutuamente exclusiva ser ativada, os botões de rádio devem ser estruturados em um grupo. Só um dos botões do grupo pode ser pressionado em um determinado momento. Por exemplo, se um usuário pressionar um botão de rádio que faça parte de um grupo, qualquer botão pressionado anteriormente nesse grupo

perderá automaticamente o estado ‘pressionado’. O grupo de botões é criado pela classe **ButtonGroup**. Seu construtor padrão é chamado para esse fim. Os elementos são então adicionados ao grupo de botões pelo método a seguir:

```
void add(AbstractButton botão)
```

Aqui, *botão* é uma referência ao botão a ser adicionado ao grupo.

Um **JRadioButton** gera eventos de ação, eventos de item e eventos de mudança sempre que a seleção do botão muda. Geralmente, é o evento de ação que é tratado, ou seja, você terá que implementar a interface **ActionListener**. Lembre-se, o único método definido por **ActionListener** é **actionPerformed()**. Dentro desse método, você pode usar várias técnicas para determinar que botão foi pressionado. Vejamos três. Em primeiro lugar, você pode verificar o comando de ação chamando **getActionCommand()**. Por padrão, o comando de ação é o mesmo do rótulo do botão, mas você pode redefini-lo chamando **setActionCommand()** no botão de rádio. Em segundo lugar, você pode chamar **getSource()** no objeto **ActionEvent** e verificar essa referência em relação aos botões. Para concluir, você pode simplesmente verificar cada botão de rádio para descobrir qual está pressionado chamando **isSelected()** em cada botão. Não se esqueça, sempre que um evento de ação ocorre, isso significa que o botão que está sendo pressionado mudou e que apenas um botão será pressionado.

O exemplo a seguir ilustra como usar botões de rádio. Três botões de rádio são criados. Em seguida, eles são adicionados a um grupo de botões. Como explicado, isso é necessário para acionar seu comportamento mutuamente exclusivo. O pressionamento de um botão de rádio gera um evento de ação, que é tratado por **actionPerformed()**. Dentro desse tratador, o método **getActionCommand()** obtém o texto associado ao botão de rádio e o usa para definir o texto de um rótulo.

```
// Demonstra JRadioButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo implements ActionListener {
    JLabel jlab;

    JRadioButtonDemo() {
        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("JRadioButton Example");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(350, 100);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void actionPerformed(ActionEvent e) {
        // Obtém o comando associado ao botão que foi pressionado.
        String cmd = ((JRadioButton)e.getSource()).getActionCommand();

        // Define o rótulo com o comando.
        jlab.setText(cmd);
    }
}
```

```

// Cria botões de rádio e os adiciona ao painel de conteúdo.
JRadioButton b1 = new JRadioButton("Debug");
b1.addActionListener(this);
jfrm.add(b1);

JRadioButton b2 = new JRadioButton("Maximize Speed");
b2.addActionListener(this);
jfrm.add(b2);

JRadioButton b3 = new JRadioButton("Minimize Size");
b3.addActionListener(this);
jfrm.add(b3);

// Define um grupo de botões.
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2); └─ Adiciona botões de rádio a um grupo de botões.
bg.add(b3);

// Cria um rótulo e o adiciona ao painel de conteúdo.
jlab = new JLabel("Select One");
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

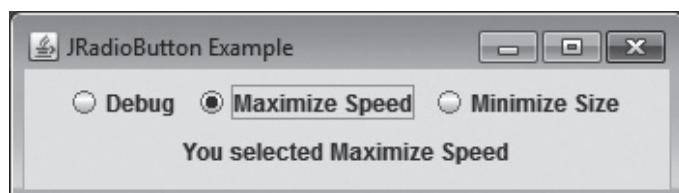
// Trata a seleção do botão.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
}

public static void main(String[] args) {

    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JRadioButtonDemo();
        }
    });
}
}

```

A saída do exemplo de botão de rádio é mostrada aqui:



Verificação do progresso

1. Que tipo de evento costuma ser tratado quando usamos **JCheckBox**?
2. **JToggleButton** cria um botão de _____ estados.
3. Para os botões de rádio funcionarem corretamente, eles devem ser adicionados a que classe?

JTextField

JTextField foi introduzido no Capítulo 17. Recapitulando, **JTextField** permite ao usuário a inserção de uma única linha de texto. Ele gera um **ActionEvent** quando o usuário pressiona ENTER no campo. Para tratar eventos de ação, você deve implementar a interface **actionListener**. Um **JTextField** tem um string de comando de ação associada a ele. Por padrão, o comando de ação é o conteúdo atual do campo de texto. No entanto, você pode configurá-lo com um comando de ação de sua escolha chamando o método **setActionCommand()**. Para obter o string que está sendo exibido atualmente no campo de texto, chame o método **getText()** na instância de **JTextField**. Ele é mostrado aqui:

```
String getText()
```

Você pode definir o texto de um **JTextField** chamando o método **setText()**, mostrado a seguir:

```
void setText(String texto)
```

Aqui, *texto* é o string que será inserido no campo de texto. Além desses recursos que você já conhece, **JTextField** dá suporte a vários outros que podem ser úteis. Discutiremos um resumo agora.

Você pode criar um **JTextField** contendo um string inicial usando um desses construtores:

```
JTextField(String str)
JTextField(String str, int cols)
```

No primeiro caso, o campo de texto é dimensionado de acordo com o string. Na segunda forma, o número de colunas é especificado.

Você pode obter a parte do texto que foi selecionada chamando o método **getSelectedText()**, mostrado aqui:

```
String getSelectedText()
```

Se não houver um texto selecionado, **null** será retornado.

Você pode mover texto para ou da área de transferência sob controle do programa usando os métodos **cut()**, **copy()** e **paste()**, mostrados abaixo:

```
void cut()
void copy()
void paste()
```

Respostas:

1. **ItemEvent**
2. dois
3. **ButtonGroup**

O método **cut()** remove qualquer texto que tenha sido selecionado dentro do campo de texto e o copia na área de transferência. O método **copy()** apenas copia, sem remover, o texto selecionado. O método **paste()** copia qualquer texto que possa estar na área de transferência para o campo de texto. Se o campo tiver texto selecionado, ele será substituído pelo conteúdo da área de transferência. Caso contrário, o texto da área de transferência será inserido imediatamente antes da posição atual do cursor.

O programa a seguir demonstra **JTextField** e vários dos métodos que acabamos de discutir. Ele cria um campo de texto de 15 colunas de largura. Sempre que você pressionar ENTER enquanto estiver no campo de texto, o conteúdo atual será exibido. Se alguma parte do campo tiver sido selecionada, o texto selecionado também será exibido. O programa tem dois botões chamados Cut e Paste. Esses botões mostram como as funções padrão de recortar e colar podem ser utilizadas sob controle do programa. Se houver um texto selecionado, o pressionamento de Cut o removerá e inserirá na área de transferência. O pressionamento de Paste copia o texto da área de transferência para o campo de texto. É claro que você também pode executar as funções de recortar e colar com os comandos padrão do teclado, como CTRL-X e CTRL-V no ambiente Windows. Um exemplo da saída é mostrado aqui:



```
// Demonstra vários recursos de um campo de texto.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class JTextFieldDemo {

    JLabel jlabAll;
    JLabel jlabSelected;

    JTextField jtf;

    JButton jbtnCut;
    JButton jbtnPaste;

    public JTextFieldDemo() {

        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("JTextField Example");
    }
}
```

```

// Especifica FlowLayout como gerenciador de leiaute.
jfrm.setLayout(new FlowLayout());

// Fornece um tamanho inicial para o quadro.
jfrm.setSize(200, 150);

// Encerra o programa quando o usuário fecha o aplicativo.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Cria rótulos.
jlabAll = new JLabel("All text: ");
jlabSelected = new JLabel("Selected text: ");

// Cria o campo de texto.
jtf = new JTextField("This is a test.", 15); ← Cria um campo de texto.

// Adiciona um ouvinte de ação para o campo de texto.
// Sempre que o usuário pressionar enter, o conteúdo
// do campo será exibido. Qualquer texto selecionado
// atualmente também será exibido.
jtf.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlabAll.setText("All text: " + jtf.getText()); ← Sempre que o usuário
        if(jtf.getSelectedText() != null)           pressionar ENTER,
            jlabSelected.setText("Selected text: " + ← exibirá o texto inteiro
                                jtf.getSelectedText()); e qualquer texto
        }
    });
}

// Cria os botões Cut e Paste.
jbtnCut = new JButton("Cut");
jbtnPaste = new JButton("Paste");

// Adiciona um ouvinte de ação para o botão Cut.
jbtnCut.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        // Recorta o texto selecionado e o insere
        // na área de transferência.
        jtf.cut(); ← Remove o texto
        jlabAll.setText("All text: " + jtf.getText()); selecionado e o
        if(jtf.getSelectedText() != null)           insere na área de
            jlabSelected.setText("Selected text: " + transferência.
                                jtf.getSelectedText());
    }
});

// Adiciona um ouvinte de ação para o botão Paste.
jbtnPaste.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        // Cola o texto da área de transferência
    }
});

```

```

        // no campo de texto.
        jtf.paste(); ← Copia texto da área de transferência.
    }
});

// Adiciona os componentes ao painel de conteúdo.
jfrm.add(jtf);
jfrm.add(jbtnCut);
jfrm.add(jbtnPaste);
jfrm.add(jlabAll);
jfrm.add(jlabSelected);

// Exibe o quadro.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JTextFieldDemo();
        }
    });
}
}

```

Grande parte desse programa é conhecida, mas há alguns pontos-chave que merecem um exame mais detalhado. Primeiro, quando o campo de texto **jtf** é criado, ele recebe tanto um tamanho inicial igual a 15 quanto um conteúdo inicial composto pelo string “This is a test”.

Sempre que o usuário pressionar ENTER quando estiver dentro do campo de texto **jtf**, um **ActionEvent** será gerado e enviado para o método **actionPerformed()** do ouvinte de ações de **jtf**. Esse método obtém o texto contido atualmente no campo chamando **getText()** em **jtf**. Em seguida, ele exibe o texto por intermédio do rótulo referenciado por **jlabAll**. Ele também obtém qualquer texto que tiver sido selecionado chamando **getSelectedText()** e o exibe no rótulo **jlabSelected**.

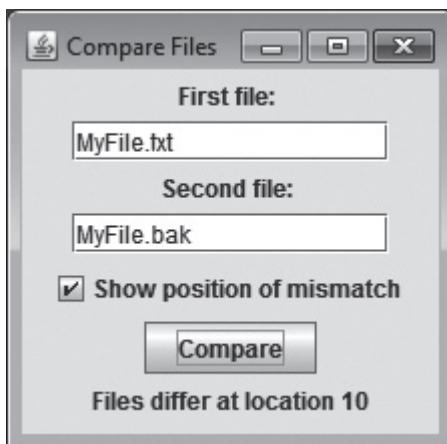
Agora examinemos os tratadores de eventos dos botões Cut e Paste. O botão Cut chama **cut()** para remover um texto selecionado e inseri-lo na área de transferência. O botão Paste copia texto da área de transferência para o campo de texto chamando **paste()**.

TENTE ISTO 18-1 Um utilitário de comparação de arquivos baseado no Swing

`SwingFC.java`

Embora apenas um pequeno número de controles tenha sido introduzido, mesmo assim você pode usá-los para criar um aplicativo prático. Na seção Tente isto 11-1, você criou um utilitário de comparação de arquivos baseado no

console. Esse projeto criará uma versão do programa baseada no Swing. Como você verá, dar a esse aplicativo uma interface de usuário baseada no Swing melhorará significativamente sua aparência e facilitará seu uso. Ela também adiciona alguma funcionalidade extra porque permite a exibição da posição da primeira discrepância quando dois arquivos diferem. Veja a aparência da versão do Swing:



Já que o Swing otimiza a criação de programas baseados em GUI, você pode ficar surpreso com a facilidade da criação desse programa.

PASSO A PASSO

1. Comece criando um arquivo chamado **SwingFC.java** e então insira o comentário e as instruções **import** a seguir:

```
/*
 * Tente isto 18-1
 *
 * Um utilitário de comparação de arquivos baseado no Swing.
 *
 * Requer o JDK 7 ou posterior.
 */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
```

2. Em seguida, crie a classe **SwingFC** como mostrado aqui:

```
class SwingFC implements ActionListener {
    JTextField jtfFirst; // contém o nome do primeiro arquivo
    JTextField jtfSecond; // contém o nome do segundo arquivo
    JButton jbtnComp; // botão para comparar os arquivos
```

```

JLabel jlabFirst, jlabSecond; // exibe avisos
JLabel jlabResult; // exibe resultados e mensagens de erro

JCheckBox jcbLoc; // marque para exibir o local da discrepância

```

Os nomes dos arquivos a serem comparados são inseridos nos campos de texto definidos por **jtfFirst** e **jtfSecond**. Para comparar os arquivos, o usuário tem que pressionar o botão **jbtnComp**. Mensagens de aviso são exibidas em **jlabFirst** e **jlabSecond**. Os resultados da comparação, ou qualquer mensagem de erro, são exibidos em **jlabResult**. A caixa de seleção **jcbLoc** permite que o usuário determine se a posição da primeira discrepância será exibida.

3. Codifique o construtor de **SwingFC** desta forma:

```

SwingFC() {
    // Cria um contêiner JFrame.
    JFrame jfrm = new JFrame("Compare Files");

    // Especifica FlowLayout como gerenciador de layout.
    jfrm.setLayout(new FlowLayout());

    // Fornece um tamanho inicial para o quadro.
    jfrm.setSize(200, 220);

    // Encerra o programa quando o usuário fecha o aplicativo.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Cria os campos de texto para os nomes de arquivo.
    jtfFirst = new JTextField(14);
    jtfSecond = new JTextField(14);

    // Cria o botão Compare.
    JButton jbtnComp = new JButton("Compare");

    // Adiciona um ouvinte de ação para o botão Compare.
    jbtnComp.addActionListener(this);

    // Cria os rótulos.
    jlabFirst = new JLabel("First file: ");
    jlabSecond = new JLabel("Second file: ");
    jlabResult = new JLabel("");

    // Cria a caixa de seleção.
    jcbLoc = new JCheckBox("Show position of mismatch");

    // Adiciona os componentes ao painel de conteúdo.
    jfrm.add(jlabFirst);
    jfrm.add(jtfFirst);
    jfrm.add(jlabSecond);
    jfrm.add(jtfSecond);
    jfrm.add(jcbLoc);
}

```

```

jfrm.add(jbtnComp);
jfrm.add(jlabResult);

// Exibe o quadro.
jfrm.setVisible(true);
}

```

Você deve conhecer grande parte do código desse construtor. No entanto, observe uma coisa: um ouvinte de ação é adicionado apenas ao botão de ação **jbtnComp**.

Não são adicionados ouvintes de ação aos campos de texto. Veja por quê: o conteúdo dos campos de texto só é necessário quando o botão Compare é pressionado. Em nenhum outro momento seu conteúdo é necessário. Logo, não há por que responder a eventos de campos de texto. Ao criar mais programas Swing, você verá que é isso o que costuma ocorrer quando se usa um campo de texto.

- Comece a criar o tratador de eventos **actionPerformed()** como mostrado a seguir. Esse método é chamado quando o botão Compare é pressionado.

```

// Compara os arquivos quando o botão Compare é pressionado.
public void actionPerformed(ActionEvent ae) {
    int i = 0, j = 0;
    int count = 0;

    // Primeiro, confirma se os dois nomes de arquivo foram
    // inseridos.
    if(jtfFirst.getText().equals("")) {
        jlabResult.setText("First file name missing.");
        return;
    }
    if(jtfSecond.getText().equals("")) {
        jlabResult.setText("Second file name missing.");
        return;
    }
}

```

O método começa confirmando se o usuário inseriu um nome de arquivo em cada um dos campos de texto. Se não tiver inserido, o nome de arquivo ausente será relatado e o tratador retornará.

- Agora, termine **actionPerformed()** adicionando o código que realmente abre os arquivos e os compara.

```

// Compara arquivos. Usa try-with-resources para gerenciar os
// arquivos.
try (FileInputStream f1 =
        new FileInputStream(jtfFirst.getText());
     FileInputStream f2 =
        new FileInputStream(jtfSecond.getText()))
{
    // Verifica o conteúdo de cada arquivo.
    do {

```

```
i = f1.read();
j = f2.read();
if(i != j) break;
count++;
} while(i != -1 && j != -1);

if(i != j) {
    if(jcbLoc.isSelected())
        jlabResult.setText("Files differ at location " + count);
    else
        jlabResult.setText("Files are not the same.");
}
else
    jlabResult.setText("Files compare equal.");

} catch(IOException exc) {
    jlabResult.setText("File Error" + exc);
}
}
```

Observe que o estado da caixa de seleção é verificado para sabermos se o local da primeira discrepancia deve ser exibido.

6. Termine **SwingFC** adicionando o método **main()** a seguir:

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

    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SwingFC();
        }
    });
}
```

7. O programa inteiro de comparação de arquivos baseado no Swing é mostrado aqui:

```
/*
Tente isto 18-1

Um utilitário de comparação de arquivos baseado no Swing.

Ele usa a instrução try-with-resources e
requer o JDK 7 ou posterior.
*/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
```

```
class SwingFC implements ActionListener {  
  
    JTextField jtfFirst; // contém o nome do primeiro arquivo  
    JTextField jtfSecond; // contém o nome do segundo arquivo  
  
    JButton jbtnComp; // botão para comparar os arquivos  
  
    JLabel jlabFirst, jlabSecond; // exibe avisos  
    JLabel jlabResult; // exibe resultados e mensagens de erro  
  
    JCheckBox jcbLoc; // marque para exibir o local da discrepância.  
  
    SwingFC() {  
  
        // Cria um contêiner JFrame.  
        JFrame jfrm = new JFrame("Compare Files");  
  
        // Especifica FlowLayout como gerenciador de leiaute.  
        jfrm.setLayout(new FlowLayout());  
  
        // Fornece um tamanho inicial para o quadro.  
        jfrm.setSize(200, 220);  
  
        // Encerra o programa quando o usuário fecha o aplicativo.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Cria os campos de texto para os nomes de arquivo.  
        jtfFirst = new JTextField(14);  
        jtfSecond = new JTextField(14);  
  
        // Cria o botão Compare.  
        JButton jbtnComp = new JButton("Compare");  
  
        // Adiciona um ouvinte de ação para o botão Compare.  
        jbtnComp.addActionListener(this);  
  
        // Cria os rótulos.  
        jlabFirst = new JLabel("First file: ");  
        jlabSecond = new JLabel("Second file: ");  
        jlabResult = new JLabel("");  
  
        // Cria a caixa de seleção.  
        jcbLoc = new JCheckBox("Show position of mismatch");  
  
        // Adiciona os componentes ao painel de conteúdo.  
        jfrm.add(jlabFirst);  
        jfrm.add(jtfFirst);  
        jfrm.add(jlabSecond);  
        jfrm.add(jtfSecond);  
        jfrm.add(jcbLoc);  
        jfrm.add(jbtnComp);
```

```
jfrm.add(jlabResult);

// Exibe o quadro.
jfrm.setVisible(true);
}

// Compara os arquivos quando o botão Compare é pressionado.
public void actionPerformed(ActionEvent ae) {
    int i = 0, j = 0;
    int count = 0;

    // Primeiro, confirma se os dois nomes de arquivo foram
    // inseridos.
    if(jtfFirst.getText().equals("")) {
        jlabResult.setText("First file name missing.");
        return;
    }
    if(jtfSecond.getText().equals("")) {
        jlabResult.setText("Second file name missing.");
        return;
    }

    // Compara arquivos. Usa try-with-resources para gerenciar os
    // arquivos.
    try (FileInputStream f1 =
            new FileInputStream(jtfFirst.getText());
        FileInputStream f2 =
            new FileInputStream(jtfSecond.getText()))
    {

        // Verifica o conteúdo de cada arquivo.
        do {
            i = f1.read();
            j = f2.read();
            if(i != j) break;
            count++;
        } while(i != -1 && j != -1);

        if(i != j) {
            if(jcbLoc.isSelected())
                jlabResult.setText("Files differ at location " + count);
            else
                jlabResult.setText("Files are not the same.");
        }
        else
            jlabResult.setText("Files compare equal.");
    } catch(IOException exc) {
        jlabResult.setText("File Error");
    }
}
```

```

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SwingFC();
        }
    });
}
}

```

JScrollPane

JScrollPane é um contêiner leve que trata automaticamente a rolagem de outro componente leve. O componente rolado pode ser individual, como uma tabela, ou um grupo de componentes contidos dentro de outro contêiner leve, como um **JPanel**. Nos dois casos, se o objeto que estiver sendo rolado for maior do que a área visualizável, barras de rolagem horizontais e/ou verticais serão fornecidas automaticamente e o componente poderá ser rolado pelo painel. **JScrollPane** automatiza a rolagem, geralmente eliminando a necessidade de gerenciarmos barras de rolagem individuais.

A área visualizável de um painel de rolagem se chama *viewport*, que é uma janela em que o componente que está sendo rolado é exibido. Logo, o viewport exibe a parte visível do componente que está sendo rolado. As barras de rolagem rolam o componente por meio do viewport. Em seu comportamento padrão, o **JScrollPane** adiciona ou remove uma barra de rolagem conforme necessário. Por exemplo, se o componente for maior do que o viewport, uma barra de rolagem vertical será adicionada. Se o componente couber todo dentro do viewport, as barras de rolagem serão removidas.

JScrollPane define vários construtores. O usado neste capítulo é mostrado abaixo:

`JScrollPane(Component comp)`

O componente a ser rolado é especificado por *comp*. Barras de rolagem são exibidas automaticamente quando o conteúdo do painel excede as dimensões do viewport.

Estas são as etapas que devemos seguir para usar um painel de rolagem:

1. Crie o componente a ser rolado.
2. Crie uma instância de **JScrollPane**, passando para ela o objeto a ser rolado.
3. Adicione o painel de rolagem ao painel de conteúdo.

O exemplo a seguir ilustra um painel de rolagem. Ele começa criando um rótulo de várias linhas. Como explicado na caixa Pergunte ao especialista apresentada anteriormente, isso é feito com o uso de texto baseado em HTML. O rótulo é então adicionado a um painel de rolagem e este é adicionado ao painel de conteúdo. Como

o conteúdo do rótulo é maior do que o quadro que o contém, aparecem automaticamente uma barra de rolagem horizontal e uma vertical. Você pode usar as barras de rolagem para rolar o texto no viewport.

```
// Um exemplo simples de JScrollPane.
import javax.swing.*;

class JScrollPaneDemo {

    JScrollPaneDemo() {

        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("JScrollPane Example");

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(200, 120);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

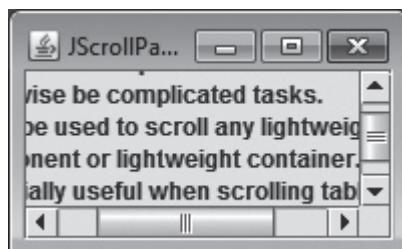
        // Cria um rótulo longo baseado em HTML.
        JLabel jlab =
            new JLabel("<html>JScrollPane simplifies what would<br>" +
                      "otherwise be complicated tasks.<br>" +
                      "It can be used to scroll any lightweight <br>" +
                      "component or lightweight container. It is <br>" +
                      "especially useful when scrolling tables, lists,<br>" +
                      "or images.");

        // Cria um painel de rolagem e o faz rolar o rótulo.
        JScrollPane jscrln = new JScrollPane(jlab); ← Cria um painel de
        // Adiciona o painel de rolagem ao quadro. ← rolagem que contém o
        jfrm.add(jscrln); ← Adiciona o painel de rolagem
                           ao painel de conteúdo.
        // Exibe o quadro.
        jfrm.setVisible(true);
    }

    public static void main(String[] args) {

        // Cria a GUI na thread de despacho de evento.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new JScrollPaneDemo();
            }
        });
    }
}
```

A saída do exemplo de painel de rolagem é mostrada aqui:



Verificação do progresso

1. Que método é chamado para obtermos o texto selecionado dentro de um **JTextField**?
2. **JScrollPane** sempre exibe barras de rolagem horizontais e verticais. Verdadeiro ou falso?
3. Em um **JScrollPane**, a área visualizável se chama _____.

TENTE ISTO 18-2 Role um JPanel

`ScrollPaneDemo.java`

Por ser um contêiner leve que herda **JComponent**, **JPanel** também pode ser rolado com o uso de **JScrollPane**. Esse recurso possibilita a rolagem do conteúdo inteiro de um **JPanel** quase sem nenhum esforço de nossa parte. Quando o espaço da tela é escasso, a rolagem de um painel pode ser a solução para uma situação que, de outra forma, seria difícil de resolver. Você pode simplesmente adicionar um conjunto de componentes ao painel e então usar um **JScrollPane** para rolar o conjunto.

Além disso, **JPanel** também é um componente, já que herda **JComponent**. Logo, **JPanel** é um componente leve que também pode ser usado para conter outros **JPanels**. Isso torna **JPanel** perfeito para a criação de um sistema de contenção com várias camadas.

Respostas:

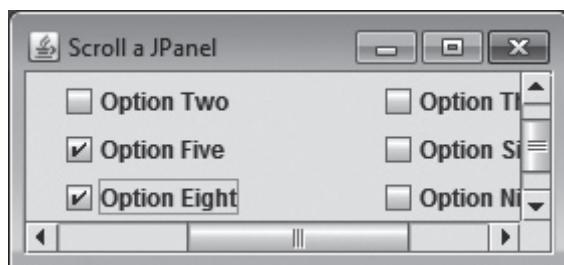
1. `getSelectedText()`
2. Falso. As barras de rolagem só são exibidas quando necessário.
3. `viewport`

JPanel define vários construtores. Neste exemplo, só o construtor padrão é usado. O exemplo mostra que você pode definir o gerenciador de layout da mesma forma que faria com um **JFrame**. Por padrão, um **JPanel** é criado com o layout de fluxo. Ele também usa um buffer duplo. Como mencionado no Capítulo 15, o buffer duplo é um mecanismo que normalmente é empregado para o fornecimento de uma melhor experiência do usuário quando ocorre a atualização da tela. Em vez de desenharmos cada componente diretamente na tela, o que pode produzir “oscilação”, os componentes são renderizados em um buffer separado. Quando a renderização é concluída, o buffer é copiado na tela em uma operação rápida e ininterrupta. Assim, o conteúdo de um painel aparece instantaneamente e em sua totalidade, em vez de lentamente e em partes. Você pode desativar o buffer duplo se desejar, mas raramente vai querer fazê-lo. O suporte ao buffer duplo é herdado de **JComponent**.

Este exemplo demonstra como criar um **JPanel**, adicionar componentes a ele e depois adicionar o painel a um **JScrollPane**, para permitir sua rolagem automática. Você notará que o gerenciador de layout do painel é configurado com **GridLayout**. Ele dispõe os componentes em uma grade bidimensional. O tamanho da grade pode ser especificado com o uso deste construtor:

```
GridLayout(int numLinhas, int numColunas)
```

Aqui, o número de linhas é passado em *numLinhas* e o número de colunas em *numColunas*. Nesse caso, a disposição dos componentes em uma grade torna os efeitos da rolagem bem visíveis. (No entanto, em seu código, você pode selecionar um gerenciador de layout apropriado às suas necessidades. Nem **JPanel** nem **JScrollPane** requerem o uso de **GridLayout**.) No exemplo, o painel contém um rótulo e uma série de caixas de seleção. Porém, você pode usar a técnica básica sempre que tiver que rolar o conteúdo de um painel. Um exemplo da saída é mostrado abaixo:



PASSO A PASSO

- Crie um arquivo chamado **ScrollJPanelDemo.java** e adicione o comentário e as instruções **import** a seguir:

```
// Tente isto 18-2: Use um JScrollPane para rolar um JPanel.

import java.awt.*;
import javax.swing.*;
```

2. Comece a classe **ScrollJPanelDemo** desta forma:

```
class ScrollJPanelDemo {
    ScrollJPanelDemo() {
        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("Scroll a JPanel");

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(280, 130);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

3. Adicione as linhas a seguir, que criam o rótulo e as caixas de seleção:

```
// Cria um rótulo.
JLabel jlabOptions = new JLabel("Select one or more options: ");

// Cria algumas caixas de seleção.
JCheckBox jcbOpt1 = new JCheckBox("Option One");
JCheckBox jcbOpt2 = new JCheckBox("Option Two");
JCheckBox jcbOpt3 = new JCheckBox("Option Three");
JCheckBox jcbOpt4 = new JCheckBox("Option Four");
JCheckBox jcbOpt5 = new JCheckBox("Option Five");
JCheckBox jcbOpt6 = new JCheckBox("Option Six");
JCheckBox jcbOpt7 = new JCheckBox("Option Seven");
JCheckBox jcbOpt8 = new JCheckBox("Option Eight");
JCheckBox jcbOpt9 = new JCheckBox("Option Nine");
JCheckBox jcbOpt10 = new JCheckBox("Option Ten");

// Nenhum tratador de eventos é usado por este exemplo,
// mas como exercício, você pode tentar adicionar alguns.
```

O objetivo desse programa é demonstrar a rolagem de um **JPanel**, por isso, nenhum tratador de eventos é incluído, porque as caixas de seleção são usadas apenas para exibição. No entanto, como teste, você pode tentar adicionar tratadores de eventos por conta própria.

4. Adicione o código de construção do **JPanel** que será rolado e depois adicione a ele o rótulo e as caixas de seleção.

```
// Cria o JPanel que conterá as caixas de seleção das opções.
JPanel jpn1 = new JPanel();
jpn1.setLayout(new GridLayout(5, 3));

// Adiciona as caixas de seleção e o rótulo ao JPanel.
jpn1.add(jlabOptions);
jpn1.add(new JLabel("")); // rótulo de espaço reservado
jpn1.add(new JLabel("")); // rótulo de espaço reservado
```

```
jpnl.add(jcbOpt1);
jpnl.add(jcbOpt2);
jpnl.add(jcbOpt3);
jpnl.add(jcbOpt4);
jpnl.add(jcbOpt5);
jpnl.add(jcbOpt6);
jpnl.add(jcbOpt7);
jpnl.add(jcbOpt8);
jpnl.add(jcbOpt9);
jpnl.add(jcbOpt10);
```

5. Crie o **JScrollPane**, usando o painel como o objeto a ser rolado. Em seguida, adicione o painel de rolagem ao painel de conteúdo. Para concluir, torne o quadro visível.

```
// Cria o painel de rolagem que rolará o outro painel.
JScrollPane jscrIp = new JScrollPane(jpnl);

// Adiciona o painel de rolagem ao quadro.
jfrm.add(jscrIp);

// Exibe o quadro.
jfrm.setVisible(true);
}
```

6. Termine a classe da maneira usual.

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

    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ScrollJPanelDemo();
        }
    });
}
```

7. O programa inteiro é mostrado aqui:

```
// Tente isto 18-2: Usa um JScrollPane para rolar um JPanel.

import java.awt.*;
import javax.swing.*;

class ScrollJPanelDemo {

    ScrollJPanelDemo() {

        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("Scroll a JPanel");

```

```
// Fornece um tamanho inicial para o quadro.  
jfrm.setSize(280, 130);  
  
// Encerra o programa quando o usuário fecha o aplicativo.  
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
// Cria um rótulo.  
JLabel jlabOptions = new JLabel("Select one or more options: ");  
  
// Cria algumas caixas de seleção.  
JCheckBox jcbOpt1 = new JCheckBox("Option One");  
JCheckBox jcbOpt2 = new JCheckBox("Option Two");  
JCheckBox jcbOpt3 = new JCheckBox("Option Three");  
JCheckBox jcbOpt4 = new JCheckBox("Option Four");  
JCheckBox jcbOpt5 = new JCheckBox("Option Five");  
JCheckBox jcbOpt6 = new JCheckBox("Option Six");  
JCheckBox jcbOpt7 = new JCheckBox("Option Seven");  
JCheckBox jcbOpt8 = new JCheckBox("Option Eight");  
JCheckBox jcbOpt9 = new JCheckBox("Option Nine");  
JCheckBox jcbOpt10 = new JCheckBox("Option Ten");  
  
// Nenhum tratador de eventos é usado por este exemplo,  
// mas, como exercício, você pode tentar adicionar alguns.  
  
// Cria o JPanel que conterá as caixas de seleção das opções.  
JPanel jpnl = new JPanel();  
jpnl.setLayout(new GridLayout(5, 3));  
  
// Adiciona as caixas de seleção e o rótulo ao JPanel.  
jpnl.add(jlabOptions);  
jpnl.add(new JLabel("")); // rótulo de espaço reservado  
jpnl.add(new JLabel("")); // rótulo de espaço reservado  
  
jpnl.add(jcbOpt1);  
jpnl.add(jcbOpt2);  
jpnl.add(jcbOpt3);  
jpnl.add(jcbOpt4);  
jpnl.add(jcbOpt5);  
jpnl.add(jcbOpt6);  
jpnl.add(jcbOpt7);  
jpnl.add(jcbOpt8);  
jpnl.add(jcbOpt9);  
jpnl.add(jcbOpt10);  
  
// Cria o painel de rolagem que rolará o outro painel.  
JScrollPane jscrlnp = new JScrollPane(jpnl);  
  
// Adiciona o painel de rolagem ao quadro.  
jfrm.add(jscrlnp);  
  
// Exibe o quadro.
```

```
        jfrm.setVisible(true);
    }

    public static void main(String[] args) {
        // Cria a GUI na thread de despacho de evento.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new ScrollJPanelDemo();
            }
        });
    }
}
```

JList

No Swing, a classe de lista básica se chama **JList**. Ela dá suporte à seleção de um ou mais itens em uma lista. Embora geralmente a lista seja composta por strings, é possível criar uma lista com quase qualquer objeto que possa ser exibido. **JList** é tão usada em Java que é provável que você já a tenha visto.

No passado, os itens de um **JList** eram representados como referências **Object**. No entanto, com o lançamento do JDK 7, **JList** tornou-se genérico e agora é declarado assim:

```
class JList<E>
```

Aqui, **E** representa o tipo dos itens da lista. Como resultado, agora **Jlist** tem segurança de tipos.

JList fornece vários construtores. O que usaremos é

```
JList(E[ ] itens)
```

Esse construtor cria um **JList** contendo os itens do array especificado por *itens*.

Embora um **JList** funcione apropriadamente sozinho, quase sempre ele é encapsulado dentro de um **JScrollPane**. Dessa forma, listas longas são automaticamente roláveis, o que simplifica o design da GUI. Também fica fácil alterar o número de entradas de uma lista sem ser preciso mudar o tamanho do componente **JList**.

Um **JList** gera um **ListSelectionEvent** quando o usuário faz ou altera uma seleção. Esse evento também é gerado quando o usuário desmarca um item. Ele é tratado pela implementação de **ListSelectionListener**. Esse ouvinte especifica apenas um método, chamado **valueChanged()**, que é mostrado abaixo:

```
void valueChanged(ListSelectionEvent le)
```

Aqui, *le* é uma referência ao evento. Embora **ListSelectionEvent** forneça alguns métodos próprios, geralmente é preciso interrogar o objeto **JList** para saber o que ocorreu. Tanto **ListSelectionEvent** quanto **ListSelectionListener** ficam no pacote **javax.swing.event**.

Por padrão, um **JList** permite que o usuário selecione vários intervalos de itens dentro da lista, mas você pode alterar esse comportamento chamando o método **setSelectionMode()**, que é definido por **JList**. Ele é mostrado aqui:

```
void setSelectionMode(int modo)
```

Nesse caso, *modo* especifica o modo de seleção. Ele deve ter um dos valores a seguir definidos por **ListSelectionModel** (que é o modelo usado por **JList**):

```
SINGLE_SELECTION  
SINGLE_INTERVAL_SELECTION  
MULTIPLE_INTERVAL_SELECTION
```

A seleção padrão de múltiplos intervalos permite que o usuário selecione vários intervalos de itens em uma lista. Na seleção de intervalo simples, o usuário pode selecionar um intervalo de itens. Na seleção simples, o usuário só pode selecionar um item. É claro que um único item também pode ser selecionado nos outros dois modos, mas eles também permitem que um intervalo seja selecionado.

Você pode obter o índice do primeiro item selecionado, que também será o índice do único item selecionado quando o modo de seleção simples for usado, chamando o método **getSelectedIndex()**, mostrado aqui:

```
int getSelectedIndex()
```

A indexação começa em zero. Logo, se o primeiro item estiver selecionado, esse método retornará 0. Se nenhum item estiver selecionado, -1 será retornado.

Em vez de obter o índice de uma seleção, você pode obter o valor associado a ela chamando o método **getSelectedValue()**:

```
E getSelectedValue()
```

Ele retorna uma referência ao primeiro valor selecionado. Se nenhum valor estiver selecionado, **null** será retornado.

O programa a seguir demonstra um **JList** simples, que contém uma lista de variedades de maçã. Sempre que uma maçã é selecionada na lista, é gerado um **ListSelectionEvent**, que é tratado pelo método **valueChanged()** definido por **ListSelectionListener**. Ele responde obtendo o índice do item selecionado e exibindo o nome da maçã selecionada em um rótulo. Observe que ele define o tamanho da lista chamando **setPreferredSize()**.

```
// Demonstra um JList simples.

// Este programa requer o JDK 7 ou posterior.

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
```

```
class JListDemo {  
  
    JList<String> jlst;  
    JLabel jlab;  
    JScrollPane jscrlp;  
  
    // Cria um array de variedades de maçãs.  
    String[] apples = { "Winesap", "Cortland", "Red Delicious",  
                        "Golden Delicious", "Gala", "Fuji",  
                        "Granny Smith", "Jonathan" };  
  
    JListDemo() {  
        // Cria um contêiner JFrame.  
        JFrame jfrm = new JFrame("JList Demo");  
  
        // Especifica o gerenciador FlowLayout.  
        jfrm.setLayout(new FlowLayout());  
  
        // Fornece um tamanho inicial para o quadro.  
        jfrm.setSize(240, 200);  
  
        // Encerra o programa quando o usuário fecha o aplicativo.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Cria um JList.                                     Cria um JList a partir de  
        jlst = new JList<String>(apples); ← um array de strings.  
  
        // Define o modo de seleção da lista com a seleção simples.  
        jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION); ← Define o modo  
        // Adiciona a lista a um painel de rolagem.          de seleção com a  
        jscrlp = new JScrollPane(jlst); ← Insere o JList em um  
        // Define o tamanho ótimo do painel de rolagem.       painel de rolagem.  
        jscrlp.setPreferredSize(new Dimension(120, 90));  
  
        // Cria um rótulo para exibir a seleção.  
        jlab = new JLabel("Please Choose an Apple..");  
  
        // Adiciona um ouvinte de seleção para a lista.  
        jlst.addListSelectionListener(new ListSelectionListener() { ← Ouve eventos  
            public void valueChanged(ListSelectionEvent le) {           de seleção.  
                // Obtém o índice do item alterado.  
                int idx = jlst.getSelectedIndex(); ← Obtém o índice da  
                // Exibe a seleção, se o item for selecionado.  
                if(idx != -1)  
            }  
        });
```

```

        jlab.setText("Current selection: " + apples[idx]);
    else // Caso contrário, solicita novamente.
        jlab.setText("Please Choose an Apple.");
    }

// Adiciona a lista e o rótulo ao painel de conteúdo.
jfrm.add(jscrlp);
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JListDemo();
        }
    });
}
}

```

A saída do exemplo de lista é mostrada aqui:



Algo que seria interessante você testar é selecionar um item em uma lista sob controle do programa. Isso pode ser feito com uma chamada ao método **setSelectedIndex()**, que é mostrado abaixo:

```
void setSelectedIndex(int índice)
```

Aqui, *índice* especifica o índice do item que você deseja selecionar na lista. O índice tem base zero, ou seja, o primeiro item da lista fica no índice zero. Você pode desmarcar uma seleção controlada pelo programa chamando **clearSelection()**:

```
void clearSelection()
```

Quando esse método é executado, todas as seleções são desmarcadas.

Pergunte ao especialista

P Na discussão sobre o **JList**, você mencionou que o exemplo define o tamanho de um componente usando **setPreferredSize()**. Poderia explicar?

R Sim. Por padrão, o tamanho de um componente é determinado por seu conteúdo e pelo gerenciador de leiaute. No entanto, você pode especificar explicitamente um tamanho ideal para um componente chamando **setPreferredSize()**, que é definido por **JComponent**. Ele é mostrado abaixo:

```
void setPreferredSize(Dimension novaDO)
```

Aqui, *novaDO* especifica a nova dimensão ótima para o componente. A classe **Dimension** faz parte do pacote **java.awt**. Este é um de seus construtores:

```
Dimension(int l, int a)
```

Nele, *l* especifica a largura e *a* a altura.

Uma vez que você tiver definido o tamanho ótimo, um gerenciador de leiaute usará essas dimensões como guia para dimensionar apropriadamente o componente. É bom ressaltar, no entanto, que alguns gerenciadores de leiaute, como **GridLayout**, ignoram essas dimensões.

Uma última coisa: você também pode especificar um tamanho mínimo e um tamanho máximo chamando **setMinimumSize()** e **setMaximumSize()**, também definidos por **JComponent**. Eles são mostrados abaixo.

```
void setMaximumSize(Dimension novoTamanho)
void setMinimumSize(Dimension novoTamanho)
```

Como em **setPreferredSize()**, as dimensões definidas por esses métodos são sugestões que podem ser ignoradas pelo gerenciador de leiaute.

JComboBox

O Swing fornece uma *caixa de combinação* (uma combinação de campo de texto com lista suspensa) por intermédio da classe **JComboBox**. No passado, os itens de um **JComboBox** eram representados como referências **Object**. No entanto, com o lançamento do JDK 7, **JComboBox** tornou-se genérico e agora é declarado assim:

```
class JComboBox<E>
```

E é o tipo dos elementos contidos na lista.

Normalmente uma caixa de combinação exibe uma entrada, mas ela também exibe uma lista suspensa para permitir que o usuário selecione uma entrada diferente. Você também pode criar uma caixa de combinação que permita ao usuário inserir uma seleção no campo de texto, mas não faremos isso aqui.

O construtor de **JComboBox** usado pelo exemplo é mostrado abaixo:

```
JComboBox(E[ ] itens)
```

Nesse caso, *itens* é um array que inicializa a caixa de combinação. Há outros construtores disponíveis.

JComboBox gera um evento de ação quando o usuário seleciona um item na lista. Também gera um evento de item quando o estado da seleção muda, o que ocorre quando um item é marcado ou desmarcado. Logo, a mudança de uma seleção significa que dois eventos de item ocorrerão: um para o item desmarcado e outro para o item selecionado. Muitas vezes, é suficiente apenas ouvir eventos de ação, mas os dois tipos de evento estão disponíveis para uso.

Uma maneira de obter o item selecionado na lista é chamando o método **getSelectedItem()** na caixa de combinação. Ele é mostrado aqui:

```
Object getSelectedItem()
```

Você terá que converter o valor retornado para o tipo de objeto armazenado na lista.

O exemplo a seguir demonstra **JComboBox** retrabalhando o exemplo anterior de **JList**.

```
// Demonstra uma caixa de combinação simples.

// Este programa requer o JDK 7 ou posterior.

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class JComboBoxDemo {

    JComboBox<String> jcbb;
    JLabel jlab;

    // Cria um array de variedades de maçãs.
    String[] apples = { "Winesap", "Cortland", "Red Delicious",
                        "Golden Delicious", "Gala", "Fuji",
                        "Granny Smith", "Jonathan" };

    JComboBoxDemo() {
        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("JComboBox Demo");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(380, 240);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

```

// Cria um JComboBox.
jcbb = new JComboBox<String>(apples); ← Cria um JComboBox a partir
// de um array de strings.

// Cria um rótulo para exibir a seleção.
jlab = new JLabel("Please Choose an Apple.");

// Adiciona um ouvinte de ação para a caixa de combinação.
jcbb.addActionListener(new ActionListener() { ← Ouve eventos de ação na
    public void actionPerformed(ActionEvent ae) {   caixa de combinação.
        // Obtém uma referência ao item selecionado.
        String item = (String) jcbb.getSelectedItem(); ← Obtém o item
            selecionado.

        // Exibe o item selecionado.
        jlab.setText("Current selection: " + item);
    }
});

// Adiciona a caixa de combinação e o rótulo ao painel de conteúdo.
jfrm.add(jcbb);
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JComboBoxDemo();
        }
    });
}
}

```

A saída do exemplo de caixa de combinação é mostrada aqui:



Um recurso útil da caixa de combinação que você pode querer testar é a possibilidade de adicionar ou remover itens na lista suspensa dinamicamente, durante a execução do programa. Esse recurso é suportado pelos métodos **addItem()** e **removeItem()**, mostrados abaixo:

```
void addItem(E item)
void removeItem(Object item)
```

Aqui, *item* é o item a ser adicionado ou removido. Um ponto importante: esses métodos só estão disponíveis para caixas de combinação mutáveis. Felizmente, por padrão, é isso o que ocorre com **JComboBox**. Quando um item é adicionado, ele é inserido no fim da lista.

Verificação do progresso

1. Que controle cria uma lista?
2. Em um **JList**, o índice do primeiro item selecionado na lista pode ser obtido com o uso de que método?
3. Em um **JComboBox**, que método obtém a seleção atual?

ÁRVORES

A *árvore* é um componente que apresenta uma visão hierárquica dos dados. O usuário pode expandir ou reduzir subárvore individuais com essa estrutura. As árvores são implementadas no Swing pela classe **JTree**. Dois de seus construtores são mostrados abaixo:

```
JTree(Object[ ] obj)
JTree(TreeNode tn)
```

Na primeira forma, a árvore é construída a partir dos elementos do array *obj*. Na segunda, ela é definida pela árvore cujo nó raiz é especificado por *tn*.

Embora **JTree** faça parte do pacote **javax.swing**, as classes e interfaces que lhe dão suporte ficam no pacote **javax.swing.tree**. Isso ocorre porque o número de classes e interfaces necessárias para **JTree** ter suporte é muito grande.

Um **JTree** gera vários eventos. O tratado pelo exemplo de programa mostrado nesta seção é **TreeSelectionEvent**. Para ouvir esse evento, é preciso implementar a interface **TreeSelectionListener**. Ela só define um método, chamado **valueChanged()**, que recebe o objeto **TreeSelectionEvent**. Sua forma é:

```
void valueChanged(TreeSelectionEvent te)
```

Respostas:

1. **JList**
2. **getSelectedIndex()**
3. **getSelectedItem()**

Você pode obter o caminho do objeto selecionado chamando o método **getPath()**, mostrado a seguir, no objeto de evento:

```
TreePath getPath()
```

O método retorna um objeto **TreePath** que descreve o caminho do nó selecionado. A classe **TreePath** encapsula informações sobre o caminho de um nó específico em uma árvore. Ela fornece vários construtores e métodos. Aqui, só um é usado: **getLastPathComponent()**. Sua forma geral é:

```
Object getLastPathComponent()
```

Ele retorna o último nó do caminho selecionado atualmente.

A interface **TreeNode** declara métodos que obtêm informações sobre o nó de uma árvore. Por exemplo, é possível obter uma referência ao nó pai ou uma enumeração dos nós filhos. A interface **MutableTreeNode** estende **TreeNode**. Ela declara métodos que podem inserir e remover nós filhos ou mudar o nó pai.

A classe **DefaultMutableTreeNode** implementa a interface **MutableTreeNode**. Ela representa um nó de uma árvore. Um de seus construtores é mostrado abaixo:

```
DefaultMutableTreeNode(Object obj)
```

Aqui, *obj* é o objeto a ser inserido nesse nó da árvore. O novo nó da árvore não tem um pai ou filhos.

Para criarmos uma hierarquia de nós de árvore, podemos usar o método **add()** de **DefaultMutableTreeNode**. Ele é mostrado a seguir:

```
void add(MutableTreeNode filho)
```

Nesse caso, *filho* é um nó de árvore mutável que deve ser adicionado como filho ao nó atual.

JTree não fornece nenhum recurso próprio de rolagem. Em vez disso, normalmente é inserido dentro de um **JScrollPane**. Dessa forma, uma árvore grande pode ser rolada em um viewport menor.

Estas são as etapas que você pode seguir para usar uma árvore:

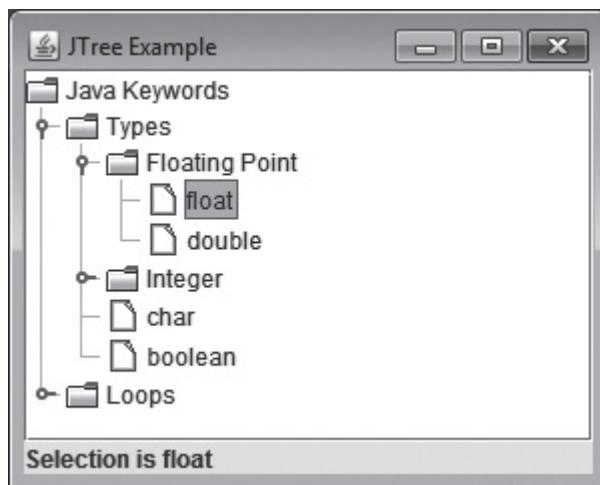
1. Crie uma instância de **JTree**.
2. Adicione nós à árvore.
3. Crie um **JScrollPane** e especifique a árvore como o objeto a ser rolado.
4. Adicione o painel de rolagem ao painel de conteúdo.

O próximo exemplo ilustra como criar uma árvore e manipular seleções. Ele cria uma árvore que organiza várias das palavras-chave Java. O programa cria uma instância de **DefaultMutableTreeNode** chamada “Java Keywords”. Essa é a raiz hierárquica da árvore. Subárvore adicionais para tipos e laços são criadas. Nós finais são adicionados a essas subárvore indicando palavras-chave. Em todos os casos, o método **add()** é chamado para conectar os nós à árvore. Uma referência ao nó raiz da árvore é passada como argumento para o construtor de **JTree**. A árvore é então fornecida como argumento para o construtor de **JScrollPane**. O painel de rolagem é adicionado ao painel de conteúdo. Em seguida, um rótulo é criado e adicionado ao

painel de conteúdo. A seleção feita na árvore é exibida nesse rótulo. Para recebermos eventos de seleção, um **TreeSelectionListener** é registrado para a árvore. Dentro do método **valueChanged()**, a seleção atual é obtida e exibida. Embora este exemplo exiba apenas algumas palavras-chave Java, você pode expandi-lo para incluir palavras-chave adicionais.

```
// Cria nós para char e boolean.  
types.add(new DefaultMutableTreeNode("char")); ————— Adiciona nós para  
types.add(new DefaultMutableTreeNode("boolean")); ————— char e boolean.  
  
// Cria subárvore para laços.  
DefaultMutableTreeNode loops = new DefaultMutableTreeNode("Loops"); —————  
root.add(loops);  
loops.add(new DefaultMutableTreeNode("for"));  
loops.add(new DefaultMutableTreeNode("while")); ————— Cria subárvore  
loops.add(new DefaultMutableTreeNode("do")); ————— para laços.  
  
// Cria a árvore.  
tree = new JTree(root); ←———— Constrói um JTree a partir da árvore recém-criada.  
  
// Adiciona a árvore a um painel de rolagem. ————— Insere a árvore em um  
JScrollPane jsp = new JScrollPane(tree); ←———— painel de rolagem.  
  
// Adiciona o painel de rolagem ao centro do BorderLayout padrão.  
jfrm.add(jsp);  
  
// Adiciona o rótulo à região sul do BorderLayout padrão.  
jlab = new JLabel("Select from the tree.");  
jfrm.add(jlab, BorderLayout.SOUTH);  
  
// Trata eventos de seleção na árvore.  
tree.addTreeSelectionListener(new TreeSelectionListener() {  
    public void valueChanged(TreeSelectionEvent tse) {  
        jlab.setText("Selection is " +  
                    tse.getPath().getLastPathComponent());  
    }  
});  
  
// Exibe o quadro.  
jfrm.setVisible(true);  
}  
  
public static void main(String[] args) {  
    // Cria a GUI na thread de despacho de evento.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new JTreeDemo();  
        }  
    });  
}
```

A saída do exemplo de árvore é mostrada abaixo:



JTable

JTable é um componente que exibe linhas e colunas de dados. Para redimensionar as colunas, só temos que arrastar o cursor sobre os limites da coluna. Também podemos arrastar uma coluna para uma nova posição. Dependendo de sua configuração, é possível selecionar uma linha, coluna ou célula da tabela, e alterar os dados de uma célula. **JTable** é um componente sofisticado que oferece muito mais opções e recursos do que os discutidos aqui. (Talvez seja o componente mais complicado do Swing.) No entanto, mesmo em sua configuração padrão, oferece funcionalidade significativa e fácil de usar – principalmente se só quisermos usar a tabela para apresentar dados em formato tabular. Este resumo lhe dará uma ideia geral desse poderoso componente.

Como **JTree**, a classe **JTable** tem muitas classes e interfaces associadas a ela. Elas ficam no pacote **javax.swing.table**.

Em essência, **JTable** é conceitualmente simples. É um componente com uma ou mais colunas de informações. No topo de cada coluna há um cabeçalho. Além de descrever os dados da coluna, o cabeçalho também fornece o mecanismo pelo qual o usuário pode alterar o tamanho ou o local de uma coluna na tabela. **JTable** não fornece nenhum recurso próprio de rolagem. Em vez disso, normalmente inserimos um **JTable** dentro de um **JScrollPane**.

JTable fornece vários construtores. O que usaremos é

```
JTable(Object[ ] [ ] dados, Object[ ] cabeçalhosDeColuna)
```

Aqui, *dados* é um array bidimensional com as informações a serem apresentadas, e *cabeçalhosDeColuna* é um array unidimensional com os cabeçalhos das colunas.

Um **JTable** pode gerar vários eventos diferentes. No entanto, o tratamento desses eventos requer um pouco mais de trabalho do que tratar os eventos gerados pelos componentes já descritos e não faz parte do escopo deste livro. Mas se você quiser usar **JTable** apenas para exibir dados e permitir que eles sejam editados (como o exemplo a seguir faz), não terá que tratar nenhum evento.

Estas são as etapas requeridas para a definição de um **JTable** simples que pode ser usado para exibir dados:

1. Crie uma instância de **JTable**.
2. Crie um objeto **JScrollPane**, especificando a tabela como o objeto a ser rolado.
3. Adicione o painel de rolagem ao painel de conteúdo.

O próximo exemplo ilustra como criar e usar uma tabela simples. Um array unidimensional de strings chamado **Headings** é criado para os cabeçalhos das colunas. Um array bidimensional de strings chamado **data** é criado para as células da tabela. Observe que cada elemento do array é um array de quatro strings. Esses arrays são passados para o construtor de **JTable**. A tabela é adicionada a um painel de rolagem e o painel de rolagem é adicionado ao painel de conteúdo. A tabela exibe os dados do array **data**. A configuração padrão das tabelas também permite que o conteúdo de uma célula seja editado. As alterações afetam o array subjacente, que nesse caso é **data**.

```
// Demonstra JTable.
import java.awt.*;
import javax.swing.*;

public class JTableDemo {

    JTableDemo() {

        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("JTable Example");

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(400, 300);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Inicializa os cabeçalhos das colunas.
        String[] Headings = { "Name", "Student ID", "Midterm", "Final" };

        // Inicializa data.
        String[][] data = {
            { "Tom", " 4-616", " 97", " 87" },
            { "Ken", " 3-786", " 88", " 95" },
            { "Rachel", " 4-674", " 92", " 83" },
            { "Sherry", " 3-235", " 91", " 99" },
            { "Adam", " 2-923", " 76", " 96" },
            { "Jon", " 3-561", " 84", " 80" },
            { "Stuart", " 1-337", " 62", " 74" },
        };
    }
}
```

```

    { "Mary", " 4-731", " 68", " 58" },
    { "Todd", " 2-924", " 82", " 72" },
    { "Shane", " 2-434", " 93", " 91" },
    { "Robert", " 3-769", " 99", " 92" },
};

// Cria a tabela.
JTable table = new JTable(data, Headings); ← Cria um JTable usando os dados
// e cabeçalhos especificados.

// Adiciona a tabela a um painel de rolagem.
JScrollPane jsp = new JScrollPane(table); ← Insere a tabela em um
// painel de rolagem.

// Adiciona o painel de rolagem ao painel de conteúdo.
jfrm.add(jsp);

// Exibe o quadro.
jfrm.setVisible(true);
}

public static void main(String[] args) {

    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JTableDemo();
        }
    });
}
}

```

A saída desse exemplo é mostrada abaixo:

Name	Student ID	Midterm	Final
Tom	4-616	97	87
Ken	3-786	88	95
Rachel	4-674	92	83
Sherry	3-235	91	99
Adam	2-923	76	96
Jon	3-561	84	80
Stuart	1-337	62	74
Mary	4-731	68	58
Todd	2-924	82	72
Shane	2-434	93	91
Robert	3-769	99	92

Verificação do progresso

1. Um controle de árvore é criado com o uso de que classe?
2. Quando um item de um **JTree** é selecionado, que evento é gerado?
3. Que controle cria uma tabela?

UMA EXPLICAÇÃO RÁPIDA DOS MODELOS

Para dar suporte à arquitetura Modelo-Delegação descrita no capítulo anterior, a maioria dos controles do Swing utiliza um modelo que determina como os dados associados a um controle são tratados. Por exemplo, o modelo para um botão é definido pela interface **ButtonModel**. Os exemplos deste capítulo não fizeram uso explícito de nenhum dos modelos associados a um controle. Agimos assim porque provavelmente seu código não terá que usar um modelo diretamente, já que (como regra geral) o componente expõe a funcionalidade do modelo. Além disso, para muitas aplicações (talvez a maioria), o modelo padrão fornecido pelos controles é o desejado. Logo, geralmente não há razão para alterá-lo. No entanto, o Swing permite o trabalho com o modelo diretamente, se necessário, o que pode ser útil em GUIs mais avançadas. A possibilidade de acessar o modelo é outra vantagem que torna o Swing tão poderoso.

Pergunte ao especialista

P Além dos controles que você mencionou, que outros tipos de controles o Swing oferece?

R O Swing oferece uma rica variedade de controles que facilitam a criação de GUIs amigáveis e sofisticadas. Em primeiro lugar, e o mais importante, o Swing dá suporte a um poderoso sistema de menus, que é descrito no Capítulo 19. Em segundo lugar, ele dá suporte a vários controles de diálogo internos, que são descritos no Capítulo 20. Para concluir, o Swing dá suporte a muitos outros controles de interface de usuário. Por exemplo, esses são alguns que você pode achar interessantes. Seus nomes indicam sua função.

JFormattedTextField	JPasswordField	JProgressBar
JScrollBar	JSlider	JSpinner
JTabbedPane	JTextArea	JToolBar

Você vai querer testar esses controles do Swing, e outros, à medida que aprimorar suas habilidades de programação de GUIs.

Respostas:

1. **JTree**
2. **TreeSelectionEvent**
3. **JTable**

EXERCÍCIOS

1. **JLabel** gera um **ActionEvent**?
2. Que evento é gerado quando um botão de ação é pressionado?
3. **JButton** pode incluir um ícone?
4. Que controle se alterna entre dois estados: ativo e inativo?
5. No uso de **JTextField**, o recurso de recortar e colar é suportado por quais métodos?
6. Mostre como criar um campo de texto com 32 colunas.
7. Podemos definir o comando de ação de um **JTextField**? Se sim, como?
8. Que componente do Swing cria uma caixa de seleção? Que evento é gerado quando uma caixa de seleção é marcada ou desmarcada?
9. **JRadioButton** cria uma lista de botões com a forma de rádios. Verdadeiro ou falso?
10. **JList** exibe uma lista de itens para o usuário selecionar. Verdadeiro ou falso?
11. Que evento é gerado quando o usuário marca ou desmarca um item em um **JList**?
12. O que **JScrollPane** faz?
13. Que método define o modo de seleção de um **JList**? Que método obtém o índice do primeiro item selecionado?
14. Para exibir informações em formato tabular, você pode usar _____.
15. Para exibir informações em formato de árvore, você pode usar _____.
16. O que é **JComboBox**?
17. Melhore o programa **JCheckBoxDemo** discutido neste capítulo para que o texto de cada caixa de seleção fique vermelho se a caixa for marcada e preto se for desmarcada. Para configurar a cor com vermelho, use HTML no texto e insira-o entre as tags **** e ****.
18. Altere o programa **JCheckBoxDemo** discutido neste capítulo adicionando um novo **JButton** com o texto “Count”. Quando o botão for clicado, o rótulo **jlabSupported** exibirá o número de caixas de seleção marcadas em vez de uma lista dos sistemas operacionais suportados.
19. Modifique o programa da seção Tente isto 18-2 para que o rótulo **jlabOptions** não faça parte do painel de rolagem e fique centralizado acima de todas as caixas de seleção, não importando o tamanho da janela. Para fazer isso, adicione **jlabOptions** ao lado norte do painel de conteúdo da janela. Ao centro do painel de conteúdo, adicione o painel de rolagem com as caixas de seleção.
20. Dois construtores da classe **JTree** foram mencionados neste capítulo, mas só o segundo foi usado. O primeiro construtor tem um array de elementos **Object** como seu parâmetro. Como esse array é convertido em uma árvore? Isto é, qual é a raiz da árvore e quais são os outros nós? Para descobrir, crie um pequeno programa de teste.

21. Na seção que descreve a classe **JTree**, usamos o termo “raiz” para o nó que é passado como argumento para o construtor de **JTree**. Por que usamos a palavra “raiz”?
22. Na seção que descreve a classe **JTree**, usamos os termos “nó pai”, “nó filho”, “subárvore” e “nó final”. Defina esses termos em suas próprias palavras.
23. Escreva um programa que crie um **JFrame**. O **JFrame** exibirá um **JTree** mostrando a hierarquia de herança das classes Java que fazem parte do Swing discutidas neste capítulo. A raiz da árvore é a classe **JComponent**. Ela tem nove filhos: **AbstractButton**, **JComboBox**, **JLabel**, **JList**, **JScrollPane**, **JTable**, **JTextComponent**, **JTree** e **JPanel**. A classe **AbstractButton** tem dois filhos: **JButton** e **JToggleButton**. A classe **JToggleButton** tem dois filhos: **JCheckBox** e **JRadioButton**. A classe **JTextComponent** tem um filho: **JTextField**. Insira a árvore em um painel de rolagem. Não é preciso lidar com nenhum evento.
24. Retrabalhe a versão do sistema de ajuda da seção Tente isto 4-1 do Capítulo 4 para que use um **JList** para exibir os tópicos da ajuda em uma janela. Use o construtor de **JList** que tem um array como seu parâmetro. O array deve conter os tópicos (como strings). Quando o usuário clicar em um item da lista, as informações de ajuda sobre esse tópico serão exibidas em um rótulo na parte inferior da janela.
25. Escolha três imagens. Em seguida, escreva um programa que crie uma janela contendo apenas um **JToggleButton**. O botão aparecerá exibindo a primeira imagem. Quando você passar o mouse sobre ele, a segunda imagem será exibida. Quando você clicar no botão para selecioná-lo, ele exibirá a terceira imagem. A classe **JToggleButton** tem um método **setSelectedIcon()** que define o ícone a ser exibido quando o botão é pressionado. Ele tem esta forma:

```
void setSelectedIcon(Icon íconeSelecionado)
```

26. Modifique o exemplo **JComboBoxDemo** do texto de modo que, se um item do **JComboBox** for marcado, ele seja excluído da caixa.
27. Escreva um programa que crie uma janela com o conteúdo a seguir:
 - A. Um rótulo centralizado no topo exibindo “Escolha uma cor e um estilo”.
 - B. Um rótulo centralizado na parte inferior exibindo “Exemplo de texto”.
 - C. Uma coluna à esquerda composta por um rótulo exibindo “Estilo” e duas caixas de seleção chamadas “Negrito” e “Itálico”.
 - D. Uma coluna à direita composta por um rótulo exibindo “Cor” e três botões de rádio exibindo “Vermelho”, “Verde” e “Preto”.

Inicialmente o botão de rádio preto estará pressionado. Se o usuário clicar em um botão de rádio diferente, o exemplo de texto da parte inferior será alterado para a nova cor. Se o usuário marcar uma ou mais caixas de seleção à esquerda, os estilos selecionados serão aplicados ao exemplo de texto da parte inferior. Para alterar o estilo e a cor do texto, use HTML. Por exemplo, para

usar vermelho, defina o texto do rótulo da parte inferior com “<html>Exemplo de texto.”. Para usar negrito, itálico e vermelho, defina o texto do rótulo com “<html><i>Exemplo de texto.</i>”. A tag **** indica texto em negrito e **<i>** indica texto em itálico.

28. Tanto **JComboBox** quanto **JList** exibem uma lista de itens para que um item seja selecionado. Por que dois controles fazem a mesma coisa?
29. Escreva um programa que crie uma janela. Nela, insira um **JTable** com duas linhas e duas colunas, com os cabeçalhos de coluna “Nome” e “Sobrenome”. Insira nomes arbitrários nas duas linhas. Inclua um botão “Classificar” sob a tabela. Quando o botão for clicado, o programa verificará se as duas linhas estão classificadas alfabeticamente pelo sobrenome. Se não estiverem, ele trocará os dados das duas linhas. Use os métodos **getValueAt()** e **setValueAt()** da classe **JTable** para fazer a verificação e a troca. Veja os detalhes desses métodos:

```
Object getValueAt(int linha, int coluna)
Object setValueAt(Object novoValor, int linha, int coluna)
```

O primeiro método retorna o valor da linha e da coluna especificadas. Observe que seu tipo de retorno é **Object**, logo, você terá que converter o valor que ele retorna para um string. Além disso, é bom ressaltar que as linhas e colunas começam no índice 0. O segundo método define o valor da linha e da coluna especificadas com o novo valor.

30. Qual é a diferença entre uma caixa de seleção e um botão de rádio que não faz parte de nenhum grupo de botões de rádio?
31. Há duas maneiras de tratar eventos em **JToggleButton**: você pode criar um **ItemListener** para tratar **ItemEvents** do botão ou usar um **ActionListener** para tratar **ActionEvents**. Há alguma vantagem em usar um em vez do outro?
32. Qual é a principal diferença entre um **JButton** e um **JToggleButton**?
33. Escreva um programa que crie uma janela. Na janela adicione um **JList** de strings. O **JList** deve usar o modo de seleção múltipla. (Para selecionar mais de um item, mantenha a tecla do comando ou controle pressionada em seu teclado quando clicar em um item.) Sempre que um item for selecionado na lista, seu programa deve compará-lo com todos os outros itens selecionados. Se todos os itens selecionados forem iguais, um rótulo na parte inferior da janela exibirá a mensagem “Todas as x seleções são iguais”, em que x é o número de itens selecionados. Se não forem, ele exibirá a mensagem “Não são todos iguais”. Para obter todos os itens selecionados, use o método **getSelectedIndices()** da classe **JList**. Veja os detalhes:

```
int[ ] getSelectedIndices()
```

Ele retorna um array com os índices dos itens selecionados. Quando não há itens selecionados, um array de tamanho 0 é retornado.

19

Trabalhando com menus

PRINCIPAIS HABILIDADES E CONCEITOS

- Saber os aspectos básicos dos menus
- **JMenuBar**
- **JMenu**
- **JMenuItem**
- Criar um menu principal
- Adicionar mnemônicos e aceleradores a itens de menu
- Adicionar imagens e dicas de ferramentas a itens de menu
- **JRadioButtonMenuItem**
- **JCheckBoxMenuItem**

A GUI moderna é composta principalmente por dois recursos. O primeiro são os controles, que foram introduzidos no capítulo anterior. O segundo são os menus, que serão introduzidos agora. Os menus são parte integrante de todos os aplicativos, exceto os mais simples, porque exibem a funcionalidade do programa para o usuário. Devido a sua importância, o Swing dá amplo suporte aos menus. Não examinaremos todos os seus recursos aqui, mas o capítulo descreverá seus aspectos básicos, além de várias opções normalmente usadas.

ASPECTOS BÁSICOS DOS MENUS

O sistema de menus do Swing é suportado por várias classes. As discutidas neste capítulo são mostradas abaixo:

Classes de menus	Descrição
JMenuBar	Um objeto que contém o menu de nível superior do aplicativo.
JMenu	Menu padrão. Um menu é composto por um ou mais JMenuItem s.
JMenuItem	Um objeto que preenche os menus.
JCheckBoxMenuItem	Um item de menu de caixa de seleção.
JRadioButtonMenuItem	Um item de menu de botão de rádio.

Embora inicialmente os menus pareçam um pouco complicados, os menus do Swing são muito fáceis de usar. O Swing permite um alto nível de personalização, se desejado, mas geralmente usamos as classes de menus como se encontram porque elas dão suporte à maioria das opções necessárias. Por exemplo, podemos adicionar facilmente imagens e atalhos do teclado a um menu.

Aqui está um resumo rápido de como as classes se encaixam. Para criar um menu principal para um aplicativo, primeiro você criará um objeto **JMenuBar**. De modo geral, essa classe é um contêiner para menus. À instância de **JMenuBar**, você adicionará instâncias de **JMenu**. Cada objeto **JMenu** define um menu, isto é, cada objeto **JMenu** contém um ou mais itens selecionáveis. Os itens exibidos por um **JMenu** são objetos de **JMenuItem**. Logo, um **JMenuItem** define uma seleção que pode ser feita pelo usuário.

Além dos itens de menu “padrão”, você também pode incluir caixas de seleção e botões de rádio em um menu. Um item de menu de caixa de seleção é criado por **JCheckBoxMenuItem**. Um item de menu de botão de rádio é criado por **JRadioButtonMenuItem**. Essas duas classes estendem **JMenuItem**. Elas podem ser usadas tanto em menus padrão quanto em menus popup.

Um aspecto-chave dos menus do Swing é que todos os itens do menu estendem **AbstractButton**. Lembre-se, **AbstractButton** também é superclasse de todos os componentes de botão do Swing, como **JButton**. Logo, todos os itens de menu são, essencialmente, botões. É claro que eles não se parecerão realmente com botões quando usados em um menu, mas, em muitos aspectos, agirão como botões. Por exemplo, a seleção de um item de menu gera um evento de ação da mesma forma que o pressionamento de um botão.

Outro ponto importante é que **JMenuItem** é superclasse de **JMenu**. Isso permite a criação de submenus, que são, essencialmente, menus dentro de menus. Para criar um submenu, primeiro você deve criar e preencher um objeto **JMenu** para depois adicioná-lo a outro objeto **JMenu**. Você verá esse processo em ação na próxima seção.

Como mencionado anteriormente, quando um item de menu é selecionado, um evento de ação é gerado. Por padrão, o string do comando de ação associado a esse evento será o nome da opção. Logo, você pode determinar qual item foi selecionado examinando o comando de ação. Claro, há outras abordagens. Por exemplo, você também pode usar uma classe interna anônima separada para tratar os eventos de ação de cada item de menu. Mas cuidado, porque os sistemas de menus tendem a ficar muito grandes. O uso de uma classe separada para o tratamento dos eventos de cada item de menu pode fazer um grande número de classes serem criadas.

Verificação do progresso

1. Que classe cria uma barra de menus de nível superior?
2. Que classe cria um item de menu?
3. Um item de menu pode ser uma caixa de seleção ou um botão de rádio?

Respostas:

1. **JMenuBar**
2. **JMenuItem**
3. Sim.

UMA VISÃO GERAL DE JMenuBar, JMenu E JMenuItem

Antes de você criar um menu, é preciso saber algo sobre as três classes de menus básicas: **JMenuBar**, **JMenu** e **JMenuItem**. Elas formam o conjunto mínimo de classes necessárias à construção de um menu principal para um aplicativo. Logo, essas classes formam a base do sistema de menus do Swing.

JMenuBar

Como mencionado, **JMenuBar** é basicamente um contêiner para menus. Como todos os componentes, ela herda **JComponent** (que herda **Container** e **Component**). Essa classe tem apenas um construtor, que é o construtor padrão. Portanto, inicialmente a barra estará vazia e você terá que preenchê-la com menus antes de usá-la. Cada aplicativo tem uma e somente uma barra de menus.

JMenuBar define vários métodos, mas geralmente só precisamos usar um: **add()**. O método **add()** adiciona um **JMenu** à barra de menus. Ele é mostrado abaixo:

```
JMenu add(JMenu menu)
```

Aqui, *menu* é uma instância de **JMenu** que é adicionada à barra de menus. Uma referência ao menu é retornada. Os menus são posicionados na barra da esquerda para a direita, na ordem em que são adicionados. Se você quiser adicionar um menu em um local específico, use esta versão de **add()**, que é herdada de **Container**:

```
Component add(Component menu, int índice)
```

Nesse caso, o *menu* é adicionado no índice especificado por *índice*. A indexação começa em 0, com 0 sendo o menu da extrema esquerda.

Em alguns casos podemos querer remover um menu que não é mais necessário. Faremos isso chamando o método **remove()**, que é herdado de **Container**. Ele tem estas duas formas:

```
void remove(Component menu)  
void remove(int índice)
```

Agora, *menu* é uma referência ao menu a ser removido, e *índice* é o índice do menu. A indexação começa em 0.

Outro método que pode ser útil é **getMenuCount()**, mostrado abaixo:

```
int getMenuCount()
```

Ele retorna o número de elementos contidos dentro da barra de menus.

Uma vez que uma barra de menus tiver sido criada e preenchida, ela será adicionada a um **JFrame** com uma chamada a **setJMenuBar()** na instância de **JFrame**. O método **setJMenuBar()** é mostrado abaixo:

```
void setJMenuBar(JMenuBar barramenus)
```

Aqui, *barramenus* é uma referência à barra de menus. A barra de menus será exibida em uma posição determinada pela aparência. Geralmente, ela fica no topo da janela.

JMenu

JMenu encapsula um menu, que é preenchido com **JMenuItem**s. Como mencionado, ela é derivada de **JMenuItem**. Ou seja, um **JMenu** pode ser uma opção em outro **JMenu**. Isso permite que um menu seja submenu de outro menu. **JMenu** define vários construtores. O que usaremos neste capítulo é:

```
JMenu(String nome)
```

Ele cria um menu que tem o título especificado por *nome*. O menu estará vazio até itens serem adicionados.

JMenu define muitos métodos. Aqui, veremos breves descrições de alguns dos mais usados. Para adicionar um item ao menu, use o método **add()**, que tem várias formas, inclusive estas duas:

```
JMenuItem add(JMenuItem item)
```

```
JMenuItem add(Component item, int índice)
```

Aqui, *item* é o item de menu a ser adicionado. A primeira forma adiciona o item ao fim do menu. A segunda o adiciona no índice especificado por *índice*. Como esperado, a indexação começa em 0. (A segunda versão é herdada de **Container**. Normalmente, passamos uma referência **JMenuItem** para *item*.) Os dois retornam uma referência ao item adicionado. Também é possível usar **insert()** para adicionar itens a um menu.

Você pode adicionar um separador visual a um menu chamando o método **addSeparator()**, mostrado abaixo:

```
void addSeparator()
```

O separador é adicionado no fim do menu. É possível inserir um separador em um índice especificado com uma chamada ao método **insertSeparator()**, mostrado a seguir:

```
void insertSeparator(int índice)
```

Aqui, *índice* especifica o índice baseado em zero em que o separador será adicionado.

Se quiser remover um item de um menu, chame **remove()**. Abaixo temos duas de suas formas:

```
void remove(JMenuItem menu)
```

```
void remove(int índice)
```

Nesse caso, *menu* é uma referência ao item a ser removido e *índice* é o índice do item.

Você pode obter o número de itens do menu chamando o método **getMenuComponentCount()**, mostrado aqui:

```
int getMenuComponentCount()
```

E pode obter um array dos itens do menu chamando o método **getMenuComponents()**, cuja forma é:

```
Component[ ] getMenuComponents()
```

Um array contendo os componentes é retornado.

JMenuItem

JMenuItem encapsula um elemento de um menu. Esse elemento pode ser uma opção vinculada a alguma ação do programa, como Salvar ou Fechar, ou pode fazer um submenu ser exibido. Como mencionado, **JMenuItem** é derivado de **AbstractButton**, e todo item de um menu pode ser considerado um tipo especial de botão. Portanto, quando um item de menu é selecionado, um evento de ação é gerado. (Isso é semelhante à maneira como um **JButton** aciona um evento de ação quando é pressionado.) **JMenuItem** define vários construtores. Este é o que usaremos primeiro:

```
JMenuItem(String nome)
```

Ele cria um item de menu com o nome especificado por *nome*.

Já que os itens de menu herdam **AbstractButton**, você terá acesso à funcionalidade fornecida por essa classe. Por exemplo, você pode ativar/desativar um item de menu chamando o método **setEnabled()**, mostrado aqui:

```
void setEnabled(boolean enable)
```

Se *enable* for igual a **true**, o item de menu foi ativado. Se for **false**, ele está desativado e não pode ser selecionado.

Verificação do progresso

1. Que método adiciona um item a um menu?
2. O que **addSeparator()** faz?
3. Um item de menu não pode ser desativado. Verdadeiro ou falso?

CRIE UM MENU PRINCIPAL

Talvez o menu mais usado seja o *menu principal*. Ele é definido pela barra de menus e estabelece a funcionalidade principal de um aplicativo. Felizmente, o Swing facilita muito a criação e o gerenciamento do menu principal. Esta seção mostrará como construir um menu principal básico. As seções subsequentes mostrarão como adicionar opções a ele.

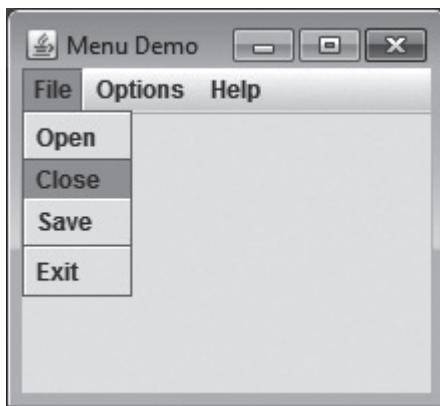
A construção do menu principal requer várias etapas. Primeiro, crie o objeto **JMenuBar** que conterá os menus. Em seguida, construa cada menu que estará na barra de menus. Em geral, um menu é construído com a criação de um objeto **JMenu** e então com a inclusão de **JMenuItem**s. Após os menus terem sido criados, adicione-os à barra de menus. Agora a própria barra de menus deve ser adicionada ao quadro com uma

Respostas:

1. **add()**
2. Adiciona um separador (que separa visualmente os itens de menu) a um menu.
3. Falso.

chamada a `setJMenuBar()`. Para concluir, para cada item de menu, você deve adicionar um ouvinte de ação que trate o evento de ação acionado quando o item é selecionado.

A melhor maneira de entender o processo de criação e gerenciamento de menus é usando um exemplo. Este é um programa que cria uma barra de menus simples contendo três menus. O primeiro é um menu File padrão que contém as opções Open, Close, Save e Exit. O segundo se chama Options e contém dois submenus chamados Colors e Priority. O terceiro menu se chama Help e tem um item chamado About. Quando um item de menu é selecionado, o nome da opção é exibido em um rótulo no painel de conteúdo. Um exemplo da saída é mostrado aqui:



```
// Demonstra um menu principal simples.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MenuDemo implements ActionListener {

    JLabel jlab;

    MenuDemo() {
        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("Menu Demo");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(220, 200);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void actionPerformed(ActionEvent evt) {
        String s = evt.getActionCommand();
        jlab.setText(s);
    }
}
```

```

// Cria um rótulo que exibirá a seleção feita no menu.
jlab = new JLabel();

// Cria a barra de menus.
JMenuBar jmb = new JMenuBar(); ← Cria uma barra de menus.

// Cria o menu File.
JMenu jmFile = new JMenu("File");
JMenuItem jmiOpen = new JMenuItem("Open");
JMenuItem jmiClose = new JMenuItem("Close");
JMenuItem jmiSave = new JMenuItem("Save");
JMenuItem jmiExit = new JMenuItem("Exit");
jmFile.add(jmiOpen); ← Cria os itens do menu File.
jmFile.add(jmiClose);
jmFile.add(jmiSave); ← Adiciona os itens ao menu File.
jmFile.addSeparator();
jmFile.add(jmiExit); ← Adiciona o menu File à barra de menus.

jmb.add(jmFile); ← Adiciona o menu File à barra de menus.

// Cria o menu Options.
JMenu jmOptions = new JMenu("Options"); ← Cria o menu Options.

// Cria o submenu Colors.
JMenu jmColors = new JMenu("Colors");
JMenuItem jmiRed = new JMenuItem("Red");
JMenuItem jmiGreen = new JMenuItem("Green");
JMenuItem jmiBlue = new JMenuItem("Blue");
jmColors.add(jmiRed); ← Cria o submenu Colors.
jmColors.add(jmiGreen);
jmColors.add(jmiBlue); ← Adiciona o submenu Colors ao menu Options.

jmOptions.add(jmColors); ← Adiciona o submenu Colors ao menu Options.

// Cria o submenu Priority.
JMenu jmPriority = new JMenu("Priority");
JMenuItem jmiHigh = new JMenuItem("High");
JMenuItem jmiLow = new JMenuItem("Low");
jmPriority.add(jmiHigh); ← Cria o submenu Priority.
jmPriority.add(jmiLow); ← Adiciona o submenu Priority ao menu Options.

jmOptions.add(jmPriority); ← Adiciona o submenu Priority ao menu Options.

// Cria o item de menu Reset.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator(); ← Cria o item de menu Reset e
jmOptions.add(jmiReset); ← o adiciona ao menu Options.

// Para finalizar, adiciona o menu Options inteiro à barra de menus
jmb.add(jmOptions); ← Adiciona o menu Options à barra de menus.

// Cria o menu Help.
JMenu jmHelp = new JMenu("Help");
JMenuItem jmiAbout = new JMenuItem("About"); ← Cria o menu Help e o
jmHelp.add(jmiAbout); ← adiciona à barra de menus.

```

```

jmHelp.add(jmiAbout);
jmb.add(jmHelp);

// Adiciona ouvintes de ação para os itens de menu.
jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
jmiAbout.addActionListener(this); └─ Adiciona os ouvintes de ação para os itens de menu.

// Adiciona o rótulo ao painel de conteúdo.
jfrm.add(jlab);

// Adiciona a barra de menus ao quadro.
jfrm.setJMenuBar(jmb); ← Adiciona a barra de menus ao quadro.

// Exibe o quadro.
jfrm.setVisible(true);
}

// Trata eventos de ação dos itens de menu.
public void actionPerformed(ActionEvent ae) {
    // Obtém o comando de ação da seleção feita no menu.
    String comStr = ae.getActionCommand();

    // Se o usuário selecionar Exit, encerra o programa.
    if(comStr.equals("Exit")) System.exit(0); ← Encerra o programa quando o usuário seleciona Exit no menu File.

    // Caso contrário, exibe a opção selecionada.
    jlab.setText(comStr + " Selected");
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MenuDemo();
        }
    });
}

```

Examinemos em detalhes como os menus são criados, começando com o construtor de **MenuDemo**. Ele começa com as instruções de praxe. Em seguida,

a barra de menus é construída e uma referência a ela é atribuída a **jmb** por esta instrução:

```
// Cria a barra de menus.  
JMenuBar jmb = new JMenuBar();
```

Em seguida, o menu File **jmFile** e suas entradas são criados por esta sequência:

```
// Cria o menu File.  
JMenu jmFile = new JMenu("File");  
JMenuItem jmiOpen = new JMenuItem("Open");  
JMenuItem jmiClose = new JMenuItem("Close");  
JMenuItem jmiSave = new JMenuItem("Save");  
JMenuItem jmiExit = new JMenuItem("Exit");
```

Os nomes Open, Close, Save e Exit serão exibidos como opções no menu. Agora, as entradas do menu são adicionadas ao menu File pela sequência abaixo:

```
jmFile.add(jmiOpen);  
jmFile.add(jmiClose);  
jmFile.add(jmiSave);  
jmFile.addSeparator();  
jmFile.add(jmiExit);
```

Para concluir, o menu File é adicionado à barra de menus por esta linha:

```
| jmb.add(jmFile);
```

Quando a sequência de código anterior for concluída, a barra de menus terá uma entrada: File. O menu File conterá quatro opções nesta ordem: Open, Close, Save e Exit. No entanto, observe que um separador foi adicionado antes de Exit. Ele separa visualmente o item de menu Exit das três opções anteriores.

O menu Options é construído com o uso do mesmo processo básico do menu File. Porém, ele é composto por dois submenus, Colors e Priority, e uma entrada Reset. Primeiro os submenus são construídos individualmente e então são adicionados ao menu Options. O item Reset é adicionado por último. Em seguida, o menu Options é adicionado à barra de menus. O menu Help é construído pelo mesmo processo.

Observe que **MenuDemo** implementa a interface **ActionListener**, e eventos de ação gerados por uma seleção no menu são tratados pelo método **actionPerformed()** definido por **MenuDemo**. Portanto, o programa adiciona **this** como o ouvinte de ação dos itens de menu. Repare que nenhum ouvinte é adicionado aos itens Colors e Priority porque eles não são realmente opções. Apenas ativam submenus.

Por fim, a barra de menus é adicionada ao quadro pela linha a seguir:

```
| jfrm.setJMenuBar(jmb);
```

As barras de menus não são adicionadas ao painel de conteúdo. São adicionadas diretamente ao **JFrame**.

O método **actionPerformed()** trata os eventos de ação gerados pelo menu. Ele obtém o string do comando de ação associado à seleção chamando **getActionCom-**

mand() no evento e armazena uma referência a esse string em **comStr**. Em seguida, compara o comando de ação com “Exit”, como mostrado aqui:

```
| if(comStr.equals("Exit")) System.exit(0);
```

Se o comando de ação for “Exit”, o programa será encerrado com uma chamada a **System.exit()**. Esse método causa o encerramento imediato de um programa e passa seu argumento como código de status para o processo chamador, que geralmente é o sistema operacional ou o navegador. Por convenção, um código de status igual a 0 significa encerramento normal. Qualquer outro valor indica que o programa terminou anormalmente.

A essa altura, seria recomendável você fazer alguns testes com o programa **MenuDemo**. Tente adicionar outro menu ou adicionar mais itens a um menu existente. É importante que você entenda os conceitos básicos dos menus antes de avançarmos, porque aperfeiçoaremos o programa durante o resto deste capítulo.

ADICIONE MNEMÔNICOS E ACELERADORES AOS ITENS DE MENU

O menu criado no exemplo anterior é funcional, mas podemos melhorá-lo. Em aplicativos reais, geralmente um menu inclui o suporte a atalhos do teclado. Esses atalhos podem ter duas formas: mnemônicos e aceleradores. No que diz respeito aos menus, um mnemônico define uma tecla cuja digitação permite a seleção de um item em um menu ativo. Logo, um mnemônico permite o uso do teclado na seleção de um item em um menu que já está sendo exibido. Um acelerador é uma tecla que permite a seleção de um item de menu sem ser preciso ativar o menu antes. Por exemplo, você pode usar CTRL-S para ativar a função “salvar”.

Um mnemônico pode ser especificado para objetos **JMenuItem** e **JMenu**. Há duas maneiras de definir o mnemônico para **JMenuItem**. Em primeiro lugar, ele pode ser especificado quando um objeto for construído com o uso deste construtor:

```
JMenuItem(String nome, int mnem)
```

Nele, o mnemônico é especificado por *mnem*. Em segundo lugar, você pode definir o mnemônico chamando **setMnemonic()**. Para especificar um mnemônico para **JMenu**, temos que chamar **setMnemonic()**. Esse método é herdado de **AbstractButton** pelas duas classes e é mostrado abaixo:

```
void setMnemonic(int mnem)
```

Aqui, *mnem* especifica o mnemônico. Ele deve ser uma das constantes definidas em **java.awt.event.KeyEvent**, que define constantes nomeadas para as teclas do teclado, como **KeyEvent.VK_A**, **KeyEvent.VK_B**, **KeyEvent.VK_C**, e assim por diante. As teclas não alfabeticas também são definidas. Os mnemônicos não diferenciam maiúsculas de minúsculas, logo, tomando como exemplo **VK_A**, podemos digitar *a* ou *A*.

Por padrão, a primeira letra coincidente do item de menu será sublinhada. Nos casos em que você quiser sublinhar uma letra diferente da primeira coincidente, es-

pecifique o índice da letra como argumento do método `setDisplayedMnemonicIndex()`, que é herdado tanto por **JMenu** quanto por **JMenuItem** de **AbstractButton**. Ele é mostrado aqui:

```
void setDisplayedMnemonicIndex(int índice)
```

O índice da letra a ser sublinhada é especificado por *índice*.

Um acelerador pode ser associado a um objeto **JMenuItem**. Ele é especificado com uma chamada ao método `setAccelerator()`, mostrado a seguir:

```
void setAccelerator(KeyStroke ks)
```

Nele, *ks* é a combinação de teclas que é pressionada na seleção do item de menu. **KeyStroke** é uma classe que contém métodos factory para a construção de vários tipos de aceleradores por pressionamento de teclas. O usado aqui é

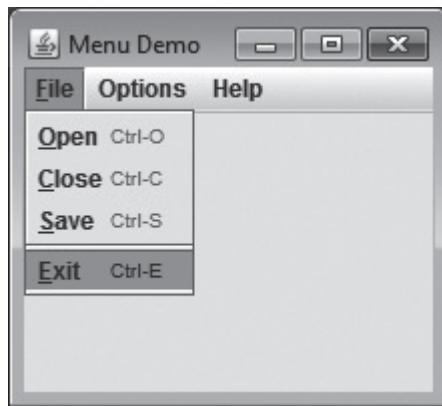
```
static KeyStroke getKeyStroke(int ch, int modificador)
```

Nesse caso, *ch* especifica o caractere acelerador, que é um valor de tipo **KeyEvent**, descrito anteriormente. O valor do *modificador* deve ser uma ou mais das constantes a seguir, definidas na classe **java.awt.event.InputEvent**:

InputEvent.ALT_DOWN_MASK	InputEvent.ALT_GRAPH_DOWN_MASK
InputEvent.CTRL_DOWN_MASK	InputEvent.META_DOWN_MASK
InputEvent.SHIFT_DOWN_MASK	

Portanto, se você passar **VK_A** como a tecla de caractere e **InputEvent.CTRL_DOWN_MASK** como o modificador, a combinação de teclas do acelerador será CTRL-A.

A sequência a seguir adiciona tanto mnemônicos quanto aceleradores ao menu File criado pelo programa **MenuDemo** da seção anterior. O menu ficará assim quando ativado:



```
// Cria o menu File com mnemônicos e aceleradores.
JMenu jmFile = new JMenu("File");
jmFile.setMnemonic(KeyEvent.VK_F); ← O menu File tem F como seu mnemônico.

JMenuItem jmiOpen = new JMenuItem("Open",
                                  KeyEvent.VK_O);
jmiOpen.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_O,
                          InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiClose = new JMenuItem("Close",
                                   KeyEvent.VK_C);
jmiClose.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_C,
                          InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiSave = new JMenuItem("Save",
                                 KeyEvent.VK_S);
jmiSave.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_S,
                          InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiExit = new JMenuItem("Exit",
                                 KeyEvent.VK_E);
jmiExit.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_E,
                          InputEvent.CTRL_DOWN_MASK));
```

Em todos os casos, o mnemônico é a primeira letra do nome do item. O acelerador é a mesma letra em combinação com a tecla CTRL.

Após fazer essa alteração, você poderá selecionar o menu File digitando ALT-F. Em seguida, poderá usar os mnemônicos O, C, S ou E para selecionar uma opção. Como alternativa, você pode selecionar uma opção do menu File diretamente pressionando CTRL-O, CTRL-C, CTRL-S ou CTRL-E.

Pergunte ao especialista

P Já que os aceleradores funcionam com o menu sendo exibido ou não, por que deveria me preocupar em definir mnemônicos para seleções no menu?

R Embora os aceleradores possam ser usados isoladamente, pela razão mencionada em sua pergunta, eles apresentam uma desvantagem: devem ser usados junto com uma tecla modificadora, como CTRL ou ALT. No entanto, especificando um mnemônico, você dará ao usuário a opção de selecionar um item apenas digitando sua tecla (sem um modificador) quando um menu for exibido. Por exemplo, se você definir a letra *S* como o mnemônico de uma opção Salvar e CTRL-S como seu acelerador, quando o menu for exibido, o usuário poderá selecionar Salvar simplesmente digitando *S*, sem ter que pressionar também a tecla CTRL. Mesmo parecendo irrelevante, é assim que muitos aplicativos reais funcionam. Logo, para que seus programas tenham uma aparência profissional, tanto os mnemônicos quanto os aceleradores são necessários.

ADICIONE IMAGENS E DICAS DE FERRAMENTAS AOS ITENS DE MENU

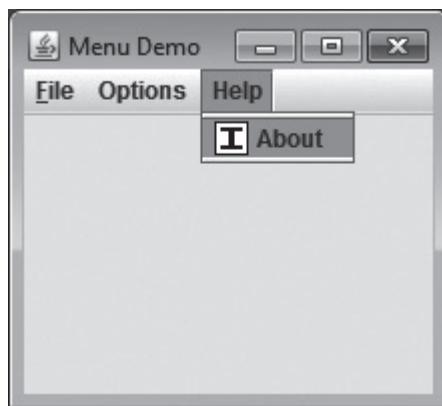
Os itens de menu não estão limitados somente a texto. Você pode usar imagens em vez de texto, ou usar ambos. A maneira mais fácil de adicionar uma imagem é especificá-la quando o item de menu é construído. Por exemplo, estes são os construtores de **JMenuItem** que permitem a inclusão de um ícone no item de menu:

```
JMenuItem(Icon imagem)
JMenuItem(String nome, Icon imagem)
```

O primeiro cria um item de menu que exibe a imagem especificada por *imagem*. O segundo cria um item de menu com o nome especificado por *nome* e a imagem especificada por *imagem*. Por exemplo, aqui uma imagem é adicionada ao item de menu About quando ele é criado:

```
 ImageIcon icon = new ImageIcon("AboutIcon.gif");
JMenuItem jmiAbout = new JMenuItem("About", icon);
```

Após esse acréscimo, o ícone especificado por **icon** será exibido próximo ao texto “About” quando o menu Help for aberto, como mostrado abaixo:



Você também pode adicionar um ícone a um item de menu após o item ter sido criado chamando `setIcon()` (que é herdado de `AbstractButton`).

Também podemos especificar um ícone de desativação, que é exibido quando o item de menu estiver inativo, chamando `setDisabledIcon()`. Normalmente, quando um item de menu está inativo, o ícone padrão é exibido em cinza. Se um ícone de desativação for especificado, ele será exibido quando o item de menu estiver inativo.

Uma *dica de ferramenta* é um texto pequeno que é exibido automaticamente quando o mouse passa sobre um componente. Ela costuma ser usada para dar ao usuário informações adicionais sobre o componente. No Swing, você pode adicionar facilmente uma dica de ferramenta a qualquer componente leve. Isso ocorre porque `JComponent` (que, claro, é a classe base de todos os componentes leves do Swing) fornece um método chamado `setToolTipText()`. Ele é mostrado aqui:

TENTE ISTO 19-1 Adicione e remova dinamicamente itens de menu

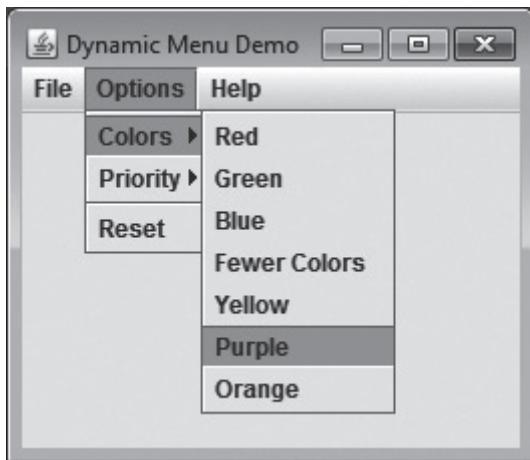
`DynMenuDemo.java`

É possível alterar o conteúdo de um menu durante a execução do programa. Por exemplo, você pode adicionar um item quando ele for necessário e removê-lo quando não precisar mais dele. Também pode alterar o nome de um item no tempo de execução. Esses recursos possibilitam a criação de menus dinâmicos que mudam quando preciso para atender às necessidades do usuário. Nossa demonstração mostra o processo percorrendo as etapas necessárias à inclusão e remoção de cores adicionais ao menu Colors do programa `MenuDemo` mostrado no início deste capítulo.

```
void setToolTipText(String stringDaDica)
```

Esse método adiciona a dica de ferramenta especificada por `stringDaDica` ao componente. Como todos os itens de menu herdam `JComponent`, você pode adicionar

O projeto adiciona um item de menu ao menu Colors chamado More Colors. Quando More Colors é selecionado, ele é alterado para Fewer Colors e faz as cores Yellow, Purple e Orange serem adicionadas ao menu. Quando Fewer Colors é selecionado, as cores são removidas e Fewer Colors volta a ser More Colors. Um exemplo da saída é mostrado aqui:



PASSO A PASSO

1. Copie o programa **MenuDemo** mostrado no início deste capítulo para um arquivo chamado **DynMenuDemo.java**. (Se você vem acompanhando o desenvolvimento, poderá usar a versão de **MenuDemo** que inclui os aceleradores, os mnemônicos e o ícone. No entanto, para simplificar, o código mostrado aqui usa apenas o programa **MenuDemo** original em seu ponto de partida.)
2. Altere o nome da classe de **MenuDemo** para **DynMenuDemo**. Em seguida, adicione três **JMenuItem**s chamados **jmiYellow**, **jmiPurple** e **jmiOrange** como variáveis de instância. Converta também a variável local **jmiColors** em uma variável de instância. Após essas alterações, o início do programa terá esta aparência:

```
// Tente isto 19-1: Adiciona e remove itens de menu dinamicamente.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class DynMenuDemo implements ActionListener {
    JLabel jlab;
    JMenuItem jmiYellow;
```

```

JMenuItem jmiPurple;
JMenuItem jmiOrange;

JMenu jmColors;
```

3. Crie um item de menu chamado **jmiMoreLess** com o nome More Colors e adicione-o ao menu **jmColors**, como mostrado abaixo:

```

// Cria o item de menu More/Fewer Colors.
JMenuItem jmiMoreLess = new JMenuItem("More Colors");
jmColors.add(jmiMoreLess);
```

4. Crie as opções de cores Yellow, Purple e Orange desta forma:

```

// Cria as cores adicionais. Elas serão
// adicionadas ou removidas sob demanda.
jmiYellow = new JMenuItem("Yellow");
jmiPurple = new JMenuItem("Purple");
jmiOrange = new JMenuItem("Orange");
```

5. Adicione ouvintes para os novos itens de menu, como mostrado a seguir:

```

// Adiciona ouvintes para a opção de cores adicionais.
jmiMoreLess.addActionListener(this);
jmiYellow.addActionListener(this);
jmiPurple.addActionListener(this);
jmiOrange.addActionListener(this);
```

6. Altere o método **actionPerformed()** para tratar as novas opções, como mostrado aqui:

```

// Trata eventos de ação dos itens de menu.
public void actionPerformed(ActionEvent ae) {
    // Obtém o comando de ação da seleção feita no menu.
    String comStr = ae.getActionCommand();

    // Se o usuário selecionar Exit, encerra o programa.
    if(comStr.equals("Exit"))
        System.exit(0);
    else if(comStr.equals("More Colors")) {
        jmColors.add(jmiYellow);
        jmColors.add(jmiPurple);
        jmColors.add(jmiOrange);
        JMenuItem mi = (JMenuItem) ae.getSource();
        mi.setText("Fewer Colors");
    } else if(comStr.equals("Fewer Colors")) {
        jmColors.remove(jmiYellow);
        jmColors.remove(jmiPurple);
        jmColors.remove(jmiOrange);
        JMenuItem mi = (JMenuItem) ae.getSource();
        mi.setText("More Colors");
    }

    // Caso contrário, exibe a opção selecionada.
    jlab.setText(comStr + " Selected");
}
```

Observe que, quando More Colors é selecionado, as três novas cores são adicionadas ao menu Colors e o nome do item de menu **jmiMoreLess** é alterado para Fewer Colors. Quando Fewer Colors é selecionado, as novas cores são removidas do menu e o nome de **jmiMoreLess** volta a ser More Colors. Lembre-se, um item por ser removido de um menu com uma chamada a **remove()**.

7. Após todas as alterações, o programa ficará com esta aparência:

```
// Tente isto 19-1: Adiciona e remove itens de menu dinamicamente.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class DynMenuDemo implements ActionListener {

    JLabel jlab;

    JMenuItem jmiYellow;
    JMenuItem jmiPurple;
    JMenuItem jmiOrange;

    JMenu jmColors;

    DynMenuDemo() {
        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("Dynamic Menu Demo");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(220, 200);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria um rótulo que exibirá a seleção feita no menu.
        jlab = new JLabel();

        // Cria a barra de menus.
        JMenuBar jmb = new JMenuBar();

        // Cria o menu File.
        JMenu jmFile = new JMenu("File");
        JMenuItem jmiOpen = new JMenuItem("Open");
        JMenuItem jmiClose = new JMenuItem("Close");
        JMenuItem jmiSave = new JMenuItem("Save");
        JMenuItem jmiExit = new JMenuItem("Exit");
    }
}
```

```
jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
jmb.add(jmFile);

// Cria o menu Options.
JMenu jmOptions = new JMenu("Options");

// Cria o submenu Colors.
jmColors = new JMenu("Colors");
JMenuItem jmiRed = new JMenuItem("Red");
JMenuItem jmiGreen = new JMenuItem("Green");
JMenuItem jmiBlue = new JMenuItem("Blue");
jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);

// Cria o item de menu More/Fewer Colors.
JMenuItem jmiMoreLess = new JMenuItem("More Colors");
jmColors.add(jmiMoreLess);

// Adiciona o menu Colors ao menu Options.
jmOptions.add(jmColors);

// Cria as cores adicionais. Elas serão
// adicionadas ou removidas sob demanda.
jmiYellow = new JMenuItem("Yellow");
jmiPurple = new JMenuItem("Purple");
jmiOrange = new JMenuItem("Orange");

// Cria o submenu Priority.
JMenu jmPriority = new JMenu("Priority");
JMenuItem jmiHigh = new JMenuItem("High");
JMenuItem jmiLow = new JMenuItem("Low");
jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);

// Adiciona o menu Priority ao menu Options.
jmOptions.add(jmPriority);

// Cria o item de menu Reset.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Para finalizar, adiciona o menu Options
// inteiro à barra de menus
jmb.add(jmOptions);
```

```
// Cria o menu Help.  
JMenu jmHelp = new JMenu("Help");  
JMenuItem jmiAbout = new JMenuItem("About");  
jmHelp.add(jmiAbout);  
jmb.add(jmHelp);  
  
// Adiciona ouvintes de ação para os itens de menu.  
jmiOpen.addActionListener(this);  
jmiClose.addActionListener(this);  
jmiSave.addActionListener(this);  
jmiExit.addActionListener(this);  
jmiRed.addActionListener(this);  
jmiGreen.addActionListener(this);  
jmiBlue.addActionListener(this);  
jmiHigh.addActionListener(this);  
jmiLow.addActionListener(this);  
jmiReset.addActionListener(this);  
jmiAbout.addActionListener(this);  
  
// Adiciona ouvintes para a opção de cores adicionais.  
jmiMoreLess.addActionListener(this);  
jmiYellow.addActionListener(this);  
jmiPurple.addActionListener(this);  
jmiOrange.addActionListener(this);  
  
// Adiciona o rótulo ao painel de conteúdo.  
jfrm.add(jlab);  
  
// Adiciona a barra de menus ao quadro.  
jfrm.setJMenuBar(jmb);  
  
// Exibe o quadro.  
jfrm.setVisible(true);  
}  
  
// Trata eventos de ação dos itens de menu.  
public void actionPerformed(ActionEvent ae) {  
    // Obtém o comando de ação da seleção feita no menu.  
    String comStr = ae.getActionCommand();  
  
    // Se o usuário selecionar Exit, encerra o programa.  
    if(comStr.equals("Exit"))  
        System.exit(0);  
    else if(comStr.equals("More Colors")) {  
        jmColors.add(jmiYellow);  
        jmColors.add(jmiPurple);  
        jmColors.add(jmiOrange);  
        JMenuItem mi = (JMenuItem) ae.getSource();
```

```

        mi.setText("Fewer Colors");
    } else if(comStr.equals("Fewer Colors")) {
        jmColors.remove(jmiYellow);
        jmColors.remove(jmiPurple);
        jmColors.remove(jmiOrange);
        JMenuItem mi = (JMenuItem) ae.getSource();
        mi.setText("More Colors");
    }

    // Caso contrário, exibe a opção selecionada.
    jlab.setText(comStr + " Selected");
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new DynMenuDemo();
        }
    });
}
}

```

uma dica de ferramenta a um item de menu. Por exemplo, esta linha cria uma dica de ferramenta para o item About:

```
| jmiAbout.setToolTipText("Info about the MenuDemo program.");
```

Verificação do progresso

1. Tanto **setMnemonic()** quanto **setAccelerator()** recebem objetos de tipo **KeyEvent** como argumento. Verdadeiro ou falso?
2. Qual é a máscara da tecla CTRL?
3. Que método define o texto da dica de ferramenta de um item de menu?

Pergunte ao especialista

P Você mencionou que uma dica de ferramenta pode ser adicionada a qualquer componente leve. Ou seja, posso adicionar dicas de ferramenta a controles como botões, listas e árvores, por exemplo?

R Sim. Se quiser, faça testes adicionando dicas de ferramentas a alguns dos exemplos mostrados no capítulo anterior. É só usar a mesma abordagem mostrada para os itens de menu: chame **setToolTipText()** no controle.

Respostas:

1. Falso, **setAccelerator()** requer um objeto **KeyStroke**.
2. **InputEvent.CTRL_DOWN_MASK**
3. **setToolTipText()**

USE JRadioButtonMenuItem E JCheckBoxMenuItem

Embora os tipos de itens de menu descritos pelos exemplos anteriores sejam usados com frequência, o Swing define outros dois que você pode achar útil: caixas de seleção e botões de rádio. Esses itens podem otimizar uma GUI ao permitir que o menu forneça uma funcionalidade que, de outra forma, exigiria componentes autônomos adicionais. Além disso, às vezes a inclusão de caixas de seleção ou botões de rádio em um menu parece simplesmente o local mais natural para um conjunto específico de recursos.

Para adicionar uma caixa de seleção a um menu, crie um **JCheckBoxMenuItem**. Essa classe herda **JMenuItem**. **JCheckBoxMenuItem** funciona como uma caixa de seleção autônoma. Por exemplo, ela gera eventos de ação e um evento de item quando seu estado muda. As caixas de seleção são especialmente úteis nos menus quando temos opções que podem ser selecionadas e queremos exibir seu status de marcada/desmarcada.

JCheckBoxMenuItem tem vários construtores. Os dois usados aqui são:

```
JCheckBoxMenuItem(String nome)  
JCheckBoxMenuItem(String nome, boolean estado)
```

O primeiro cria um item de menu de caixa de seleção com o nome especificado por *nome*. A caixa vem desmarcada. O segundo cria um item de menu de caixa de seleção com o nome e o estado de seleção especificados. Se *estado* for igual a **true**, inicialmente a caixa estará marcada; caso contrário, estará desmarcada.

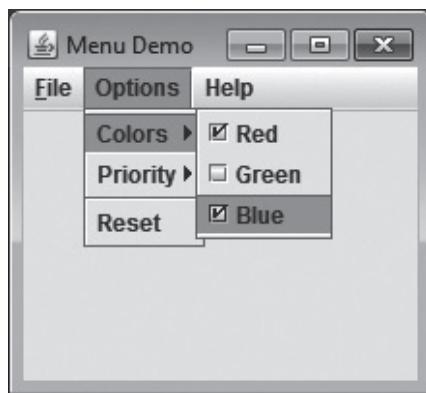
Um botão de rádio pode ser adicionado a um menu com a criação de um objeto de tipo **JRadioButtonMenuItem**. Ele também herda **JMenuItem**. **JRadioButtonMenuItem** funciona como um botão de rádio autônomo, gerando eventos de item e de ação. Como os botões de rádio autônomos, os botões de rádio baseados em menu devem ser inseridos em um grupo de botões para exibirem um comportamento de seleção mutuamente exclusivo.

JRadioButtonMenuItem também tem vários construtores. Esses são os dois usados aqui:

```
JRadioButtonMenuItem(String nome)  
JRadioButtonMenuItem(String nome, boolean estado)
```

O primeiro cria um item de menu de botão de rádio associado ao nome especificado por *nome*. O botão não vem pressionado. O segundo cria um item de menu de botão de rádio com o nome e o estado de seleção especificados. Se *estado* for igual a **true**, inicialmente o botão estará pressionado; caso contrário, não estará.

Para testar os itens de menu de caixa de seleção e de botão de rádio, primeiro remova o código que cria o menu Options do exemplo de programa **MenuDemo**. Depois, insira a sequência de código a seguir, que usa caixas de seleção para o submenu Colors e botões de rádio para o submenu Priority. Após a substituição, o menu Options ficará como o mostrado abaixo:



```

// Cria o menu Options.
JMenu jmOptions = new JMenu("Options");

// Cria o submenu Colors.
JMenu jmColors = new JMenu("Colors");

// Usa caixas de seleção para as cores. Isso permite
// que o usuário selecione mais de uma cor. Observe
// que inicialmente Red está selecionado.
JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red", true);
JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");           | Usa caixas de
                                                               | seleção para
                                                               | as cores.

jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// Cria o submenu Priority.
JMenu jmPriority = new JMenu("Priority");

// Usa botões de rádio para a definição da prioridade. Isso
// permite que o menu exiba que prioridade está sendo usada,
// mas também assegura que uma e somente uma prioridade
// possa ser selecionada em um determinado momento. Observe
// que inicialmente o botão de rádio High está pressionado.
JRadioButtonMenuItem jmiHigh = _____
    new JRadioButtonMenuItem("High", true);                         | Usa botões de rádio
JRadioButtonMenuItem jmiLow = _____
    new JRadioButtonMenuItem("Low");                                | para as prioridades.

jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

```

```

// Cria grupo de botões para os itens de menu de botão de rádio.
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);
```

Insere os botões de rádio do menu Priority em um grupo de botões.

```

// Cria o item de menu Reset.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Para finalizar, adiciona o menu Options inteiro
// à barra de menus
jmb.add(jmOptions);

```

Pergunte ao especialista

P Posso usar ícones com itens de menu de caixa de seleção e/ou botão de rádio?

R Sim. Por exemplo, você pode especificar um ícone para um item de botão de rádio usando este construtor:

`JRadioButtonMenuItem(Icon icone)`

E pode especificar um ícone para uma caixa de seleção usando esse:

`JCheckBoxMenuItem(Icon icone)`

Há outros construtores disponíveis que permitem a inclusão de texto e/ou a definição do estado de seleção.

Você também pode definir uma tecla mnemônica e/ou aceleradora para itens de menu de botão de rádio e caixa de seleção. Para fazê-lo, use os métodos `setMnemonic()` e `setAccelerator()`, já descritos.

Verificação do progresso

1. Que classe cria um item de menu de caixa de seleção?
2. Quando `JRadioButtonMenuItem`s são adicionados a um menu, eles passam automaticamente a fazer parte de um grupo de botões. Verdadeiro ou falso?

Respostas:

1. `JCheckBoxMenuItem`
2. Falso. Você deve inserir explicitamente itens de menu de botão de rádio em um grupo de botões.

Pergunte ao especialista

P Posso criar menus popup que sejam ativados com um clique no botão direito do mouse?

R Sim. Para criar um menu popup, é preciso usar **JPopupMenu**. Como você menciona em sua pergunta, normalmente um menu popup é ativado com um clique no botão direito do mouse quando o mouse está sobre o componente para o qual o menu foi definido. Para criar um menu popup, primeiro crie um objeto de tipo **JPopupMenu**. Em seguida, adicione **JMenuItem**s a ele. Você também terá que ouvir eventos do mouse (que são eventos de tipo **MouseEvent**) implementando a interface **MouseListener**, que é declarada em **java.awt.event**. Embora o processo não seja muito difícil, uma descrição completa não faz parte do escopo deste livro.

EXERCÍCIOS

1. Quais são as principais classes de menu do Swing?
2. Que classe cria um menu? Para criarmos um menu principal, que classe usariámos?
3. Que evento é gerado quando um item de menu é selecionado?
4. Imagens não são permitidas em menus. Verdadeiro ou falso?
5. Que método adiciona uma barra de menus a uma janela?
6. Que método adiciona um mnemônico a um item de menu?
7. Um ícone pode ser usado como item de menu? Se puder, isso impede o uso de um nome?
8. Que classe cria um item de menu de botão de rádio?
9. Embora sejam permitidos itens de menu de caixa de seleção, seu uso é desencorajado porque eles dão uma aparência estranha ao menu. Verdadeiro ou falso?
10. No decorrer deste capítulo, foram sugeridas várias alterações no programa **MenuDemo** que demonstram recursos adicionais dos menus. Exceto pelos itens de menu dinâmicos da seção Tente isto 19-1, integre as outras alterações ao programa **MenuDemo** original. No processo, reorganize o programa para simplificá-lo usando métodos separados para construir os diversos menus.
11. Verdadeiro ou falso?
 - A. Você pode desativar um item de menu sem desativar o menu inteiro.
 - B. Você pode desativar um menu sem desativar todos os seus itens.
12. Como mencionado no capítulo, a classe **JMenu** tem um método **getMenuComponents()** que retorna um array contendo os componentes do menu. Suponhamos que um menu tivesse um ou mais separadores. Esses separadores seriam incluídos no array retornado por **getMenuComponents()**?
13. Suponhamos que você tivesse um menu **m** apenas com três partes: um item de menu Abrir, um separador e um item de menu Fechar. Se você executar **m.remove(1)**, será removido o separador ou o item de menu Fechar?

14. Escreva um programa que crie uma janela com uma barra de menus e um menu Arquivo. O menu Arquivo tem três itens: Abrir, Recente e Sair. O item Recente é um submenu que inicialmente não tem itens. Sempre que Abrir é selecionado, um novo item de menu é adicionado ao submenu Recente. O nome do novo item é “Abrir” seguido de um inteiro que é incrementado para cada novo item. Por exemplo, o primeiro item do submenu Recente seria Abrir1, o item seguinte seria Abrir2 e assim por diante. Quando o item de menu Sair é selecionado, o programa é encerrado. Nenhum ouvinte de ação é necessário para os itens do menu Recente.
15. Melhore o exercício anterior de modo que o submenu Recente exiba apenas os cinco itens de menu mais recentes adicionados.
16. Suponhamos que você tivesse um menu com dois itens.
 - A. Se os dois itens tiverem a mesma tecla mnemônica, o que acontecerá quando você pressionar a tecla e o menu estiver sendo exibido?
 - B. Se os dois itens tiverem o mesmo acelerador, o que acontecerá quando você pressionar essa tecla?
 - C. O que acontecerá se você tentar usar, para a tecla mnemônica ou para um item de menu, um caractere que não faça parte do texto do item de menu? Por exemplo, e se você tentasse usar **VK_A** como a tecla mnemônica do item de menu Abrir?
17. Crie um programa com um menu Ajuda composto por um item FAQ que abra um novo **JFrame** com pergunta e resposta. O fechamento do novo **JFrame** não deve fazer o aplicativo ser encerrado.
18. Crie um programa que exiba uma janela com quatro menus: Arquivo, Editar, Texto e Exibir. Adicione um item a cada um dos três primeiros menus e três itens ao menu Exibir: Arquivo, Editar e Texto. Todos os itens do menu Exibir devem ser itens de caixa de seleção e inicialmente devem estar marcados. Se algum dos itens do menu Exibir for desmarcado pelo usuário, o menu de mesmo nome deve ser desativado. Se posteriormente o item for marcado, o menu de mesmo nome deve ser ativado.
19. Crie um programa que exiba uma janela com um menu contendo um item e contendo um rótulo no centro. Adicione dicas de ferramenta apropriadas à barra de menus, ao menu, ao item de menu e ao rótulo.
20. Na seção do capítulo sobre mnemônicos e aceleradores, foi mencionado que uma ou mais teclas modificadoras podem ser especificadas como parte do acelerador. No entanto, os exemplos do capítulo só usaram uma tecla modificadora. Se você quiser que o acelerador use mais de uma tecla modificadora, terá que combinar as constantes da classe **InputEvent** correspondentes às teclas desejadas. Suponhamos que você tivesse um item de menu **m** e quisesse um acelerador para **m** composto pela letra ‘A’ em conjunto tanto com a tecla CTRL quanto com a tecla SHIFT. Que instrução Java definirá o acelerador para **m**?

20

Caixas de diálogo

PRINCIPAIS HABILIDADES E CONCEITOS

- **JOptionPane**
- **showMessageDialog()**
- **showConfirmDialog()**
- **showInputDialog()**
- **showOptionDialog()**
- **JDialog**
- Criar uma caixa de diálogo não modal
- **JFileChooser**

Embora os controles individuais fornecidos pelo Swing, como campos de texto, botões e listas, formem a base de qualquer GUI, você pode encontrar situações em que precise vincular dois ou mais desses controles como uma unidade para tratar operações de entrada mais sofisticadas. Por exemplo, se seu programa tiver que exibir uma mensagem de erro e oferecer opções (como Tentar novamente e Cancelar), você pode querer reuni-las na mesma unidade lógica e visual. Isso é feito com a criação de um *dialogo*.

Um diálogo é uma janela separada que solicita algum tipo de resposta ao usuário. Ele contém pelo menos uma mensagem e um botão, mas diálogos muito mais sofisticados são possíveis e comuns. Os diálogos também são chamados de *caixas de diálogo* e *janelas de diálogo*.

A caixa de diálogo fornece um meio pelo qual o programa pode atingir dois objetivos importantes. Em primeiro lugar, ela fornece uma maneira de organizar os componentes necessários a situações de entrada complexas que vão além do que os componentes básicos podem tratar individualmente. Por exemplo, um processador de texto usaria uma caixa de diálogo para permitir que o usuário selecionasse a fonte, o tamanho de ponto e o estilo do texto. Essa caixa de diálogo seria uma combinação de vários componentes individuais que, quando reunidos, definiriam um tipo de letra. Em segundo lugar, uma caixa de diálogo fornece ao programa uma maneira de ele solicitar ao usuário a entrada que precisa para prosseguir. Por exemplo, podemos usar uma caixa de diálogo para pedir uma senha ao usuário e esperar até ela ser inserida. Qualquer que seja o uso, as caixas de diálogo são parte importante de muitos aplicativos Swing.

O Swing dá amplo suporte às caixas de diálogo. As que examinaremos aqui são **JDialog**, **JOptionPane** e **JFileChooser**. A principal classe de caixas de diálogo é **JDialog**, mas, por estranho que pareça, ela não costuma ser usada diretamente. Em vez disso, muitas situações de diálogo podem ser tratadas com o uso de **JOptionPane**,

que fornece uma grande variedade de estilos de diálogo internos. **JFileChooser** é uma caixa de diálogo interna que permite que o usuário selecione um arquivo.

JOptionPane

No ambiente moderno de programação de GUIs, as caixas de diálogo são usadas de duas maneiras básicas. Na primeira, há caixas de diálogo que são maiores e mais complicadas e vinculam vários componentes de entrada. Um exemplo seria uma caixa de diálogo que permitisse ao usuário configurar uma conexão de modem. Ela poderia conter componentes que permitissem ao usuário especificar a velocidade e o protocolo do modem, ativar o controle de fluxo de hardware, especificar um comando de inicialização e assim por diante. A criação de uma caixa de diálogo tão sofisticada requer o uso de **JDialog**, que será discutida posteriormente neste capítulo.

O segundo tipo de caixa de diálogo é muito mais simples. É uma caixa de diálogo que interpela o usuário e espera uma resposta. Talvez o exemplo mais comum seja a caixa de diálogo “Sair? Sim/Não”, que apenas confirma se queremos realmente sair de um programa. Como essas caixas de diálogo relativamente simples formam uma parte tão importante da programação Java, o Swing dá amplo suporte interno a elas por intermédio da classe **JOptionPane**.

JOptionPane é uma classe de diálogo fácil de usar que oferece soluções para muitos problemas comuns relativos a diálogos. Ela dá suporte a quatro tipos básicos de caixas de diálogo:

- De mensagem
- De confirmação
- De entrada
- De opções

Uma caixa de diálogo de mensagem exibe uma mensagem e espera até o usuário pressionar o botão OK. Essa caixa de diálogo fornece uma maneira fácil e eficaz de assegurar que o usuário está ciente de algumas informações. Por exemplo, poderíamos usar uma caixa de diálogo de mensagem para informar ao usuário que uma conexão de rede foi perdida exibindo “Conexão perdida” dentro da caixa.

Uma caixa de diálogo de confirmação faz ao usuário uma pergunta que normalmente tem a resposta Sim/Não e então espera uma resposta. Essa caixa de diálogo é usada em casos em que um curso de ação tem que ser confirmado. Por exemplo, uma caixa de diálogo de confirmação que exibisse a mensagem “Sair sem salvar alterações?” poderia ser usada para verificar se o usuário deseja *realmente* sair de um programa sem salvar as alterações.

Uma caixa de diálogo de entrada permite que o usuário insira um string ou selecione um item em uma lista. A vantagem dessa caixa de diálogo é que ela permite que os usuários respondam inserindo o string que quiserem. Logo, ela vai além de uma simples resposta Sim/Não/OK/Cancelar. Você poderia usar esse tipo de caixa de diálogo para obter o URL de algum recurso, por exemplo.

Uma caixa de diálogo de opções permite a especificação de uma lista de opções para o usuário selecionar. Logo, ela nos permite criar um diálogo com opções que não estão disponíveis nas outras caixas de diálogo.

O interessante é que **JOptionPane** não é derivada de **JDialog**. Em vez disso, é um contêiner para os componentes que serão usados pela caixa de diálogo. Porém,

usa **JDialog**. **JOptionPane** constrói um objeto **JDialog** automaticamente e se adiciona a esse objeto. Em seguida, trata os detalhes de exibição da caixa de diálogo, obtenção da resposta e fechamento da caixa. Na verdade, **JOptionPane** fornece uma maneira otimizada de criação e gerenciamento de caixas de diálogo simples.

Todas as caixas de diálogo criadas por **JOptionPane** são *modais*. Uma caixa de diálogo modal demanda uma resposta antes de o programa continuar. Como resultado, você não pode redirecionar o foco para outra parte do aplicativo sem antes fechar a caixa de diálogo. Logo, uma caixa de diálogo modal interrompe o programa até o usuário responder. Embora as caixas de diálogo modais sejam muito comuns, você também pode criar caixas de diálogo *não modais* (também chamadas de *modeless*). Uma caixa de diálogo não modal *não* impede que outras partes do programa sejam usadas. Portanto, o resto do programa permanece ativo e o foco pode ser redirecionado para outras janelas. Mas você não pode criar uma caixa de diálogo não modal usando **JOptionPane**. (No entanto, caixas de diálogo não modais são facilmente criadas por **JDialog**, como será descrito neste capítulo.)

Embora possamos criar um **JOptionPane** usando um de seus construtores, normalmente não é o que se faz. Em vez disso, geralmente usamos um de seus métodos factory **show**. Esses métodos constroem automaticamente uma caixa de diálogo com um dos quatro estilos e retornam a resposta do usuário. Por exemplo, para criar uma simples caixa de diálogo de mensagem, podemos usar o método **showMessageDialog()**. Ele cria uma caixa de diálogo que exibe uma mensagem e espera até o usuário clicar no botão OK. Como veremos, os métodos factory tornam **JOptionPane** muito fácil de usar.

JOptionPane dá suporte a duas categorias básicas de métodos **show**. A primeira cria uma caixa de diálogo que usa **JDialog** para conter o diálogo. Esse é o tipo de **JOptionPane** mais usado e é o único descrito neste livro. A segunda categoria usa um **JInternalFrame** para conter o diálogo. Esse tipo de caixa de diálogo é muito menos comum e o tópico dos quadros internos não faz parte do escopo deste livro.

JOptionPane define os quatro métodos factory a seguir que criam caixas de diálogo padrão baseadas em **JDialog**: **showConfirmDialog()**, **showInputDialog()**, **showMessageDialog()** e **showOptionDialog()**. Cada um cria o tipo de caixa de diálogo sugerido por seu nome. São todos métodos estáticos. Os três primeiros têm várias formas sobrecarregadas. O restante dessa discussão examinará cada tipo de caixa de diálogo separadamente.

Verificação do progresso

1. Quais são os quatro tipos de caixas de diálogo que podem ser criados por **JOptionPane**?
2. Qual é a diferença entre caixas de diálogo modais e não modais?
3. **JOptionPane** é derivada de **JDialog**?

Respostas:

1. De mensagem, de confirmação, de entrada e de opções.
2. Uma caixa de diálogo modal demanda uma resposta antes que outras partes do programa possam ser usadas. Uma caixa de diálogo não modal permite que o resto do programa permaneça ativo.
3. Não.

showMessageDialog()

O método **showMessageDialog()** cria a caixa de diálogo mais simples que pode ser construída. Ela exibe uma mensagem e espera até o usuário pressionar o botão OK. Apesar de sua simplicidade, **showMessageDialog()** tem três formas. Sua versão mais curta é mostrada abaixo:

```
static void showMessageDialog(Component pai, Object msg)
    throws HeadlessException
```

Aqui, *pai* especifica o componente em relação ao qual a caixa de diálogo será exibida. Se você passar **null** para esse argumento, a caixa de diálogo será exibida no centro da tela. A mensagem a ser exibida é passada em *msg*. Tecnicamente, não é preciso que seja um string. Por exemplo, você poderia passar um **JLabel**. No entanto, para caixas de diálogo simples, um string costuma ser usado. Quando a caixa de diálogo é exibida, um botão OK é incluído. A caixa de diálogo espera até o usuário pressionar OK. Logo, a chamada a **showMessageDialog()** não retornará até o usuário pressionar OK ou fechar a caixa de diálogo clicando na caixa Fechar. (Como mencionado, todas as caixas de diálogo criadas pelos métodos **show** são modais.) Ou seja, a thread chamador a espera até a chamada retornar. Já que há apenas uma opção para o usuário (o botão OK), não há necessidade de retornar uma resposta, e o tipo de retorno é **void**.

Observe que **showMessageDialog()** pode lançar uma **HeadlessException**. Essa exceção será lançada se você tentar exibir uma caixa de diálogo em um ambiente não interativo, como quando não há uma tela, mouse ou teclado conectado. Os métodos **show** em geral lançam uma **HeadlessException** quando se tenta exibir uma caixa de diálogo em um ambiente não interativo.

O programa a seguir mostra **showMessageDialog()** em ação. Ele exibe uma caixa de diálogo que informa ao usuário que há pouco espaço em disco.

```
// Uma demonstração muito simples de JOptionPane.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MsgDialogDemo {

    JLabel jlab;
    JButton jbtnShow;
    JFrame jfrm;

    MsgDialogDemo() {
        // Cria um contêiner JFrame.
        jfrm = new JFrame("Simple Message Dialog");

        // Especifica FlowLayout como gerenciador de layout.
        jfrm.setLayout(new FlowLayout());
    }

    public static void main(String[] args) {
        MsgDialogDemo demo = new MsgDialogDemo();
        demo.jbtnShow.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(demo.jfrm, "Disco com pouco espaço!");
            }
        });
    }
}
```

```

// Fornece um tamanho inicial para o quadro.
jfrm.setSize(400, 250);

// Encerra o programa quando o usuário fecha o aplicativo.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Cria um rótulo para exibir que a caixa de diálogo foi fechada.
jlab = new JLabel();

// Cria um botão que exibirá a caixa de diálogo.
jbtnShow = new JButton("Show Dialog");

// Adiciona um ouvinte de ação para o botão.
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Dialog Opened");

        // Cria uma caixa de diálogo que exibirá uma mensagem.
        JOptionPane.showMessageDialog(jfrm, ←
            "Disk space is low."); ← Cria uma caixa
                                         de diálogo de
                                         mensagem.

        // Essa instrução não será executada enquanto a
        // chamada a showMessageDialog() não retornar.
        jlab.setText("Dialog Closed");
    }
});

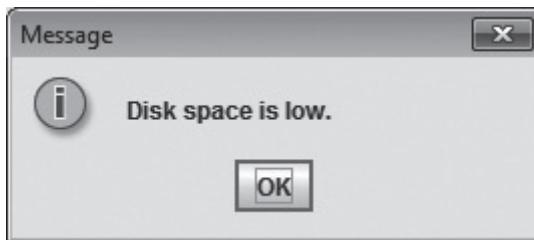
// Adiciona o botão e o rótulo ao painel de conteúdo.
jfrm.add(jbtnShow);
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MsgDialogDemo();
        }
    });
}
}

```

A caixa de diálogo produzida é mostrada abaixo:



Veja como o programa funciona. A janela principal contém um botão chamado Show Dialog. Quando esse botão é pressionado, o ouvinte de ações vinculado a ele exibe a caixa de diálogo chamando `showMessageDialog()`. O pai da caixa de diálogo é `jfrm`, que é a janela principal do programa. A mensagem a ser exibida é “Disk space is low”. Quando a chamada a `showMessageDialog()` é executada, a caixa de diálogo é exibida. Nesse momento, a caixa de diálogo recebe o foco de entrada e, já que ela é modal, o foco não pode ser redirecionado para a janela principal. Logo, a janela principal fica inativa até a caixa de diálogo ser fechada pelo usuário com um clique no botão OK ou na caixa Fechar. Quando a caixa de diálogo é fechada, a chamada a `showMessageDialog()` retorna e o texto de `jlab` é configurado para indicar esse fato.

Há duas outras formas de `showMessageDialog()` que nos permitem configurar com mais precisão vários aspectos da caixa de diálogo. A primeira é esta:

```
static void showMessageDialog(Component pai, Object msg,
                           String título, int tMsg)
                           throws HeadlessException
```

Os dois primeiros parâmetros são iguais aos da versão do método mostrada anteriormente. O parâmetro *título* permite a especificação de um título para a caixa de diálogo. Por padrão, o título é Message, mas é melhor especificar um título que represente com precisão a mensagem que você está exibindo. O parâmetro *tMsg* indica a natureza da mensagem. Ele deve ter um dos valores definidos por `JOptionPane`:

ERROR_MESSAGE	Indica que uma mensagem de erro será exibida. O ícone de erro padrão é usado.
INFORMATION_MESSAGE	Indica que uma mensagem informativa será exibida. O ícone de informação padrão é usado. Esse é o tipo de mensagem padrão.
PLAIN_MESSAGE	Indica uma mensagem “simples”, que é aquela em que nenhum ícone é exibido.
QUESTION_MESSAGE	Indica que uma mensagem interrogativa será exibida. O ícone de ponto de interrogação é usado.
WARNING_MESSAGE	Indica que uma mensagem de aviso será exibida. O ícone de aviso padrão é usado.

Como muitos outros aspectos do Swing, o efeito exato do parâmetro *tMsg* é determinado pela aparência da GUI.

Para ver os benefícios da especificação de um título e um tipo de mensagem, use a chamada a **showMessageDialog()** a seguir no exemplo anterior:

```
JOptionPane.showMessageDialog(jfrm,
                           "Disk Space is Low.",
                           "Warning",
                           JOptionPane.WARNING_MESSAGE);
```

Agora a caixa de diálogo de mensagem ficará assim:



Normalmente, uma caixa de diálogo de mensagem exibe um ícone padrão do sistema. Você pode especificar outro usando essa versão de **showMessageDialog()**:

```
static void showMessageDialog(Component pai, Object msg,
                           String título, int tMsg, Icon imagem)
                           throws HeadlessException
```

Aqui, o ícone a ser exibido é passado em *imagem*. Lembre-se, no entanto, de que se sua mensagem se enquadrar em uma das categorias predefinidas, pode ser melhor usar o ícone padrão em vez de um personalizado, porque o ícone padrão será mais facilmente reconhecido.

Verificação do progresso

1. Que tipo de resposta **showMessageDialog()** solicita?
2. Qual é a constante que indica uma mensagem de aviso?
3. A mensagem exibida por **showMessageDialog()** deve ser um string?

Respostas:

1. OK
2. **WARNING_MESSAGE**
3. Não, pode ser qualquer tipo de objeto.

showConfirmDialog()

Outro tipo de caixa de diálogo muito importante é o que solicita uma resposta básica Sim/Não ao usuário. No Swing, ela se chama caixa de diálogo de confirmação e é criada com uma chamada a `showConfirmDialog()`. Há várias versões desse método. A mais simples é mostrada abaixo:

```
static int showConfirmDialog(Component pai, Object msg)
                            throws HeadlessException
```

Aqui, `pai` especifica o componente em relação ao qual a caixa de diálogo será exibida. Se você passar `null` para esse argumento, a caixa de diálogo será exibida no centro da tela. A mensagem a ser exibida é passada em `msg`. Ela pode ser qualquer tipo de objeto, mas normalmente um string é usado. A caixa de diálogo contém automaticamente três botões chamados Yes, No e Cancel. O título da caixa é Select an Option.

O método retorna um valor inteiro que indica a seleção do usuário (isto é, que botão foi pressionado). O valor de retorno será uma destas constantes definidas por `JOptionPane`:

CANCEL_OPTION	Retornado se o usuário clicar em Cancel.
CLOSED_OPTION	Retornado se o usuário fechar a caixa de diálogo sem fazer uma seleção.
NO_OPTION	Retornado se o usuário clicar em No.
YES_OPTION	Retornado se o usuário clicar em Yes.

Vejamos `CLOSED_OPTION`. Esse valor é retornado quando o usuário fecha a caixa de diálogo (clicando na caixa Fechar) em vez de pressionar um dos botões. Normalmente, devemos tratar uma resposta `CLOSED_OPTION` como se ela representasse “Não” ou “Cancelar”, dependendo do contexto. `CLOSED_OPTION` nunca deve ser interpretada como uma resposta `YES_OPTION`.

O programa a seguir cria uma caixa de diálogo de confirmação simples e mostra como tratar as respostas.

```
// Usa uma caixa de diálogo de confirmação simples.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ConfirmDialogDemo {

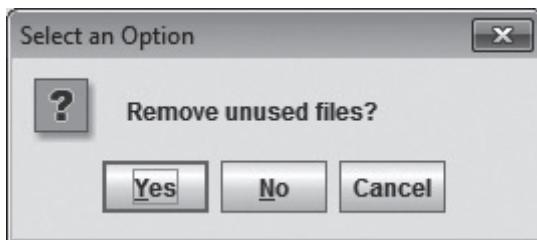
    JLabel jlab;
    JButton jbtnShow;
    JFrame jfrm;

    ConfirmDialogDemo() {
```

```
// Cria um contêiner JFrame.  
jfrm = new JFrame("A Confirmation Dialog");  
  
// Especifica FlowLayout como gerenciador de leiaute.  
jfrm.setLayout(new FlowLayout());  
  
// Fornece um tamanho inicial para o quadro.  
jfrm.setSize(400, 250);  
  
// Encerra o programa quando o usuário fecha o aplicativo.  
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
// Cria um rótulo para exibir a resposta do usuário.  
jlab = new JLabel();  
  
// Cria o botão que exibirá a caixa de diálogo.  
jbtnShow = new JButton("Show Dialog");  
  
// Adiciona um ouvinte de ação para o botão.  
jbtnShow.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent le) {  
        // Cria uma caixa de diálogo de confirmação.  
        int response = JOptionPane.showConfirmDialog( ← Cria uma caixa de  
                jfrm,                                            diálogo de confirmação.  
                "Remove unused files?");  
  
        // Exibe a resposta.  
        switch(response) { ← Verifica a resposta do usuário e responde apropriadamente.  
            case JOptionPane.YES_OPTION:  
                jlab.setText("You answered Yes.");  
                break;  
            case JOptionPane.NO_OPTION:  
                jlab.setText("You answered No.");  
                break;  
            case JOptionPane.CANCEL_OPTION:  
                jlab.setText("Cancel pressed.");  
                break;  
            case JOptionPane.CLOSED_OPTION:  
                jlab.setText("Dialog closed without response.");  
                break;  
        }  
    }  
});  
  
// Adiciona o botão e o rótulo ao painel de conteúdo.  
jfrm.add(jbtnShow);  
jfrm.add(jlab);
```

```
// Exibe o quadro.  
jfrm.setVisible(true);  
}  
  
public static void main(String[] args) {  
    // Cria a GUI na thread de despacho de evento.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new ConfirmDialogDemo();  
        }  
    });  
}
```

A caixa de diálogo produzida é mostrada aqui:



Como você pode ver, o código de tratamento da resposta do usuário é simples. Uma vez que todas as caixas de diálogo criadas com os métodos **show** de **JOptionPane** são modais, a chamada a **showConfirmDialog()** não retorna até o usuário selecionar uma opção ou fechar a janela de diálogo clicando em sua caixa Fechar. A resposta é retornada por **showConfirmDialog()** e atribuída a uma variável chamada **response**. Essa variável controla uma instrução **switch** que exibe a escolha do usuário.

Embora a forma anterior de **showConfirmDialog()** seja muito fácil de usar, ela pode não ser adequada em algumas aplicações porque sempre usa o título Select an Option. Em certos casos, um título mais descriptivo para a caixa de diálogo, como "Disk Space Is Low", pode ser melhor. Felizmente, é fácil alterar o título e outros aspectos da caixa de diálogo usando uma das versões sobrecarregadas de **showConfirmDialog()** mostradas a seguir:

```
static int showConfirmDialog(Component pai, Object msg,  
                           String título, int tOpç)  
                           throws HeadlessException  
  
static int showConfirmDialog(Component pai, Object msg,  
                           String título, int tOpç, int tMsg)  
                           throws HeadlessException  
  
static int showConfirmDialog(Component pai, Object msg,  
                           String título, int tOpç, int tMsg,  
                           Icon imagem)  
                           throws HeadlessException
```

Aqui, *pai* e *msg* continuam representando o que já descrevemos. O título da caixa de diálogo é especificado por *título*. As opções (isto é, os botões) que o usuário pode selecionar são especificadas por *tOpç*. Elas devem ser uma das constantes a seguir definidas por **JOptionPane**:

OK_CANCEL_OPTION	A caixa de diálogo inclui botões OK e Cancel.
YES_NO_OPTION	A caixa de diálogo inclui botões Yes e No.
YES_NO_CANCEL_OPTION	A caixa de diálogo inclui botões Yes, No e Cancel.

Por padrão, Yes, No e Cancel são fornecidos. No entanto, Cancel não é apropriado em todos os casos. Em situações em que só uma resposta Yes ou No for adequada, passe **YES_NO_OPTION** para o parâmetro *tOpç*. Se **OK_CANCEL_OPTION** for usada, a caixa de diálogo também poderá retornar o valor **OK_OPTION**.

O tipo geral da caixa de diálogo exibida é passado em *tMsg*. Ele deve ser uma das constantes a seguir, que já foram descritas:

ERROR_MESSAGE	INFORMATION_MESSAGE	PLAIN_MESSAGE
QUESTION_MESSAGE	WARNING_MESSAGE	

O tipo de mensagem padrão é **QUESTION_MESSAGE**.

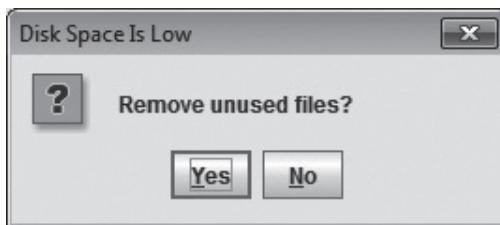
O ícone exibido dentro da caixa de diálogo é especificado por *imagem*. Por padrão, um ponto de interrogação é usado.

A caixa de diálogo exibida pelo exemplo anterior pode ser facilmente melhorada com a inclusão de um título e a remoção do botão Cancel, como mostrado nesta versão de **ActionListener** do exemplo anterior:

```
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Cria uma caixa de diálogo que exibe uma mensagem.
        int response = JOptionPane.showConfirmDialog(
            jfrm,
            "Remove unused files?", ← Esta é a pergunta.
            "Disk Space Is Low", ← Este é o título.
            JOptionPane.YES_NO_OPTION); ← Só exibe uma opção Yes e No.

        switch(response) {
            case JOptionPane.YES_OPTION:
                jlab.setText("You answered Yes.");
                break;
            case JOptionPane.NO_OPTION:
                jlab.setText("You answered No.");
                break;
            case JOptionPane.CLOSED_OPTION:
                jlab.setText("Dialog closed without response.");
                break;
        }
    }
});
```

Após a substituição do código, a caixa de diálogo ficará assim:



Observe que agora o título é “Disk Space Is Low” e que o botão Cancel foi removido.

Verificação do progresso

1. Por padrão, `showConfirmDialog()` solicita uma resposta Yes/No/Cancel. Verdadeiro ou falso?
2. Que efeito produz a passagem de `YES_NO_OPTION` para o parâmetro *tOpç* de `showConfirmDialog()`?

showInputDialog()

Embora uma resposta Yes/No seja adequada para algumas caixas de diálogo simples, com frequência entradas mais flexíveis são requeridas. Para tratar esses casos, **JOptionPane** fornece dois outros tipos de caixas de diálogo. Um é criado pelo método `showOptionDialog()`, que será descrito na próxima seção. O outro é `showInputDialog()`, que será descrito agora.

O método `showInputDialog()` dá suporte a várias formas. A mais simples exibe um campo de texto em que o usuário pode inserir um string. Essa versão é mostrada abaixo:

```
static String showInputDialog(Object msg) throws HeadlessException
```

Aqui, a mensagem a ser exibida é passada em *msg*. Como regra geral, a caixa de diálogo é centralizada na tela porque nenhuma janela pai é especificada. O método retorna o string inserido pelo usuário. A caixa de diálogo exibe os botões OK e Cancel. O pressionamento de OK faz o string ser retornado. O pressionamento de Cancel (ou um clique no botão Fechar da janela) faz a caixa de diálogo descartar qualquer string inserido pelo usuário e retornar **null**. O pressionamento de OK sem que nenhum string tenha sido inserido faz um string de tamanho zero ser retornado.

Respostas:

1. Verdadeiro.
2. Faz apenas as opções Yes e No serem exibidas.

Veja um exemplo que demonstra a forma mais simples de **ShowInputDialog()**. Ele apenas pede um nome.

```
// Uma caixa de diálogo de entrada simples.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class InputDialogDemo {

    JLabel jlab;
    JButton jbtnShow;
    JFrame jfrm;

    InputDialogDemo() {
        // Cria um contêiner JFrame.
        jfrm = new JFrame("A Simple Input Dialog");

        // Especifica FlowLayout como gerenciador de layout.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(400, 250);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria um rótulo que exibe a resposta.
        jlab = new JLabel();

        // Cria um botão que exibirá a caixa de diálogo.
        jbtnShow = new JButton("Show Dialog");

        // Adiciona um ouvinte de ação para o botão.
        jbtnShow.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {
                // Cria uma caixa de diálogo para receber o string.
                String response = JOptionPane.showInputDialog( ← Cria uma caixa de
                                                                "Enter Name"); diálogo de entrada
                                                                que lê um string.

                // Se a resposta for null, a caixa de diálogo
                // foi cancelada ou fechada. Se a resposta for
                // um string de tamanho zero, nenhuma entrada
                // foi inserida. Caso contrário, a resposta
            }
        });
    }
}
```

```

// conterá um string inserido pelo usuário.
if(response == null) ← Uma resposta null
    jlab.setText("Dialog cancelled or closed");
else if(response.length() == 0)
    jlab.setText("No string entered");
else
    jlab.setText("Hi there " + response);
}

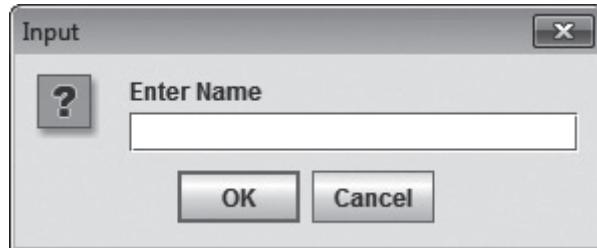
// Adiciona o botão e o rótulo ao painel de conteúdo.
jfrm.add(jbtnShow);
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new InputDialogDemo();
        }
    });
}
}

```

A caixa de diálogo produzida é mostrada aqui:



Embora a caixa de diálogo de entrada criada pelo programa anterior seja totalmente funcional, é muito limitada. Por exemplo, ela sempre usa o título “Input”, posiciona o diálogo no centro da tela e não dentro da janela do aplicativo e não permite o fornecimento de um valor inicial para o campo de texto. Felizmente, todas essas deficiências são fáceis de reparar com o uso de uma das formas sobrecarregadas de `showInputDialog()`, mostradas abaixo:

```

static String showInputDialog(Object msg, Object valInic)
    throws HeadlessException

```

```

static String showInputDialog(Component pai, Object msg)
    throws HeadlessException

static String showInputDialog(Component pai, Object msg, Object valInic)
    throws HeadlessException

static String showInputDialog(Component pai, Object msg, String título,
    int tMsg) throws HeadlessException

```

Aqui, *pai* especifica o componente em relação ao qual a caixa de diálogo será exibida. O valor inicial a ser inserido no campo de texto é passado via *valInic*. O valor passado em *título* especifica o título. O tipo de caixa de diálogo é passado em *tMsg*. Ele deve ser uma das constantes a seguir, as quais já foram descritas:

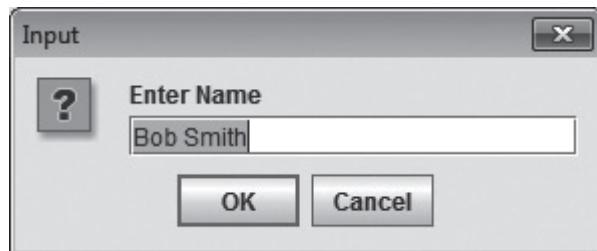
ERROR_MESSAGE	INFORMATION_MESSAGE	PLAIN_MESSAGE
QUESTION_MESSAGE	WARNING_MESSAGE	

O tipo de mensagem padrão é **QUESTION_MESSAGE**.

Por exemplo, a chamada a **showInputDialog()** a seguir cria uma caixa de diálogo que será posicionada em relação à janela principal e inicializada com o nome “Bob Smith”.

```
String response = JOptionPane.showInputDialog(
    jfrm, "Enter Name", "Bob Smith");
```

Para ver o efeito da alteração, insira essa chamada a **showInputDialog()** no programa anterior. Após a alteração, a janela de diálogo ficará assim:



Há mais uma forma de **showInputDialog()** que permite a especificação de uma lista com as opções que o usuário poderá selecionar e do ícone que será exibido. Essa forma é mostrada abaixo:

```

static Object showInputDialog(Component pai, Object msg,
    String título, int tMsg, Icon imagem,
    Object[ ] vals, Object valInic)
    throws HeadlessException

```

Aqui, *pai*, *título* e *tMsg* representam o que já descrevemos. O ícone a ser exibido é passado em *imagem*. Se *imagem* for **null**, o ícone padrão associado ao tipo de mensagem especificado será exibido. Um conjunto de opções é passado em *vals*. Essas opções serão exibidas em uma lista. No entanto, se *vals* for **null**, será exibido um campo de texto em que o usuário poderá inserir um valor. O valor inicial exibido é passado em *valInic*.

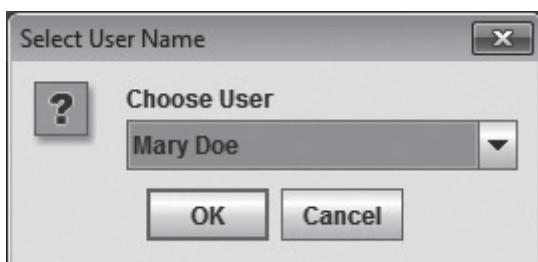
A caixa de diálogo criada por essa versão de **showInputDialog()** será particularmente útil se quisermos limitar o usuário a um intervalo de opções. Para ver esse tipo de caixa de diálogo de entrada em ação, substitua o ouvinte de ações do exemplo de programa pelo mostrado a seguir:

```
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        String[] names = { "Tom Jones", "Bob Smith",
                           "Mary Doe", "Nancy Oliver" };

        // Cria uma caixa de diálogo que permite que o
        // usuário faça uma escolha em uma lista de nomes.
        String response =
            (String) JOptionPane.showInputDialog( ← Cria uma caixa de diálogo
                                                jfrm,
                                                "Choose User",
                                                "Select User Name",
                                                JOptionPane.QUESTION_MESSAGE,
                                                null,
                                                names,
                                                "Bob Smith");

        if(response == null)
            jlab.setText("Dialog cancelled or closed");
        else if(response.length() == 0)
            jlab.setText("No string entered");
        else
            jlab.setText("Hi there " + response);
    }
});
```

Após essa alteração, a caixa de diálogo se parecerá com a mostrada abaixo:



Verificação do progresso

1. Quais são os dois tipos de entrada suportadas por `showInputDialog()`?
2. Você pode inicializar a resposta ao usar `showInputDialog()`?

`showOptionDialog()`

Embora as caixas de diálogo criadas por `showMessageDialog()`, `showConfirmDialog()` e `showInputDialog()` atendam muitas das necessidades comuns de diálogos, há situações para as quais elas não são apropriadas. Portanto, **JOptionPane** fornece mais um método `show`: `showOptionDialog()`. Esse método cria uma caixa de diálogo que conterá os elementos que você especificar. Logo, usando `showOptionDialog()` você pode criar uma caixa de diálogo direcionada às suas necessidades.

O método `showOptionDialog()` é mostrado abaixo:

```
static int showOptionDialog(Component pai, Object msg, String título,
                           int tOpç, int tMsg, Icon imagem,
                           Object[ ] opções, Object valInic)
                           throws HeadlessException
```

Você já conhece a maioria dos parâmetros. Os parâmetros *pai*, *msg* e *título* especificam o pai da caixa de diálogo (que pode ser **null** se o quadro padrão for usado), a mensagem exibida e o título da caixa. As opções (isto é, os botões) que o usuário pode selecionar são especificadas por *tOpç*. Elas devem ser uma das constantes a seguir definidas por **JOptionPane**:

```
DEFAULT_OPTION
OK_CANCEL_OPTION
YES_NO_OPTION
YES_NO_CANCEL_OPTION
```

É bom ressaltar, no entanto, que o parâmetro *tOpç* só será usado se o parâmetro *opções* for **null**. Caso contrário, ele será ignorado. Nesse caso, você pode usar a constante **DEFAULT_OPTION** como espaço reservado. O tipo geral da caixa de diálogo exibida é passado em *tMsg*. Ele deve ser uma das constantes a seguir, também descritas anteriormente:

ERROR_MESSAGE	INFORMATION_MESSAGE	PLAIN_MESSAGE
QUESTION_MESSAGE	WARNING_MESSAGE	

O ícone exibido dentro da caixa de diálogo é especificado por *imagem*. Para usar o ícone padrão, simplesmente passe **null** para esse parâmetro. A seleção inicial é passada por *valInic*.

Respostas:

1. Dependendo do que você solicitar, `showOptionDialog()` pode fazer o usuário inserir um string ou selecionar um item em uma lista.
2. Sim.

O único parâmetro novo é *opções*. Ele é um array de tipo **Object** que contém as opções que serão exibidas na caixa de diálogo. Normalmente, passamos um array de strings. Nesse caso, cada string passa a ser o nome de um botão. Quando pressionamos um botão, a caixa de diálogo é fechada e o índice do string é retornado. Também podemos passar um array de ícones ou um array contendo uma combinação de strings ou ícones. Quando um ícone é passado, ele é embutido automaticamente em um botão baseado em ícone. Outros tipos de objetos podem ser passados. (Consulte a caixa “Pergunte ao especialista” no fim desta seção.)

Veja um exemplo que usa **showOptionDialog()** para permitir que o usuário selecione como se conectará à rede.

```
// Uma caixa de diálogo de opções.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class OptionDialogDemo {

    JLabel jlab;
    JButton jbtnShow;
    JFrame jfrm;

    OptionDialogDemo() {
        // Cria um contêiner JFrame.
        jfrm = new JFrame("A Simple Option Dialog");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(400, 250);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria um rótulo que exibirá a seleção.
        jlab = new JLabel();

        // Cria um botão que exibirá a caixa de diálogo.
        jbtnShow = new JButton("Show Dialog");

        // Adiciona um ouvinte de ação para o botão.
        jbtnShow.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {

                // Define as opções de conexão.
                String[] connectOpts = { "Modem", "Wireless", ← Estas são as opções
                                         "Satellite", "Cable" };           que serão exibidas.
            }
        });
    }
}
```

```

// Cria uma caixa de diálogo que permite que o
// usuário selecione como se conectará à rede.
int response = JOptionPane.showOptionDialog( ← Cria uma caixa de
                                              jfrm,           diálogo de opções
                                              "Choose Network Connection",   que exibe uma
                                              "Connection Type",           lista de opções de
                                              JOptionPane.DEFAULT_OPTION,   conexão. Cada opção
                                              JOptionPane.QUESTION_MESSAGE, é exibida em seu
                                              null,                      próprio botão.
                                              connectOpts,
                                              "Wireless") ;

// Exibe a seleção.          A resposta contém o índice
switch(response) { ← da opção que foi clicada.
    case 0:
        jlab.setText("Connect via modem.");
        break;
    case 1:
        jlab.setText("Connect via wireless.");
        break;
    case 2:
        jlab.setText("Connect via satellite.");
        break;
    case 3:
        jlab.setText("Connect via cable.");
        break;
    case JOptionPane.CLOSED_OPTION:
        jlab.setText("Dialog cancelled.");
        break;
    }
}

// Adiciona o botão e o rótulo ao painel de conteúdo.
jfrm.add(jbtnShow);
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new OptionDialogDemo();
        }
    });
}
}

```

A caixa de diálogo produzida é mostrada aqui:



Examinando mais detalhadamente como o tratador `actionPerformed()` ativa a caixa de diálogo, primeiro observe o array `names` de tipo `String`, que especifica os nomes dos diversos métodos de conexão. Esse nome, junto com as outras informações, é passado para `showOptionDialog()`. Quando a caixa de diálogo é criada, os nomes são usados como os títulos dos botões. Quando um desses botões é pressionado, um inteiro correspondente ao índice do nome dentro do array é retornado. Portanto, o pressionamento de Modem retorna 0, o pressionamento de Wireless retorna 1, e assim por diante.

Na especificação das opções, é importante lembrar de uma coisa: elas sobrepõem as opções padrão. Logo, se você especificar uma opção “Cancelar”, o valor retornado quando ela for selecionada será seu índice no array e não `CANCEL_OPTION`. A única opção padrão que você encontrará é `CLOSED_OPTION`, porque ela é gerada quando o usuário clica na caixa Fechar da janela.

Embora `showOptionDialog()` ofereça uma flexibilidade significativa na determinação do que uma caixa de diálogo conterá, seu uso é um pouco mais limitado do que parece. Isso ocorre porque não temos nenhum controle real sobre o leiaute da caixa de diálogo. Em geral, todas as caixas de diálogo `show`, inclusive as criadas por `showOptionDialog()`, usam o mesmo padrão. Todas têm um título, uma mensagem de linha única, um ícone à esquerda da mensagem e apenas uma linha de opções. As caixas de diálogo de entrada adicionam um campo de texto ou uma lista. Não há como alterar esse leiaute.

Pergunte ao especialista

P Ao usar `showOptionDialog()`, posso passar algo que não sejam strings ou ícones para o parâmetro *opções*?

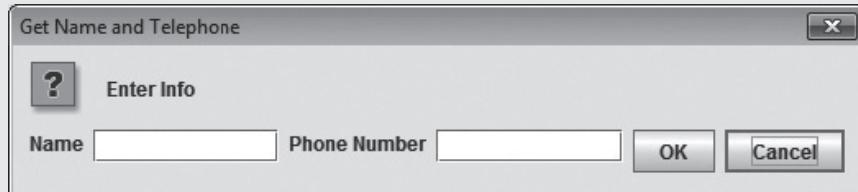
R Sim. Você pode passar qualquer tipo de objeto, inclusive outros componentes do Swing. Se passar um array de componentes, eles serão adicionados à caixa de diálogo. Logo, é possível construir uma caixa de diálogo contendo um campo de texto, duas caixas de seleção e um botão OK passando referências a esses componentes. Se você passar outro tipo de objeto, um botão será criado contendo os resultados da chamada a `toString()` no objeto.

Infelizmente, devido às limitações de layoute inerentes a **showOptionDialog()**, geralmente passamos strings ou ícones. Outros componentes costumam produzir resultados insatisfatórios. Por exemplo, se usarmos a sequência a seguir para construir uma caixa de diálogo:

```
Object[] ops = { new JLabel("Name"),
                 new JTextField(10),
                 new JLabel("Phone Number"),
                 new JTextField(10),
                 "OK", "Cancel" };

int response = JOptionPane.showOptionDialog(
    jfrm,
    "Enter Info",
    "Get Name and Telephone",
    JOptionPane.OK_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE,
    null,
    ops,
    "Cancel");
```

ela produzirá a saída abaixo:



Como você pode ver, o layoute está longe do ideal.

Verificação do progresso

1. **showOptionDialog()** é muito limitado porque suas opções não podem ser alteradas. Verdadeiro ou falso?
2. Por que **showOptionDialog()** não é tão útil como deveria?

Respostas:

1. Falso.
2. **showOptionDialog()** não é tão útil quanto o esperado porque seu formato básico é fixo e temos controle limitado sobre o layoute das opções.

JDialog

Embora **JOptionPane** ofereça a maneira mais fácil de exibir uma caixa de diálogo, ela não é aplicável a todas as situações. Quando você precisar de uma caixa de diálogo contendo mais campos ou que precise de tratamento especial, terá que usar **JDialog**. **JDialog** é a classe do Swing que cria uma caixa de diálogo. É um contêiner de nível superior que *não* é derivado de **JComponent**. Logo, é um componente pesado. Como explicado anteriormente, **JDialog** é a classe que **JOptionPane** usa para construir caixas de diálogo.

Em geral, criamos e gerenciamos um **JDialog** de maneira semelhante a como criamos um **JFrame**. Por exemplo, adicionamos componentes ao painel de conteúdo do **JDialog** como os adicionamos a um **JFrame**. Podemos definir o gerenciador de leiaute e especificar seu tamanho. Usamos **setVisible()** para exibir ou ocultar sua janela. Também podemos dar a ele uma barra de menus. **JDialog** herda várias classes do AWT: **Container**, **Component**, **Window** e **Dialog**. Logo, tem toda a funcionalidade oferecida pelo AWT.

JDialog permite a construção de uma caixa de diálogo modal ou não modal. Como explicado, uma caixa de diálogo modal faz o aplicativo pausar até a caixa ser fechada. Uma caixa de diálogo não modal permite que outras partes do aplicativo permaneçam ativas. A possibilidade de criar uma caixa de diálogo não modal é uma das razões para precisarmos criar um **JDialog** em vez de usar um dos métodos **show** de **JOptionPane**. Como veremos, é fácil construir os dois tipos de caixa de diálogo.

JDialog define muitos construtores. O que usaremos primeiro é mostrado aqui:

`JDialog(Frame pai, String título, boolean éModal)`

Ele cria uma caixa de diálogo cujo proprietário é especificado por *pai*. Se *éModal* for **true**, a caixa de diálogo será modal. Se for **false**, a caixa será não modal. A caixa de diálogo terá o título especificado por *título*.

Estas são as etapas que você seguirá para criar e exibir uma caixa de diálogo criada por **JDialog**:

1. Crie um objeto **JDialog**.
2. Especifique o gerenciador de leiaute, o tamanho e a política de fechamento padrão da caixa de diálogo.
3. Adicione componentes ao painel de conteúdo da caixa de diálogo.
4. Exiba a caixa de diálogo chamando **setVisible(true)** nela.

Para remover uma caixa de diálogo da tela, use **setVisible(false)** ou **dispose()**, que é herdado de **Window**. Use **setVisible(false)** quando for reutilizar a caixa de diálogo com frequência dentro do mesmo aplicativo. Use **dispose()** quando houver poucas chances da caixa de diálogo ser exibida novamente. A chamada a **dispose()** libera todos os recursos associados à caixa de diálogo. A chamada a **setVisible(false)** apenas remove a caixa da tela.

O programa a seguir mostra como criar uma caixa de diálogo modal simples com o uso de **JDialog**. A caixa de diálogo, que tem o título *Direction*, permite que o usuário selecione uma direção. Ela exibe dois botões. Um se chama Up e o outro Down. Quando um botão é pressionado, a caixa é fechada. A janela principal do

aplicativo contém dois botões e um rótulo. O rótulo exibe a direção atual. O botão chamado Show Dialog exibe a caixa de diálogo Direction. O botão chamado Reset Direction redefine a direção exibida no rótulo.

```
// Demonstra um JDialog simples.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JDialogDemo {

    JLabel jlab;

    JButton jbtnShow;
    JButton jbtnReset;

    // Esses botões ficam dentro da caixa de diálogo.
    JButton jbtnUp;
    JButton jbtnDown;

    JDialog jdlg;

    JDialogDemo() {
        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("JDialog Demo");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(400, 200);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria um rótulo que exibe a direção.
        jlab = new JLabel("Direction is pending.");

        // Cria um botão que exibirá a caixa de diálogo.
        jbtnShow = new JButton("Show Dialog");

        // Cria um botão que redefinirá a direção.
        jbtnReset = new JButton("Reset Direction");

        // Cria uma caixa de diálogo modal simples.
        jdlg = new JDialog(jfrm, "Direction", true);
        jdlg.setSize(200, 100);
        jdlg.setLayout(new FlowLayout());
```

Cria e configura uma caixa de diálogo modal.

```

// Cria os botões usados pela caixa de diálogo.
jbtnUp = new JButton("Up");
jbtnDown = new JButton("Down");

// Adiciona os botões à caixa de diálogo.
jdlg.add(jbtnUp); ← Adiciona botões e um rótulo à caixa de diálogo.
jdlg.add(jbtnDown); ←

// Adiciona um rótulo à caixa de diálogo.
jdlg.add(new JLabel("Press a button.")); ←

// Exibe a caixa de diálogo quando o botão Show Dialog é pressionado.
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jdlg.setVisible(true); ← Exibe a caixa de diálogo quando o usuário a solicita.
    }
});
```

// Redefine a direção quando o botão Reset Direction
// é pressionado.

```
jbtnReset.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Direction is pending.");
    }
});
```

// Responde ao botão Up da caixa de diálogo.

```
jbtnUp.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Direction is Up");

        // Oculta a caixa de diálogo após o usuário
        // selecionar uma direção.
        jdlg.setVisible(false); ←
    }
});
```

// Responde ao botão Down da caixa de diálogo.

```
jbtnDown.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Direction is Down");

        // Oculta a caixa de diálogo após o usuário
        // selecionar uma direção.
        jdlg.setVisible(false); ←
    }
});
```

Oculta a caixa de diálogo após o usuário selecionar uma direção.

```

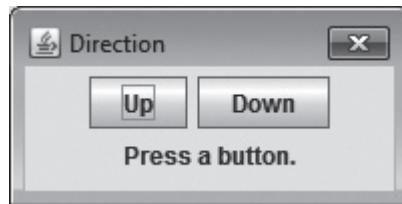
// Adiciona o botão Show Dialog e o rótulo ao painel de conteúdo.
jfrm.add(jbtnShow);
jfrm.add(jbtnReset);
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JDialo

```

A caixa de diálogo produzida pelo programa é mostrada aqui:



Há vários pontos importantes nesse programa. Para começar, observe como a sequência a seguir constrói um **JDialog** modal:

```

// Cria uma caixa de diálogo modal simples.
jdlg = new JDialo
jdlg.setSize(200, 100);
jdlg.setLayout(new FlowLayout());

```

Primeiro, uma caixa de diálogo modal chamada **jdlg** é criada, tendo **Direction** como seu nome, de propriedade da janela principal do aplicativo, **jfrm**, e que é modal porque **true** é passado como terceiro parâmetro do construtor de **JDialog**. Em seguida, seu tamanho é definido e ela recebe um gerenciador de layout de fluxo. Nesse ponto, a caixa de diálogo está totalmente construída, mas não contém nenhum componente e não pode ser vista.

Agora, dois botões e um rótulo são criados e adicionados a **jdlg**:

```
// Cria os botões usados pela caixa de diálogo.  
jbtnUp = new JButton("Up");  
jbtnDown = new JButton("Down");  
  
// Adiciona os botões à caixa de diálogo.  
jdlg.add(jbtnUp);  
jdlg.add(jbtnDown);  
  
// Adiciona um rótulo à caixa de diálogo.  
jdlg.add(new JLabel("Press a button."));
```

Como você pode ver, os componentes são adicionados à caixa de diálogo da mesma maneira como são adicionados à janela principal. Após essa sequência ser executada, a caixa de diálogo está totalmente formada, mas não pode ser vista. Ela não ficará visível até o botão Show Dialog (exibido na janela principal) ser pressionado. Para exibir a caixa de diálogo, o tratador do botão Show Dialog simplesmente chama **setVisible(true)** em **jdlg**.

Agora, observe os tratadores dos botões Up e Down que estão contidos dentro de **jdlg**. Eles apenas definem o texto de **jlab** e então chamam **setVisible(false)**, o que faz a caixa de diálogo ser removida da tela. Usar **setVisible(false)** é a maneira mais eficiente de ocultar uma caixa de diálogo que será necessária posteriormente.

CRIE UMA CAIXA DE DIÁLOGO NÃO MODAL

Usando **JDialog**, você achará muito fácil criar uma caixa de diálogo não modal (ou “modeless”), porque **JDialog** cria esse tipo de caixa por padrão. Por exemplo, você pode usar este construtor para criar uma caixa de diálogo não modal:

JDialog(Frame pai, String título)

Aqui, o proprietário é especificado por *pai*. A caixa de diálogo tem o título especificado por *título*. Como alternativa, você pode passar **false** explicitamente para o terceiro parâmetro do construtor mostrado. Além disso, muitas outras variações do construtor de **JDialog** estão disponíveis e permitem o fornecimento de outras opções.

A vantagem de uma caixa de diálogo não modal é que o resto do aplicativo permanece ativo. É claro que isso só é útil quando o resto do aplicativo não depende da entrada do usuário solicitada pela caixa de diálogo. Por exemplo, um aplicativo de retoque de fotos poderia ter uma caixa de diálogo que permitisse a seleção de vários filtros de retoque. Nesse caso, uma caixa de diálogo não modal seria a melhor opção porque permitiria que o usuário alterasse os filtros interativamente sem ter que constantemente fechar e reabrir a caixa. Em geral, a maioria das caixas de diálogo é modal, mas em casos em que uma caixa não modal pode ser usada, elas tendem a ser *muito úteis*.

Para ver os efeitos de uma caixa de diálogo não modal, você vai refazer o programa anterior convertendo a caixa de diálogo Direction em não modal. Antes de começar, execute o programa **JDialogDemo** e ative a caixa de diálogo Direction. Em seguida, tente clicar no botão Reset Direction na janela principal. Como você verá, já que a caixa de diálogo Direction é modal, não é possível acessar o botão Reset Direction enquanto a caixa estiver ativa. Com a caixa de diálogo Direction passando a ser

não modal, as duas janelas estarão ativas e você poderá redefinir a direção a qualquer momento.

A transformação da caixa de diálogo `Direction` em não modal requer poucas alterações. Primeiro, altere a chamada ao construtor de `JDialog` removendo o terceiro argumento, como mostrado aqui:

```
| jdlg = new JDialog(jfrm, "Direction");
```

Essa forma do construtor cria automaticamente uma caixa de diálogo não modal.

Em seguida, remova as chamadas a `setVisible(false)` que estão dentro dos tratadores de eventos dos botões Up e Down. Ou seja, agora esses dois tratadores ficarão assim:

```
// Responde ao botão Up da caixa de diálogo.
jbtnUp.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Direction is Up");
    }
});

// Responde ao botão Down da caixa de diálogo.
jbtnDown.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Direction is Down");
    }
});
```

Após essas alterações, a caixa de diálogo `Direction` permanecerá na tela até você clicar em sua caixa Fechar. Portanto, você poderá fazer repetidas mudanças na direção. Também poderá pressionar o botão Reset `Direction` na janela principal sem fechar a caixa de diálogo `Direction`.

Verificação do progresso

1. `JDialog` é derivada de `JComponent`?
2. Os componentes de um `JDialog` devem ser adicionados ao painel de conteúdo. Verdadeiro ou falso?
3. O que `dispose()` faz?

SELECIONE ARQUIVOS COM JFileChooser

Um dos usos mais comuns de uma caixa de diálogo também é um dos mais complicados e tediosos de implementar: permitir que o usuário selecione um arquivo. Ainda bem que o Swing fornece uma caixa de diálogo interna que trata essa tarefa um pouco complicada. Essa caixa de diálogo se chama *selecionador de arquivos* e é instância de `JFileChooser`.

Respostas:

1. Não.
2. Verdadeiro.
3. O método `dispose()` remove uma janela da tela, liberando todos os seus recursos no processo.

JFileChooser oferece dois benefícios importantes. O primeiro é a consistência. Selecionar um arquivo é uma atividade comum. **JFileChooser** assegura que todas as caixas de diálogo de seleção de arquivos tenham a mesma aparência. Assim, se os usuários souberem como usar um selecionador de arquivos, poderão usar todos. Isso é verdade até mesmo entre programas escritos por programadores diferentes. **JFileChooser** fornece um mecanismo padrão que os usuários entendem.

O segundo benefício de **JFileChooser** é a eficiência. Embora seja conceitualmente simples, implementar uma caixa de diálogo de seleção de arquivos requer um esforço de programação significativo. Ao fornecer uma implementação interna padrão, o Swing evita que os programadores tenham que fazer esse esforço repetidamente. Portanto, exceto para alguns aplicativos especializados, se você precisar de uma caixa de diálogo que permita que o usuário selecione um arquivo, deve usar **JFileChooser**. Isso evita desperdício de trabalho.

JFileChooser é derivada de **JComponent**. Ela especifica vários construtores, como estes três:

```
JFileChooser()
JFileChooser(File dir)
JFileChooser(String dir)
```

O primeiro cria um selecionador de arquivos que inicialmente exibe o diretório padrão. O segundo e o terceiro criam um selecionador de arquivos que inicialmente exibe o diretório especificado por *dir*. Se *dir* for **null**, o diretório padrão será usado.

Após você ter criado um **JFileChooser**, ele será exibido pela chamada a um dos métodos a seguir:

```
int showOpenDialog(Component pai) throws HeadlessException
int showSaveDialog(Component pai) throws HeadlessException
int showDialog(Component pai, String nome) throws HeadlessException
```

Em todos, *pai* é o componente em relação ao qual o selecionador de arquivos é posicionado. Quando *pai* é **null**, normalmente o selecionador de arquivos é centralizado na área de trabalho. O método **showOpenDialog()** exibe a caixa de diálogo padrão Open. O método **showSaveDialog()** exibe a caixa de diálogo padrão Save. A única diferença entre os dois é o título e o nome do botão que indica que um arquivo foi selecionado. Em **showOpenDialog()**, esse botão se chama Open. Em **showSaveDialog()**, ele se chama Save. Para especificar um título e um nome de botão escolhidos por você, chame **showDialog()** com o nome desejado. Esse método também cria um selecionador de arquivos padrão; só o título e o nome do botão são diferentes. Por exemplo, se você quisesse um selecionador de arquivos para selecionar um arquivo para exclusão, passaria “Excluir” para *nome*.

Todos os métodos retornam um inteiro que indica o resultado do processo de seleção de arquivo. Ele deve ser um desses valores definidos por **JFileChooser**:

APPROVE_OPTION	O usuário selecionou um arquivo.
CANCEL_OPTION	O usuário cancelou o processo de seleção clicando no botão Cancel ou na caixa Fechar.
ERROR_OPTION	Um erro foi encontrado.

Lembre-se, o usuário pode inserir qualquer nome de arquivo. Ele não tem que existir nem mesmo ser um nome válido. Portanto, o fato de um dos métodos **show** retornar **APPROVE_OPTION** não significa que o arquivo é válido.

Se um arquivo for selecionado, você pode conhecê-lo chamando o método **getSelectedFile()** na instância de **JFileChooser**, mostrado abaixo:

```
File getSelectedFile()
```

O método retorna um objeto **File** que representa o arquivo selecionado. (Lembre-se, esse arquivo não é aberto por **FileChooser**.)

Vimos no Capítulo 11 que a classe **File** encapsula informações sobre um arquivo. **File** fica no pacote **java.io** e contém vários métodos úteis. Por exemplo, para obter o nome do arquivo, chame o método **getName()** no objeto **File**. Ele é mostrado abaixo:

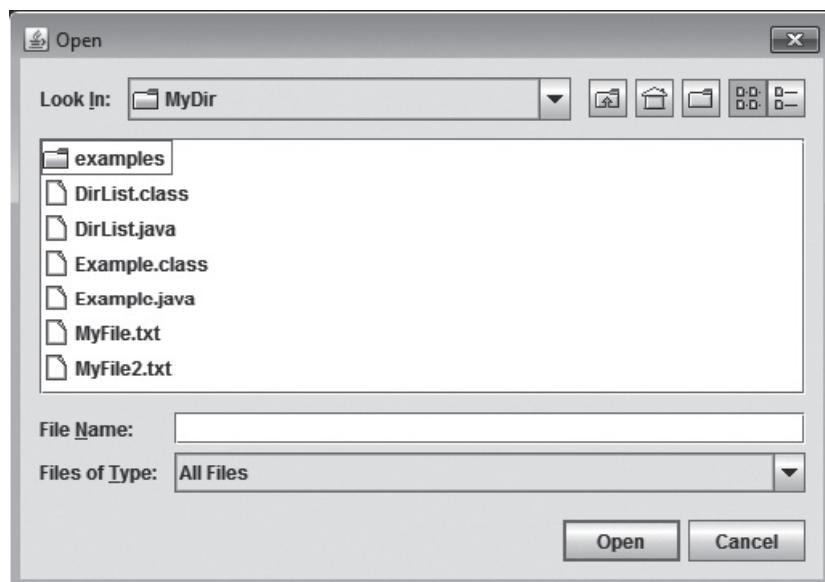
```
String getName()
```

O nome é retornado como um string. Dependendo das opções que você especificar, é possível o selecionador de arquivos permitir a seleção tanto de arquivos quanto de diretórios. (Diretórios são, na verdade, apenas tipos especiais de arquivos.) Você pode determinar se o objeto **File** retornado por **getSelectedFile()** é mesmo um arquivo chamando **isFile()** ou se é um diretório chamando **isDirectory()**. Esses métodos têm a assinatura a seguir:

```
boolean isFile()
boolean isDirectory()
```

Eles retornam **true** se o objeto for do tipo indicado; caso contrário, retornam **false**.

O próximo programa demonstra **JFileChooser**. Ele exibe uma caixa de diálogo Open inicializada com o diretório padrão. A caixa de diálogo de seleção de arquivos pode ser vista abaixo:



```
// Demonstra JFileChooser.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class FileChooserDemo {

    JLabel jlab;
    JButton jbtnShow;
    JFileChooser jfc;

    FileChooserDemo() {
        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("JFileChooser Demo");

        // Especifica FlowLayout como gerenciador de layout.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(400, 200);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria um rótulo que exibirá o arquivo selecionado.
        jlab = new JLabel();

        // Cria um botão que exibirá a caixa de diálogo.
        jbtnShow = new JButton("Show File Chooser");

        // Cria o selecionador de arquivos.          Cria um selecionador de arquivos
        jfc = new JFileChooser(); ← que começa no diretório padrão.

        // Exibe o selecionador de arquivos quando o botão
        // Show File Chooser é pressionado.
        jbtnShow.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {
                // Passa null para o pai. Isso centraliza a
                // caixa de diálogo na tela.
                int result = jfc.showOpenDialog(null); ← Exibe o selecionador de arquivos.

                if(result == JFileChooser.APPROVE_OPTION) ← Se um arquivo for
                    jlab.setText("Selected file is: " +           selecionado, exibe seu nome.
                                jfc.getSelectedFile().getName());
                else
                    jlab.setText("No file selected.");
            }
        });
    }
}
```

```
});  
  
// Adiciona o botão Show File Chooser e o rótulo  
// ao painel de conteúdo.  
jfrm.add(jbtnShow);  
jfrm.add(jlab);  
  
// Exibe o quadro.  
jfrm.setVisible(true);  
}  
  
public static void main(String[] args) {  
    // Cria a GUI na thread de despacho de evento.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new FileChooserDemo();  
        }  
    });  
}
```

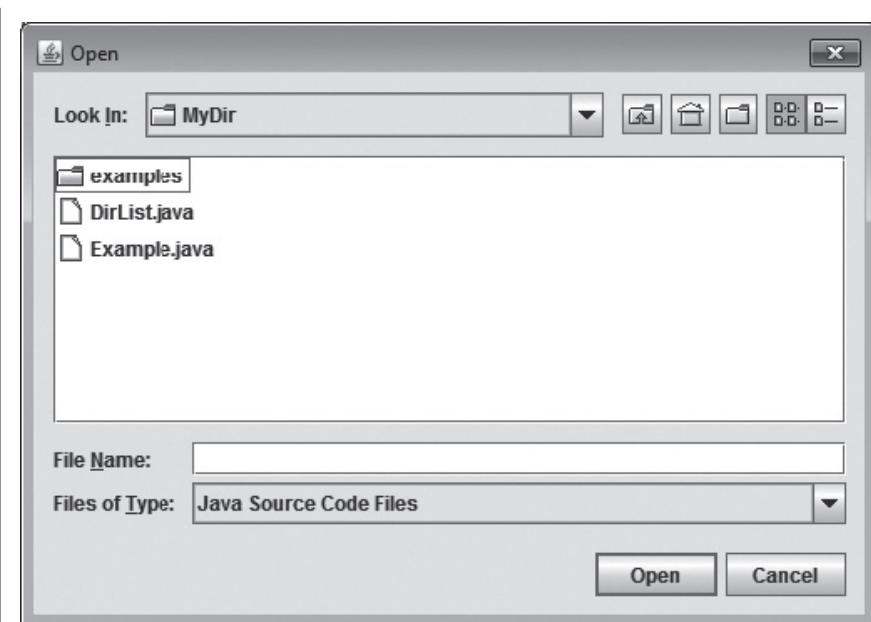
A operação do programa é simples. Quando ele começa, um **JFileChooser** chamado **jfc** é criado. Observe que não é especificado um diretório inicial, logo, o selecionador de arquivos usa o diretório padrão. Quando o botão Show File Chooser é pressionado, o tratador **actionPerformed()** exibe a caixa de diálogo de seleção de arquivos chamando **showOpenDialog()**. Quando o método retorna, se o valor de retorno for **APPROVE_OPTION**, o nome do arquivo selecionado é exibido em **jlab**. Caso contrário, é exibida uma mensagem indicando que nenhum arquivo foi selecionado. Outra coisa: com a construção do selecionador de arquivos fora do tratador **actionPerformed()**, ele pode ser usado repetidamente sem ter que ser reconstruído sempre que é chamado.

Antes de prosseguir, tente chamar `showSaveDialog()` em vez de `showOpenDialog()`. Como você verá, a única diferença é o nome.

TENTE ISTO 20-1 Use um filtro de arquivos com JFileChooser

FileFilterDemo.java

Por padrão, **JFileChooser** exibe todos os arquivos do diretório selecionado. Ao usar um **JFileChooser**, você pode alterar esse comportamento especificando um nome de arquivo que inclua caracteres curingas. Também pode alterar esse comportamento sob controle do programa usando um filtro de arquivos personalizado. Este projeto mostra como. Ele cria um filtro que exibe arquivos-fonte Java, os quais terminam em **.java**. Também exibe diretórios, o que permite que o usuário navegue no sistema de arquivos. Quando você executar o programa, verá algo semelhante à figura a seguir:



Observe que só arquivos-fonte Java são exibidos.

O filtro de arquivos de um **FileChooser** é um objeto que estende a classe abstrata **FileFilter** definida no pacote **javax.swing.filechooser**. A classe **FileFilter** define os dois métodos mostrados aqui:

```
abstract boolean accept(File arquivo)
abstract String getDescription()
```

Ambos devem ser implementados por seu filtro de arquivos personalizado. O método **accept()** deve retornar **true** para aceitar o arquivo passado via *arquivo*. Em outras palavras, se você quiser que o arquivo seja exibido na lista de arquivos, retorne **true**. Para impedir que o arquivo seja exibido, retorne **false**. O método **getDescription()** deve retornar um string que descreva o filtro. Ele será exibido na lista **Files of Type** do selecionador de arquivos.

Um ponto importante: quando você criar um filtro de arquivos personalizado, os diretórios não serão exibidos automaticamente. Se quiser que os diretórios sejam exibidos, você deve aceitá-los explicitamente dentro do método **accept()**.

Para permitir que um selecionador de arquivos use seu filtro, você pode chamar **setFileFilter()** no instância de **JFileChooser**. Esse método é mostrado abaixo:

```
void setFileFilter(FileFilter ff)
```

Aqui, *ff* é o filtro de arquivos que você quer que o selecionador de arquivos use.

PASSO A PASSO

- Comece criando um arquivo chamado **FileFilterDemo.java** e então insira o comentário e as instruções **import** a seguir:

```
// Tente isto 20-1: Demonstra um filtro de arquivos personalizado.

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
```

Observe como **javax.swing.filechooser.FileFilter** é importada. Seu nome é totalmente qualificado. Veja por quê. Como você aprendeu no Capítulo 11, Java também inclui uma interface chamada **FileFilter** que fica no pacote **java.io**. Ao criar um filtro de arquivos **FileChooser**, com frequência você terá que importar os dois pacotes (**java.io** e **javax.swing.filechooser**). Uma maneira de evitar um conflito de nomes é importar a versão selecionadora de arquivos de **FileFilter** usando seu nome totalmente qualificado, como mostrado acima. Essa abordagem evita o conflito de nomes entre os dois pacotes.

- Defina a classe **JavaFileFilter**, como mostrado a seguir. Ela só aceita arquivos com a extensão **.java** e diretórios.

```
// Um filtro de arquivos personalizado que exibe
// arquivos-fonte Java e diretórios.
class JavaFileFilter extends FileFilter {
    public boolean accept(File file) {
        // Retorna true se o arquivo for um arquivo-fonte
        // Java ou se for um diretório.
        if(file.getName().endsWith(".java")) return true;
        if(file.isDirectory()) return true;

        // Caso contrário, retorna false.
        return false;
    }

    public String getDescription() {
        return "Java Source Code Files";
    }
}
```

Observe como o método **accept()** funciona. Se um nome de arquivo usar a extensão “.java” ou for um diretório, ele retornará **true**. Em todos os outros casos, **accept()** retorna **false**. Ou seja, só arquivos **.java** e diretórios são exibidos. Repare o uso do método **endsWith()**. Esse método é definido pela classe **String** e retorna **true** se e somente se o string chamador terminar com a sequência de caracteres especificada.

3. Comece a classe **FileFilterDemo**, como mostrado abaixo:

```
public class FileFilterDemo {  
  
    JLabel jlab;  
    JButton jbtnShow;  
    JFileChooser jfc;  
  
    FileFilterDemo() {  
        // Cria um contêiner JFrame.  
        JFrame jfrm = new JFrame("File Filter Demo");  
  
        // Especifica FlowLayout como gerenciador de leiaute.  
        jfrm.setLayout(new FlowLayout());  
  
        // Fornece um tamanho inicial para o quadro.  
        jfrm.setSize(400, 200);  
  
        // Encerra o programa quando o usuário fecha o aplicativo.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Cria um rótulo que exibirá o arquivo selecionado.  
        jlab = new JLabel();  
  
        // Cria um botão que exibirá a caixa de diálogo.  
        jbtnShow = new JButton("Show File Chooser");  
  
        // Cria o selecionador de arquivos.  
        jfc = new JFileChooser();  
  
        // Configura o filtro de arquivos.  
        jfc.setFileFilter(new JavaFileFilter());
```

Preste atenção na última linha. Para o filtro de arquivos funcionar, ele deve ser configurado. Isso é feito com uma chamada a **setFileFilter()** em **jfc** imediatamente depois de ele ter sido criado.

4. Adicione o **ActionListener** mostrado aqui:

```
// Exibe o selecionador de arquivos quando o botão
// Show File Chooser é pressionado.
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Passa null para o pai. Normalmente isso centraliza a
        // caixa de diálogo na tela.
        int result = jfc.showOpenDialog(null);

        if(result == JFileChooser.APPROVE_OPTION)
            jlab.setText("Selected file is: " +
                        jfc.getSelectedFile().getName());
    }
})
```

```

        else
            jlab.setText("No file selected.");
    }
});
}

```

5. Termine o construtor de **FileFilterDemo** como mostrado a seguir:

```

// Adiciona o botão Show File Chooser e o rótulo
// ao painel de conteúdo.
jfrm.add(jbtnShow);
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

```

6. Termine **FileFilterDemo** adicionando o método **main()** abaixo:

```

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new FileFilterDemo();
        }
    });
}
}

```

7. O programa inteiro é mostrado abaixo. Antes de prosseguir, faça testes criando alguns filtros de arquivos projetados por você.

```

// Tente isto 20-1: Demonstra um filtro de arquivos personalizado.

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;

// Um filtro de arquivos personalizado que exibe
// arquivos-fonte Java e diretórios.
class JavaFileFilter extends FileFilter {
    public boolean accept(File file) {
        // Retorna true se o arquivo for um arquivo-fonte
        // Java ou se for um diretório.
        if(file.getName().endsWith(".java")) return true;
        if(file.isDirectory()) return true;

        // Caso contrário, retorna false.
        return false;
    }
}

```

```
    public String getDescription() {
        return "Java Source Code Files";
    }
}

public class FileFilterDemo {

    JLabel jlab;
    JButton jbtnShow;
    JFileChooser jfc;

    FileFilterDemo() {
        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("File Filter Demo");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(400, 200);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria um rótulo que exibirá o arquivo selecionado.
        jlab = new JLabel();

        // Cria um botão que exibirá a caixa de diálogo.
        jbtnShow = new JButton("Show File Chooser");

        // Cria o selecionador de arquivos.
        jfc = new JFileChooser();

        // Configura o filtro de arquivos.
        jfc.setFileFilter(new JavaFileFilter());

        // Exibe o selecionador de arquivos quando o botão
        // Show File Chooser é pressionado.
        jbtnShow.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {
                // Passa null para o pai. Normalmente isso
                // centraliza a caixa de diálogo na tela.
                int result = jfc.showOpenDialog(null);

                if(result == JFileChooser.APPROVE_OPTION)
                    jlab.setText("Selected file is: " +
                                jfc.getSelectedFile().getName());
                else

```

```
jlab.setText("No file selected.");
    }
});

// Adiciona o botão Show File Chooser e o
// rótulo ao painel de conteúdo.
jfrm.add(jbtnShow);
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Cria a GUI na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new FileFilterDemo();
        }
    });
}
```

Pergunte ao especialista

P Há outras opções suportadas por **JFileChooser**?

R Sim, **JFileChooser** dá suporte a várias opções. Aqui estão três que você pode achar interessantes. Por padrão, **JFileChooser** permite que o usuário selecione apenas arquivos. Para permitir que um diretório seja selecionado, chame o método **setFileSelectionMode()**, mostrado abaixo:

```
void setFileSelectionMode(int fsm)
```

Nele, *fsm* especifica o modo de seleção, que deve ser uma destas constantes definidas por **JFileChooser**:

FILES_ONLY
DIRECTORIES_ONLY
FILES AND DIRECTORIES

Cada constante especifica o modo indicado por seu nome. Portanto, para permitir a seleção tanto de arquivos quanto de diretórios, use **FILES AND DIRECTORIES**.

Você pode permitir que o usuário selecione mais de um arquivo chamando o método `setMultiSelectionEnabled()`, mostrado aqui:

```
void setMultiSelectionEnabled(boolean on)
```

Pergunte ao especialista (continuação)

Se *on* for **true**, será permitida a seleção de vários arquivos. Se for **false**, só poderão ser selecionados arquivos individuais. Por padrão, só a seleção de arquivos individuais é permitida. Quando usar o modo de seleção múltipla, você poderá obter uma lista dos arquivos selecionados chamando o método **getSelectedFiles()**, visto abaixo:

```
File[ ] getSelectedFiles()
```

Ele retorna um array de objetos **File** contendo os arquivos (ou diretórios) selecionados.

Por padrão, arquivos ocultos não são exibidos pelo selecionador de arquivos. Para mudar isso, chame o método **setFileHidingEnabled()**, mostrado a seguir:

```
void setFileHidingEnabled(boolean on)
```

Se *on* for **true**, os arquivos ocultos não serão exibidos. Se for **false**, os arquivos ocultos serão exibidos na janela de arquivos.

Verificação do progresso

1. Que método exibe o selecionador de arquivos Open?
2. Em relação a **JFileChooser**, que classe você estenderia para criar um filtro de arquivos?
3. Que método de **FileFilter** é usado para filtrar arquivos?

EXERCÍCIOS

1. Uma caixa de diálogo é uma combinação de dois ou mais componentes que interage com o usuário e espera uma resposta. Verdadeiro ou falso?
2. Que método de **JOptionPane** cria uma caixa de diálogo de entrada? E qual cria uma caixa de diálogo de mensagem?
3. Que método de **JOptionPane** você usaria para criar uma caixa de diálogo que confirmasse se o usuário deseja salvar alterações em um documento? Mostre como seria a chamada.
4. Com o uso de uma caixa de diálogo de confirmação, que tipo de retorno indica que o usuário clicou no botão Yes?
5. Que tipo de opção é usado para exibir só os botões Yes e No em uma caixa de diálogo de confirmação?

Respostas:

1. **showOpenDialog()**
2. **FileFilter**
3. **accept()**

6. Se você quiser solicitar uma resposta na forma de string ao usuário, que método de **JOptionPane** deve chamar?
7. O parâmetro de mensagem de qualquer um dos métodos **show** de **JOptionPane** deve ser um string? Explique.
8. **JDialog** é um contêiner de nível superior. Verdadeiro ou falso?
9. Quais são as quatro etapas necessárias à criação e exibição de uma caixa de diálogo baseada em **JDialog**?
10. **JDialog** pode criar uma caixa de diálogo não modal?
11. Explique a diferença entre **setVisible(false)** e **dispose()** no âmbito das caixas de diálogo.
12. Que método de **JFileChooser** cria um selecionador de arquivos Save? E qual criaria um selecionador de arquivos para usar um título que você escolhesse?
13. Quais são os dois métodos que devem ser sobrepostos na implementação de um **FileFilter** para **JFileChooser**?
14. O Capítulo 19 descreveu os menus. Neste capítulo, os exemplos incluíram um menu File que tinha uma entrada Exit. O tratador de eventos de ação que processava seleções no menu tratava a entrada Exit, como mostrado aqui:

```
// Trata eventos de ação dos itens de menu.
public void actionPerformed(ActionEvent ae) {
    // Obtém o comando de ação da seleção feita no menu.
    String comStr = ae.getActionCommand();

    // Se o usuário selecionar Exit, encerra o programa.
    if(comStr.equals("Exit")) System.exit(0);

    .
    .
    .
}
```

Altere esse código de modo que ele ative uma caixa de diálogo para confirmar se o usuário deseja realmente sair antes de encerrar o programa.

15. Por que usar uma caixa de diálogo modal e por que usar uma não modal? Dê exemplos diferentes dos deste capítulo em que cada uma delas seja útil.
16. Escreva um programa que crie uma janela com um botão chamado “Exibir mensagem”. Quando o botão for clicado, uma caixa de diálogo aparecerá perguntando “Deseja realmente ver a mensagem?” e, se você clicar em Sim, uma segunda caixa de diálogo deve aparecer com a mensagem.
17. Dê exemplos de situações em que seria relevante criar uma caixa de diálogo de mensagem de cada um dos tipos a seguir:
 - A. ERROR_MESSAGE
 - B. INFORMATION_MESSAGE
 - C. PLAIN_MESSAGE
 - D. QUESTION_MESSAGE
 - E. WARNING_MESSAGE

18. Dê exemplos de situações em que seria relevante criar uma caixa de diálogo de confirmação com cada uma das opções de botões a seguir:
 - A. OK_CANCEL_OPTION
 - B. YES_NO_OPTION
 - C. YES_NO_CANCEL_OPTION
19. Quando chamar o método `showOptionDialog()` de **JOptionPane**, você será solicitado a fornecer um array de opções e um valor inicial como os dois últimos argumentos. O que aconteceria se, para o valor inicial, você fornecesse `null` ou um valor que não estivesse no array de opções?
20. O que você deve usar como argumentos do método `showOptionDialog()` para que a caixa de diálogo surja e aja o máximo possível como uma caixa de diálogo obtida com uma chamada a `JOptionPane.showMessageDialog(null, "Hello", "Message", JOptionPane.PLAIN_MESSAGE)`?
21. Escreva um programa que crie e exiba um **JDialog** que surja e aja o máximo possível como a caixa de diálogo obtida com uma chamada a `JOptionPane.showMessageDialog(null, "Hello", "Hello message", JOptionPane.PLAIN_MESSAGE)`.
22. Como os dois exercícios anteriores mostram, para exibir uma mensagem em uma caixa de diálogo, você pode usar `showMessageDialog()` ou `showOptionDialog()`, ou pode criar um **JDialog**. Por que Java fornece tantas maneiras de fazer a mesma coisa? Especificamente, você pode usar **JDialog**, que faz tudo o que os métodos `show` fazem, então, por que se preocupar com eles?
23. Escreva um programa que crie uma janela chamada “Copiador de Arquivos”. Ela contém um botão que exibe “Selecionar o arquivo a ser copiado”. Quando o botão é clicado, aparece uma caixa de diálogo Abrir para a seleção de arquivos. Quando um arquivo é selecionado, aparece uma caixa Salvar de seleção de arquivos permitindo que o usuário especifique o nome a ser usado na cópia e o diretório que a conterá. Em seguida, o arquivo é copiado. Quando ocorre uma **IOException**, ela não exibe uma mensagem de erro no console. Em vez disso, exibe a mensagem de erro em uma caixa de diálogo.
24. Se você criar uma caixa de diálogo **JFileChooser** usando o construtor que não recebe argumentos, e então exibir a caixa, inicialmente ela conterá o diretório padrão. O que é o diretório “padrão”?
25. Modifique a classe **FileChooserDemo** discutida no capítulo para que a caixa de diálogo **JFileChooser** permita seleções múltiplas (como descrito na seção Pergunte ao especialista do fim do capítulo) tanto de arquivos quanto de diretórios. O programa deve então exibir todos os arquivos e diretórios selecionados.
26. Se você chamar `showMessageDialog(null, obj)`, em que `obj` é um objeto que não é um string, a caixa de diálogo exibirá o string retornado por uma chamada a `obj.toString()`. Para ver isso, crie uma classe **Data** contendo apenas um método `public String toString()` que retorne o string “Havia um objeto Data aqui”. Em seguida, escreva um programa que crie um objeto `obj` da classe **Data** e chame `showMessageDialog(null, obj)`.

27. Crie um programa que exiba uma janela com um botão. Quando o usuário clica no botão, uma caixa de diálogo de entrada aparece pedindo a ele um número inteiro de polegadas. Quando a entrada é um inteiro válido, o programa exibe uma caixa de diálogo informativa contendo quantas milhas são iguais ao número de polegadas fornecido. Quando a entrada não é um inteiro válido, uma caixa de diálogo de mensagem aparece. Use o método `parseInt()` discutido no Capítulo 11.
28. Crie uma classe personalizada que estenda `javax.swing.JFileChooser.FileFilter` e deixe de fora todos os arquivos, exceto diretórios e arquivos ocultos. Use o método `isHidden()` da classe `File` mencionado no Capítulo 11. Em seguida, crie um programa que teste seu novo filtro.

21

Threads, applets e geração de componentes

PRINCIPAIS HABILIDADES E CONCEITOS

- Usar threads com Swing
- Usar a classe **Timer**
- Conhecer os aspectos básicos de applets Swing
- Entender a estrutura de applets Swing
- Construir uma GUI de applet
- Saber os aspectos básicos da geração de componentes
- Usar o contexto gráfico
- Calcular a área de desenho

Como os capítulos anteriores mostraram, quando pensamos no Swing, geralmente consideramos seus controles visuais. No entanto, há outros aspectos e técnicas relacionados ao Swing que não envolvem diretamente seus controles. Este capítulo examinará três importantes aspectos: threads, applets e geração de componentes. Como veremos, todos eles precisam de tratamento especial quando usados com uma GUI do Swing.

O USO DE VÁRIAS THREADS EM SWING

Como vimos no Capítulo 12, o uso de várias threads é o aspecto de Java que permite que diferentes partes de um programa sejam executadas simultaneamente. Ele nos dá oportunidade de criar programas muito eficientes que fazem uso pleno da CPU empregando um tempo que de outra maneira seria ocioso, ou, no caso de várias CPUs, permite que partes do programa sejam executadas de forma realmente concorrente, o que otimiza a taxa de transferência. O uso de várias threads também impede que o aplicativo inteiro pause quando alguma parte dele está executando uma tarefa demorada. É essa segunda razão que torna o ambiente *multithread* especialmente importante para um aplicativo Swing. No Swing, operações demoradas devem ser executadas em sua própria thread para impedir que o aplicativo (inclusive sua interface de usuário) fique lento ou não responda. Portanto, o uso de várias threads faz parte de muitos programas Swing.

No geral, você usará threads adicionais dentro de um programa Swing da mesma maneira como os usa em qualquer outro tipo de programa Java. Logo, o que já sabe sobre a criação e gerenciamento de threads também se aplica ao Swing. No entanto, há outra questão muito importante relativa às threads que se refere especificamente ao Swing. Ela é o assunto desta seção.

No Capítulo 17, você aprendeu que qualquer código que interaja com um componente visual deve ser executado na thread de despacho de evento. Observar essa importante regra evita problemas, como duas threads diferentes tentando atualizar o mesmo componente ao mesmo tempo. É por causa dessa regra que **invokeLater()** é chamado dentro de **main()** por todos os programas Swing mostrados anteriormente para construir e exibir a GUI na inicialização do programa. Ele faz a GUI ser criada na thread de despacho de evento.

É importante enfatizar que essa mesma regra se aplica a *qualquer momento* em que você precisar atualizar, mudar, interrogar ou alterar um componente. Se um fragmento de código afetar um componente, ele *deve ser executado* a partir da thread de despacho de evento. Já que os tratadores de eventos são executados automaticamente na thread de despacho de evento, o código de um tratador pode afetar livremente a GUI. No entanto, um código que estiver sendo executado em outra thread não pode. Isso faz surgir um possível problema: com frequência, códigos de outras threads têm que atualizar um componente. Por exemplo, considere um programa que use um **JLabel** para exibir a temperatura externa, atualizada a cada minuto. Para fazê-lo, uma thread separada deve ser usada para monitorar a temperatura. Porém, essa thread não pode ser usada para atualizar o rótulo porque não é a thread de despacho de evento. Então, como a thread que monitora a temperatura atualizará o rótulo? Ou, generalizando, como qualquer outra thread atualizaria um componente da GUI?

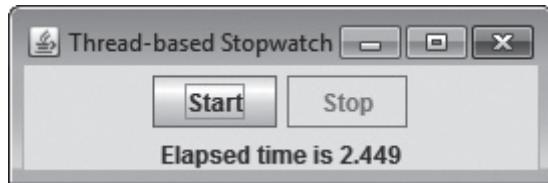
Uma solução é usar os métodos **invokeLater()** ou **invokeAndWait()** definidos por **SwingUtilities** de forma semelhante a como você tem usado **invokeLater()** para construir e exibir a GUI. Vamos recapitular. Eles são mostrados novamente abaixo:

```
static void invokeLater(Runnable obj)
static void invokeAndWait(Runnable obj)
    throws InterruptedException, InvocationTargetException
```

Aqui, *obj* é um objeto **Runnable** que terá seu método **run()** chamado pela thread de despacho de evento. Dentro de **run()** deve entrar o código que interage com um componente do Swing. Portanto, quando você precisar atualizar um componente, insira o código dentro de um objeto **Runnable** e passe esse objeto para **invokeLater()** ou **invokeAndWait()**. Isso faz o código ser executado na thread de despacho de evento, ou seja, o componente pode ser alterado com segurança.

A diferença entre os dois métodos é que **invokeLater()** retorna imediatamente e **invokeAndWait()** espera até **obj.run()** retornar. Normalmente as pessoas preferem **invokeLater()**. No entanto, como veremos em breve, na construção da GUI inicial de um applet, é melhor usar **invokeAndWait()**. Para simplificar, o restante desta discussão usará **invokeLater()**, mas os princípios gerais também se aplicam a **invokeAndWait()**.

Abaixo temos um exemplo que ilustra um modo de tratarmos uma thread separada que atualiza continuamente a GUI. É uma versão melhorada da classe **StopWatch** criada na seção Tente isto 17-1. Essa versão exibe o tempo passado enquanto o cronômetro está sendo executado. Ela faz isso criando uma thread separada que atualiza o rótulo de tempo decorrido 10 vezes por segundo. Observe o uso de **invokeLater()** dentro da segunda thread. Um exemplo da saída é mostrado aqui:



```
// Uma versão melhorada da classe StopWatch da seção Tente isto 17-1.
// Esta versão usa uma thread separada para exibir o tempo decorrido
// quando o cronômetro está sendo executado.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.Calendar;

class ThreadStopWatch {

    JLabel jlab; // exibe o tempo decorrido

    JButton jbtnStart; // inicia o cronômetro
    JButton jbtnStop; // interrompe o cronômetro

    long start; // contém a hora inicial em milissegundos

    boolean running=false; // verdadeiro quando o cronômetro está sendo executado.

    Thread thrd; // referência à thread de marcação do tempo. ← Referenciará a
                  // thread que atualiza a exibição do tempo enquanto
                  // o cronômetro é executado.

    ThreadStopWatch() {
        // Cria um container JFrame.
        JFrame jfrm = new JFrame("Thread-based Stopwatch");

        // Especifica FlowLayout como gerenciador de layout.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(230, 90);
    }
}
```

```

// Encerra o programa quando o usuário fecha o aplicativo.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Cria o rótulo de tempo decorrido.
jlab = new JLabel("Press Start to begin timing.");

// Cria os botões Start e Stop.
jbtnStart = new JButton("Start");
jbtnStop = new JButton("Stop");

// Inicialmente desativa o botão Stop.
jbtnStop.setEnabled(false);

// Cria a instância de Runnable que será a segunda thread.
Runnable myThread = new Runnable() { ← Cria um objeto Runnable
    // Este método será executado em uma thread separada. que será executado em
    public void run() { sua própria thread.
        try {
            // Relata o tempo decorrido a cada décimo de segundo.
            for( ; ; ) {
                // Faz uma pausa por um décimo de segundo.
                Thread.sleep(100); ← Faz uma pausa por um décimo de segundo.

                // Chama updateTime() na thread de despacho de evento.
                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        updateTime(); ← Atualiza o tempo decorrido. Esta
                    }
                });
            }
        } catch(InterruptedException exc) {
            System.out.println("Call to sleep was interrupted.");
            System.exit(1);
        }
    };
};

// Cria uma nova thread.
thrd = new Thread(myThread); ← Cria a thread.

// Inicia a thread.
thrd.start(); ← Inicia a thread.

// Adiciona os ouvintes de ação para os
// botões Start e Stop.
jbtnStart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {

```

```
// Armazena a hora inicial.  
start = Calendar.getInstance().getTimeInMillis();  
  
// Inverte o estado dos botões.  
jbtnStop.setEnabled(true);  
jbtnStart.setEnabled(false);  
  
// Inicia o cronômetro.  
running = true;  
}  
});  
  
jbtnStop.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {  
        long stop = Calendar.getInstance().getTimeInMillis();  
  
        // Calcula o tempo decorrido.  
        jlab.setText("Elapsed time is "  
            + (double) (stop - start)/1000);  
  
        // Inverte o estado dos botões.  
        jbtnStart.setEnabled(true);  
        jbtnStop.setEnabled(false);  
  
        // Interrompe o cronômetro.  
        running = false;  
    }  
});  
  
// Adiciona os botões e o rótulo ao painel de conteúdo.  
jfrm.add(jbtnStart);  
jfrm.add(jbtnStop);  
jfrm.add(jlab);  
  
// Exibe o quadro.  
jfrm.setVisible(true);  
}  
  
// Atualiza a exibição do tempo decorrido.  
void updateTime() { ←———— Este método é executado na  
    if(!running) return; thread de despacho de evento.  
  
    long temp = Calendar.getInstance().getTimeInMillis();  
    jlab.setText("Elapsed time is " +  
        (double) (temp - start)/1000);  
}
```

```

public static void main(String[] args) {
    // Cria o quadro na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ThreadStopWatch();
        }
    });
}
}

```

Examinemos em detalhes como esse programa funciona. Primeiro, observe os campos **running** e **thrd**. Inicialmente a variável **running** é configurada com **false**. Quando o cronômetro está sendo executado, ela é configurada com **true**. Quando **running** é igual a **true**, o tempo decorrido é exibido. O campo **thrd** conterá uma referência à thread que atualiza o tempo.

Em seguida, observe como a segunda thread é criada. Primeiro, um objeto **Runnable** chamado **myThread** é criado pelo código a seguir:

```

// Cria a instância de Runnable que será a segunda thread.
Runnable myThread = new Runnable() {
    // Este método será executado em uma thread separada.
    public void run() {
        try {
            // Relata o tempo decorrido a cada décimo de segundo.
            for(; ; ) {
                // Faz uma pausa por um décimo de segundo.
                Thread.sleep(100);

                // Chama updateTime() na thread de despacho de evento.
                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        updateTime();
                    }
                });
            }
        } catch(InterruptedException exc) {
            System.out.println("Call to sleep was interrupted.");
            System.exit(1);
        }
    }
};

```

A interface **Runnable** só define um método: **run()**. Esse método é executado em uma thread separada. Quando a thread começa a ser executada, ela chama **run()**. Dentro de **run()**, é estabelecido um laço infinito que apenas entra em suspensão por um décimo de segundo e então chama **invokeLater()**. Lembre-se, o código do laço é executado em sua própria thread e não na thread de despacho de

evento. (Se estivesse sendo executado na thread de despacho de evento, a janela poderia ficar lenta ou não responder porque o laço impediria que outras partes do programa fossem executadas.)

Agora, observe o argumento passado para o método `invokeLater()`. Ele é chamado com **Runnable**, cujo método `run()` chama o método `updateTime()`. O método `updateTime()` é mostrado aqui:

```
// Atualiza a exibição do tempo decorrido.
void updateTime() {
    if(!running) return;

    long temp = Calendar.getInstance().getTimeInMillis();
    jlab.setText("Elapsed time is " +
                (double) (temp - start)/1000);
}
```

Primeiro o código verifica o valor de **running**. Se for **false**, o método retorna imediatamente, porque o cronômetro não está sendo usado. Caso contrário, **updateTime()** obtém a hora atual, subtrai dela a hora inicial e então atualiza **jlab** com o tempo decorrido atual. Como **updateTime()** altera o conteúdo do rótulo, deve ser executado na thread de despacho de evento. Isso é feito com uma chamada a **invokeLater()** com um argumento **Runnable** que chama **updateTime()**. Assim, **updateTime()** é executado na thread de despacho de evento e pode atualizar com segurança a hora mostrada em **jlab**. Quando o programa é encerrado (com um clique na caixa Fechar), essa thread também é encerrada automaticamente.

Após a declaração do objeto **myThread**, é criada uma nova thread que o utiliza e que é então iniciada pelo código a seguir:

```
// Cria uma nova thread.
thrd = new Thread(myThread);

// Inicia a thread.
thrd.start();
```

Agora, a nova thread está sendo executada. No entanto, não está sendo usada para nada porque o cronômetro ainda não foi iniciado.

Quando o usuário clica no botão Start, a hora atual do sistema é obtida e armazenada na variável **start**. O botão Start é desativado, o botão Stop é ativado e **running** é configurada com **true**. Quando o botão Stop é pressionado, a hora atual do sistema é obtida. A diferença entre a hora atual e a hora inicial é exibida em **jlab**. Em seguida, o estado de ativação dos botões é invertido e **running** é configurada com **false**.

Como esse programa mostra, o segredo do uso seguro de threads no Swing é assegurar que qualquer código que interaja com um componente do Swing seja executado na thread de despacho de evento. Se você não tiver certeza de qual thread está executando um trecho de código, pode chamar o método `isEventDispatchThread()` definido por **SwingUtilities**.

Verificação do progresso

1. Por que seu código deve interagir com componentes do Swing por intermédio da thread de despacho de evento?
2. O que `invokeLater()` faz?

USE Timer

O exemplo do cronômetro mostrou como uma thread separada pode ser usada em conjunto com `invokeLater()` para interagir com um componente do Swing. Como você verá, na verdade isso não precisa ser tão complicado. Em alguns casos, não é necessário criar uma thread separada explicitamente. Em vez disso, o que você precisa é de um *timer* que gere um evento em intervalos periódicos. Por exemplo, para rolar um banner, você pode usar um timer para redesenhar o banner repetidamente e obter uma aparência animada. (Consulte a seção Tente isto 21-1.)

A classe de timer definida pelo Swing se chama **Timer**. Ela fica no pacote `javax.swing`. Não a confunda com outra classe chamada **Timer** que fica no pacote `java.util`. Você terá que especificar explicitamente que timer está usando quando importar os dois pacotes para o mesmo programa.

A classe **Timer** do Swing aciona automaticamente um evento de ação que será recebido pelo ouvinte especificado. Já que se trata de um evento, ele será executado automaticamente na thread de despacho de evento. Logo, o método `actionPerformed()` definido por **ActionListener** e registrado no timer será executado na thread de despacho de evento como usual. Não há necessidade de usar `invokeLater()` ou `invokeAndWait()`. Portanto, para usar **Timer**, você só precisa criar uma instância dessa classe, especificando o ouvinte de ação que receberá o evento.

Timer só define um construtor, mostrado abaixo:

```
Timer(int período, ActionListener al)
```

Aqui, *período* especifica o tempo passado entre a ocorrência de eventos em milissegundos. Em outras palavras, *período* especifica o intervalo de tempo. O ouvinte de ação que receberá os eventos é especificado por *al*. Você pode especificar ouvintes de ação adicionais para serem notificados quando o timer expirar chamando **addActionListener()** no timer.

Para iniciar o timer, chame **start()**. Para interromper o timer, chame **stop()**. Esses métodos são mostrados a seguir:

```
void start()
void stop()
```

Respostas:

1. Isso evita conflitos entre threads, como duas threads diferentes tentando atualizar o mesmo componente ao mesmo tempo.
2. Faz um objeto **Runnable** ser executado na thread de despacho de evento. Ele retorna imediatamente em vez de esperar a thread terminar.

Por padrão, o timer continua a acionar eventos no intervalo especificado. Você pode fazê-lo acionar apenas um evento chamando o método **setRepeats()**, visto abaixo:

```
void setRepeats(boolean repete)
```

Aqui, se *repete* for **true**, a contagem do tempo continua. Se for **false**, ela é interrompida após um intervalo.

Como mencionado, o programa do cronômetro baseado em thread mostrado na seção anterior é mais complicado do que deveria. Em vez de criarmos explicitamente uma thread separada, um timer pode ser usado. Essa abordagem é mostrada pela versão a seguir do programa:

```
// Esta versão do cronômetro usa a classe Timer.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.Calendar;

class TimerStopWatch {

    JLabel jlab; // exibe o tempo decorrido

    JButton jbtnStart; // inicia o cronômetro
    JButton jbtnStop; // interrompe o cronômetro

    long start; // contém a hora inicial em milissegundos

    Timer swTimer; // o timer do cronômetro ←———— Usa um Timer para atualizar
                    // a exibição da hora.

    TimerStopWatch() {

        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("Timer-based Stopwatch");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(230, 90);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria o rótulo de tempo decorrido.
        jlab = new JLabel("Press Start to begin timing.");

        // Cria os botões Start e Stop.
        jbtnStart = new JButton("Start");

```

```

jbtnStop = new JButton("Stop");
jbtnStop.setEnabled(false);

// Este ouvinte de ação é chamado quando
// o timer expira.
ActionListener timerAL = new ActionListener() { ← Este ouvinte é
    public void actionPerformed(ActionEvent ae) {
        updateTime();
    }
};

// Cria um timer que expira a cada décimo de segundo.
swTimer = new Timer(100, timerAL); ← Cria o timer. Observe
// Adiciona os ouvintes de ação para os
// botões Start e Stop.
jbtnStart.addActionListener(new ActionListener() { que timerAL é
    public void actionPerformed(ActionEvent ae) {
        // Armazena a hora inicial.
        start = Calendar.getInstance().getTimeInMillis();

        // Inverte o estado dos botões.
        jbtnStop.setEnabled(true);
        jbtnStart.setEnabled(false);

        // Inicia o cronômetro.
        swTimer.start();
    }
});

jbtnStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        long stop = Calendar.getInstance().getTimeInMillis();

        // Calcula o tempo decorrido.
        jlab.setText("Elapsed time is "
            + (double) (stop - start)/1000);

        // Inverte o estado dos botões.
        jbtnStart.setEnabled(true);
        jbtnStop.setEnabled(false);

        // Interrompe o cronômetro.
        swTimer.stop();
    }
});

// Adiciona os botões e o rótulo ao painel de conteúdo.

```

```
jfrm.add(jbtnStart);
jfrm.add(jbtnStop);
jfrm.add(jlab);

// Exibe o quadro.
jfrm.setVisible(true);
}

// Atualiza a exibição do tempo decorrido. Observe que a
// variável running não é mais necessária.
void updateTime() {
    long temp = Calendar.getInstance().getTimeInMillis();
    jlab.setText("Elapsed time is " +
                (double) (temp - start)/1000);
}

public static void main(String[] args) {
    // Cria o quadro na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new TimerStopWatch();
        }
    });
}
```

Como você pode ver, o programa é significativamente mais curto porque não contém o código que cria uma thread. Ele também não precisa usar a variável **running**. O uso de **Timer** simplifica muito a tarefa. Observe como precisamos de pouco código para usar o timer. Primeiro, ele é criado apenas por esta linha:

```
| swTimer = new Timer(100, timerAL);
```

Elá constrói um timer que expira a cada décimo de segundo. Aqui, **timerAL** especifica o ouvinte de ação que receberá os eventos de ação gerados pelo timer. Ele é mostrado abaixo:

```
ActionListener timerAL = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        updateTime();
    }
};
```

O método **actionPerformed()** apenas chama **updateTime()**. Como esse código já está sendo executado na thread de despacho de evento, não há necessidade de usar **invokeLater()**.

O timer é iniciado quando o usuário clica no botão Start. Ele é interrompido quando Stop é clicado. Como você verá quando testar o programa, o timer pode ser iniciado e interrompido repetidamente. Não há restrições a esse respeito.

Se você tiver que usar uma thread separada só para manter a GUI atualizada, usar **javax.swing.Timer** é uma opção melhor do que criar manualmente sua própria thread. Mas se for usar uma thread separada para executar outras tarefas, não há alternativa a não ser criá-la explicitamente.

Verificação do progresso

1. Que método inicia um **Timer**?
2. Que evento um **Timer** gera?

Pergunte ao especialista

P O que é a classe **SwingWorker**?

R A partir do JDK 6, o Swing passou a fornecer uma classe chamada **SwingWorker**, que pode ser usada para simplificar a execução de alguns tipos de tarefas *multithread*. Duas das principais vantagens de **SwingWorker** são as seguintes: em primeiro lugar, ela fornece um meio interno para execução de códigos na thread de despacho de evento, ou seja, evita o uso de **invokeLater()** e **invokeAndWait()**; em segundo lugar, ela otimiza a criação e o gerenciamento de threads de segundo plano.

SwingWorker é uma classe abstrata genérica que é declarada assim:

```
SwingWorker<T, V>
```

Aqui, **T** especifica o tipo de resultado de seu método **doInBackground()** (que será discutido em breve), e **V** representa o tipo de qualquer resultado temporário. Quando um dos dois (ou ambos) não é exigido pela thread de trabalho, **Void** pode ser usado.

Para criar um **SwingWorker**, você deve estender a classe, fornecendo no mínimo uma implementação para **doInBackground()**. Dentro desse método insira o código que deseja que seja executado em uma thread de segundo plano. Ele é mostrado abaixo:

```
protected T doInBackground() throws Exception
```

O método deve lançar uma exceção se, por alguma razão, a tarefa de segundo plano falhar.

Uma vez que você criar um **SwingWorker**, poderá iniciar a thread chamando o método **execute()**, mostrado a seguir:

```
final void execute()
```

A tarefa será executada em uma thread de trabalho fornecida automaticamente.

Respostas:

1. **start()**
2. **Timer** gera um evento de ação.

Pergunte ao especialista (continuação)

Para obter o valor retornado por **doInBackground()**, chame **get()**. Esta é uma das versões:

```
final T get() throws InterruptedException, ExecutionException
```

Esse método esperará até um valor estar disponível; ele não deve ser chamado pela thread de despacho de evento até que o valor tenha sido retornado por **doInBackground()**.

Quando **doInBackground()** terminar, o método **done()** será chamado na thread de despacho de evento. Ele é mostrado aqui:

```
protected void done()
```

Você pode sobrepor esse método para obter o resultado retornado por **doInBackground()**.

É possível “publicar” resultados temporários chamando o método **publish()**, mostrado abaixo:

```
protected final void publish(V ... dados)
```

Aqui, *dados* é um argumento de tamanho variável contendo os dados que você deseja que outras partes de seu aplicativo conheçam. O aplicativo tomará conhecimento trabalhando com outro método de **SwingWorker** chamado **process()**. O interessante é que **process()** é chamado na thread de despacho de evento. Ou seja, você pode usar as informações que ele recebe para atualizar a GUI. Os dados são passados para **process()** em um objeto **List**, que faz parte do Collections Framework de Java. (**List** é descrita no Capítulo 25.) O método **process()** tem esta forma:

```
protected void process(List<V> lista)
```

É possível que duas ou mais chamadas a **publish()** ocorram antes que **process()** seja realmente executado, já que ele é chamado na thread de despacho de evento. Portanto, você deve percorrer *lista* para verificar se recuperou todos os dados. Pode usar um laço **for** de estilo for-each para esse fim, como mostrado no exemplo a seguir.

Para ver **SwingWorker** em ação, você pode substituir o código de **Runnable** existente no primeiro exemplo do cronômetro, **ThreadStopWatch**, pelo **SwingWorker** mostrado aqui:

```
// Cria uma nova thread SwingWorker para atualizar a hora.
final SwingWorker<Void, Long> sw = new SwingWorker<Void, Long>() {
    public Void doInBackground() {
        try {
            // Relata o tempo decorrido a cada décimo de segundo.
            for(;;) {
                // Faz uma pausa por um décimo de segundo.
                Thread.sleep(100);
                publish(Calendar.getInstance().getTimeInMillis());
            }
        } catch(InterruptedException exc) {
            System.out.println("Call to sleep was interrupted.");
            System.exit(1);
        }
        return null; // Valor de retorno de espaço reservado
    }
}
```

```

// Atualiza a exibição do tempo decorrido. Este método
// é chamado na thread de despacho de evento
protected void process(List<Long> times) {
    if(!running) return;

    for(long curTime : times)
        jlab.setText("Elapsed time is " +
                    (double) (curTime - start)/1000);
}
};

sw.execute();

```

Você também precisa adicionar esta instrução **import**:

```
import java.util.List;
```

Após essas alterações, o programa funcionará da mesma forma, só que agora **SwingWorker** está sendo usada para fornecer a thread de segundo plano.

CRIE APPLETS SWING

Todos os exemplos anteriores do Swing são aplicativos Java autônomos executados na área de trabalho, mas você também pode usar o Swing para criar a GUI de um applet. Os applets foram introduzidos no Capítulo 15, onde a forma geral de um applet baseado no AWT foi descrita. Como você verá, embora os applets baseados no Swing sejam semelhantes aos baseados no AWT mostrados no Capítulo 15, algumas regras especiais são aplicáveis.

Como vimos no Capítulo 15, todos os applets são baseados na classe **Applet**. No entanto, para criar um applet baseado no Swing, você usará **JApplet**, que herda **Applet**. **JApplet** é um contêiner de nível superior e *não* é derivada de **JComponent**. Como **JApplet** é um contêiner de nível superior do Swing, ele inclui os diversos painéis descritos no Capítulo 17. Ou seja, todos os componentes são adicionados ao seu painel de conteúdo, da mesma forma como você vem adicionando componentes ao painel de conteúdo de **JFrame**. Além disso, toda a interação com componentes do Swing deve ocorrer na thread de despacho de evento, conforme descrito na seção anterior.

Lembre-se de que os applets não são executados diretamente na área de trabalho. Em vez disso, eles são executados dentro de um navegador habilitado com Java ou por um visualizador de applets, como o **appletviewer** fornecido pelo JDK. Durante o desenvolvimento, é mais fácil testar applets usando um visualizador de applets do que carregá-los em um navegador. No entanto, no teste final, um navegador deve ser usado.

Todos os applets (usando ou não o Swing) têm os mesmos métodos de ciclo de vida. Você deve lembrar que eles são **init()**, **start()**, **stop()** e **destroy()**. Esses métodos são definidos por **Applet** e herdados por **JApplet**. Implementações vazias padrão de todos eles são fornecidas. Portanto, os applets não precisam sobrepor os métodos que não usam.

Como os aplicativos Swing, um applet deve interagir com seus componentes usando a thread de despacho de evento. Ou seja, você não pode usar o método `init()` para construir a GUI inicial. Em vez disso, chamará `invokeAndWait()` dentro de `init()` para especificar um objeto **Runnable** cujo método `run()` será executado na thread de despacho de evento. Uma forma básica do método `init()` que usa essa abordagem é mostrada aqui:

```
public void init() {
    try {
        SwingUtilities.invokeAndWait(new Runnable () {
            public void run() {
                guiInit(); // um método que inicializa os componentes do Swing
            }
        });
    } catch(Exception exc) {
        System.out.println("Can't create because of " + exc);
    }
}
```

Um applet baseado no Swing deve criar sua GUI na thread de despacho de evento.

Dentro de `run()`, um método de nome `guiInit()` é chamado. Ele é fornecido por você e configurará e inicializará os componentes do Swing. É claro que o nome do método é arbitrário.

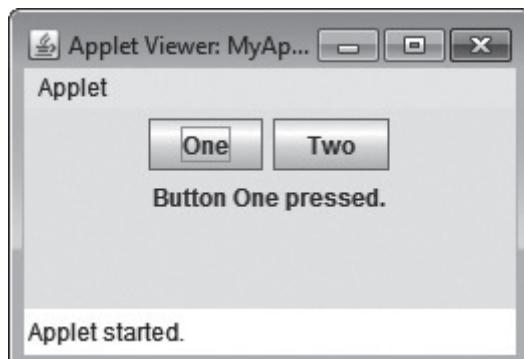
Pergunte ao especialista

P Todos os aplicativos deste livro chamam `invokeLater()` para construir a GUI. Por que um applet usa `invokeAndWait()`?

R Os applets devem usar `invokeAndWait()` porque o método `init()` não deve retornar até o processo de inicialização inteiro ter sido concluído. Na verdade, o método `start()` não pode ser chamado antes da inicialização, ou seja, a GUI deve estar totalmente construída.

Um applet Swing simples

Este é um applet simples baseado no Swing que exibe dois botões. Sempre que um botão é clicado, uma mensagem é exibida informando que botão foi pressionado. Um exemplo de saída do applet sendo executado pelo `appletviewer` é mostrado abaixo:



```
// Um applet simples baseado no Swing

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/*
Esta HTML pode ser usada para iniciar o applet:

<object code="MyApplet" width=240 height=100>
</object>

*/

public class MyApplet extends JApplet {
    JButton jbtnOne;
    JButton jbtnTwo;

    JLabel jlab;

    // Chamado primeiro.
    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable () {
                public void run() {
                    guiInit(); // inicializa a GUI.
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    // Configura e inicializa a GUI.
    private void guiInit() {
        // Configura o applet para usar o layout de fluxo.
        setLayout(new FlowLayout());

        // Cria dois botões e um rótulo.
        jbtnOne = new JButton("One");
        jbtnTwo = new JButton("Two");

        jlab = new JLabel("Press a button.");

        // Adiciona ouvintes de ação para os botões.
        jbtnOne.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {
                jlab.setText("Button One pressed.");
            }
        });
    }
}
```

```

jbtnTwo.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Button Two pressed.");
    }
});

// Adiciona os componentes ao painel de conteúdo do applet.
add(jbtnOne);
add(jbtnTwo);
add(jlab);
}
}

```

Há duas coisas importantes nesse applet. Primeiro, o método **init()** inicializa a GUI do Swing definindo uma chamada a **guiInit()**. Isso é feito com o uso de **invokeAndWait()**. Dentro de **guiInit()**, dois botões e um rótulo são criados. Em seguida, um ouvinte de ação é adicionado a cada botão. Eles simplesmente configuram o texto do rótulo para indicar qual botão foi pressionado. Embora esse exemplo seja muito simples, essa mesma abordagem geral deve ser empregada na construção de qualquer GUI do Swing a ser usada por um applet.

O segundo ponto interessante do applet é o fato de **start()**, **stop()** e **destroy()** não serem usados por ele. Isso é comum em applets simples. Nesses casos, não há razão para a especificação de implementações vazias porque **Applet** nos fornece implementações padrão. Portanto, se não precisarmos usar um dos métodos de ciclo de vida do applet, não teremos que sobrepor-lo.

Verificação do progresso

1. Que classe é usada na criação de um applet baseado no Swing?
2. Que método você chamaria para construir a GUI na thread de despacho de evento quando um applet baseado no Swing fosse inicializado?

TENTE ISTO 21-1 Role texto em um applet

`ScrollText.java`

Este projeto demonstra tanto applets baseados no Swing quanto temporizadores do Swing. Ele usa um temporizador para rolar uma mensagem de texto dentro de um **JLabel**. Como você verá, o código que faz isso é surpreendentemente curto e conciso porque a classe **Timer** do Swing torna o processo muito fácil de codificar.

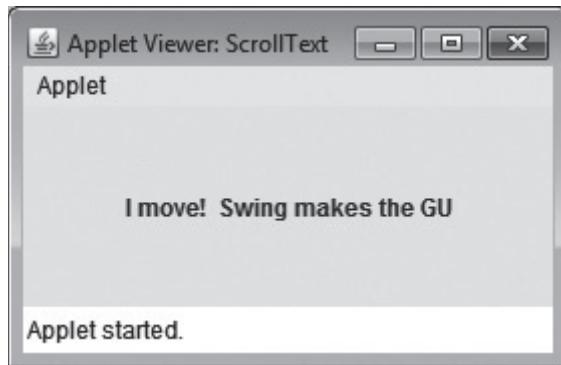
Há outro ponto interessante nesse programa. Você deve lembrar que a seção Tente isto 15-1 criou um applet baseado no AWT que exibia um banner móvel.

Respostas:

1. **JApplet**
2. **invokeAndWait()**

Foi mencionado naquela seção que ocorreria alguma oscilação e que a maneira de evitá-la seria usar um buffer duplo. Felizmente, o Swing usa um buffer duplo por padrão. Portanto, quando você executar esse exemplo, o banner móvel terá uma aparência regular e não haverá oscilação. O uso do buffer duplo é outra vantagem importante que o Swing traz para o design de GUIs.

Um exemplo da saída do applet é mostrado aqui:



PASSO A PASSO

- Crie um arquivo chamado **ScrollText.java** e adicione os comentários e as instruções **import** a seguir:

```
// Tente isto 21-1: Um applet baseado no Swing
// que rola texto em um rótulo.

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/*
Esta HTML pode ser usada para iniciar o applet:

<object code="ScrollText" width=240 height=100>
</object>

*/
```

- Comece a criar a classe **ScrollText** como mostrado aqui:

```
public class ScrollText extends JApplet {

    JLabel jlab;

    String msg = " Swing makes the GUI move! ";

    ActionListener scroller;

    Timer stTimer; // este temporizador controla a velocidade da rolagem
```

ScrollText é um applet baseado no Swing, portanto, ele estende **JApplet**. Em seguida, declara o rótulo que exibirá a mensagem rolante, a mensagem que será rolada, o ouvinte de ação que executará a rolagem e um temporizador de controle de velocidade da rolagem.

3. Adicione o método **init()** mostrado a seguir:

```
// Inicializa o applet.
public void init() {
    try {
        SwingUtilities.invokeAndWait(new Runnable {
            public void run() {
                guiInit();
            }
        });
    } catch(Exception exc) {
        System.out.println("Can't create because of " + exc);
    }
}
```

Esse método apenas chama **guiInit()** na thread de despacho de evento, como descrito anteriormente.

4. Adicione os métodos **start()**, **stop()** e **destroy()**, mostrados abaixo:

```
// Inicia o temporizador quando o applet é iniciado.
public void start() {
    stTimer.start();
}

// Interrompe o temporizador quando o applet é encerrado.
public void stop() {
    stTimer.stop();
}

// Interrompe o temporizador quando o applet é destruído.
public void destroy() {
    stTimer.stop();
}
```

Mesmo sendo muito curtos, esses métodos desempenham funções importantes. Sempre que o applet é exibido em uma página, o temporizador é iniciado. Quando o navegador encerra o applet, o temporizador é interrompido. O temporizador também é interrompido quando o applet é destruído.

5. Termine o applet com o método **guiInit** a seguir:

```
// Inicializa a GUI do temporizador.
private void guiInit() {

    // Cria o rótulo que rolará a mensagem.
    jlab = new JLabel(msg);
```

```

jlab.setHorizontalAlignment(SwingConstants.CENTER);

// Cria o ouvinte de ação para o temporizador.
scroller = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        // Rola a mensagem um caractere para a esquerda.
        char ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;
        jlab.setText(msg);
    }
};

// Cria o temporizador.
stTimer = new Timer(200, scroller);

// Adiciona o rótulo ao painel de conteúdo do applet.
add(jlab);
}
}

```

Esse método cria o rótulo em que o texto é rolado. O texto é alinhado ao centro. Isso é feito para que o texto móvel fique mais atraente. Tecnicamente, não é necessário. Em seguida, é criado um ouvinte de ação que rola o texto um caractere para a esquerda sempre que é chamado. Esse ouvinte é então passado para o construtor de **Timer** quando o temporizador é criado. O tempo cronometrado é de um quinto de segundo, mas você pode fazer testes com o intervalo de retardo. É claro que, quanto mais curto o período, mais rápida a rolagem. Para concluir, o rótulo é adicionado ao painel de conteúdo.

6. O applet **ScrollText** completo é mostrado aqui:

```

// Tente isto 21-1: Um applet baseado no Swing que
// rola texto em um rótulo.

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/*
Esta HTML pode ser usada para iniciar o applet:

<object code="ScrollText" width=240 height=100>
</object>

*/
public class ScrollText extends JApplet {
    JLabel jlab;

```

```
String msg = " Swing makes the GUI move! ";

ActionListener scroller;

Timer stTimer; // este temporizador controla a velocidade da
rolagem

// Inicializa o applet.
public void init() {
    try {
        SwingUtilities.invokeAndWait(new Runnable () {
            public void run() {
                guiInit();
            }
        });
    } catch(Exception exc) {
        System.out.println("Can't create because of " + exc);
    }
}

// Inicia o temporizador quando o applet é iniciado.
public void start() {
    stTimer.start();
}

// Interrompe o temporizador quando o applet é encerrado.
public void stop() {
    stTimer.stop();
}

// Interrompe o temporizador quando o applet é destruído.
public void destroy() {
    stTimer.stop();
}

// Inicializa a GUI do temporizador.
private void guiInit() {

    // Cria o rótulo que rolará a mensagem.
    jlab = new JLabel(msg);
    jlab.setHorizontalAlignment(SwingConstants.CENTER);

    // Cria o ouvinte de ação para o temporizador.
    scroller = new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            // Rola a mensagem um caractere para a esquerda.
            char ch = msg.charAt(0);
            msg = msg.substring(1, msg.length());
            msg += ch;
            jlab.setText(msg);
        }
    };
}
```

```

    // Cria o temporizador.
    stTimer = new Timer(200, scroller);

    // Adiciona o rótulo ao painel de conteúdo do applet.
    add(jlab);
}
}

```

GERANDO COMPONENTES

Os capítulos anteriores enfocaram principalmente os elementos do conjunto de componentes do Swing e as técnicas necessárias para usá-los. No entanto, há outra parte do Swing que permite que você crie sua própria saída visual gravando diretamente na área de exibição de um quadro, painel ou um dos outros componentes do Swing, como **JLabel**. Mesmo que muitos (talvez a maioria) dos empregos do Swing *não* envolvam o desenho diretamente na superfície de um componente, ele está disponível para os aplicativos que precisarem desse recurso. A gravação da saída diretamente na superfície de um componente envolve um ou mais métodos de desenho, como **drawLine()** ou **drawRect()**, e requer que você tenha algum controle manual sobre o processo de pintura. Embora uma discussão detalhada da pintura não faça parte do escopo deste livro, esta seção aborda as técnicas básicas.

Fundamentos da geração de componentes

A primeira coisa que é preciso entender sobre a pintura de componentes no Swing é que ela é um processo um pouco complicado. Na verdade, o Swing tem um subsistema inteiro dedicado ao gerenciamento da pintura, que segue a abordagem original baseada no AWT. Antes de examinarmos as particularidades da pintura de componentes baseada no Swing, é útil investigarmos o mecanismo do AWT em que ela se baseia.

Para começar, devemos lembrar que **JComponent** herda a classe **Component** do AWT. Essa classe define um método chamado **paint()**, o qual é chamado quando o componente tem que ser exibido na tela. Como regra geral, **paint()** não é chamado pelo programa. Em vez disso, é chamado pelo sistema de tempo de execução sempre que um componente deve ser gerado. Essa situação pode ocorrer por vários razões. Por exemplo, a janela em que o componente é exibido pode ser sobreposta por outra janela para depois surgir novamente. Ou a janela pode ser minimizada e então restaurada. O método **paint()** também é chamado quando um programa começa a ser executado. Na criação de código baseado no AWT, o aplicativo sobreporá **paint()** quando tiver que gravar a saída diretamente na superfície do componente.

Embora os componentes leves do Swing herdem o método **paint()** de **Component**, você não o sobreporá para pintar diretamente na superfície de um componente. A razão é que o Swing usa uma abordagem um pouco mais sofisticada da pintura que envolve três métodos distintos: **paintComponent()**, **paintBorder()** e **paintChildren()**. Esses métodos pintam a parte indicada de um componente e dividem o processo de pintura em

suas três ações lógicas e distintas. Em um componente leve, o método **paint()** original do AWT apenas executa chamadas a esses métodos, na ordem que acabamos de mostrar.

Para pintar a superfície de um componente do Swing, você criará uma subclasse do componente e então sobreporá seu método **paintComponent()**. Esse é o método que pinta o interior do componente. Normalmente os outros dois métodos de pintura não são sobrepostos. Como regra geral, ao sobrepor **paintComponent()**, a primeira coisa que você deve fazer é chamar **super.paintComponent()**, para que ocorra a parte do processo de pintura referente à superclasse. Depois, grave a saída que deseja exibir.

O método **paintComponent()** é mostrado aqui:

```
protected void paintComponent(Graphics g)
```

Observe que o método é protegido. Logo, pode ser chamado por uma subclasse. O parâmetro *g* é o contexto gráfico em que a saída é gravada.

O contexto gráfico

Em Java, cada componente tem um *contexto gráfico* associado. Ele fica encapsulado na classe **java.awt.Graphics** e mantém várias informações sobre o ambiente de exibição, como a cor e a fonte do desenho. Como acabamos de descrever, o contexto é passado para os diversos métodos de pintura, como **paintComponent()**. Quando pintamos em um componente, fazemos isso por intermédio desse contexto, usando os métodos que ele fornece.

Graphics define muitos métodos que gravam a saída no componente. Um deles, **drawString()**, foi usado no Capítulo 15. Aqui estão mais cinco exemplos:

```
void drawLine(int xInicial, int yInicial, int xFinal, int yFinal)  
void drawRect(int esquerda, int topo, int largura, int altura)  
void fillRect(int esquerda, int topo, int largura, int altura)  
void drawOval(int esquerda, int topo, int largura, int altura)  
void fillOval(int esquerda, int topo, int largura, int altura)
```

O método **drawLine()** desenha uma linha com a cor de desenho atual começando em *xInicial,yInicial* e terminando em *xFinal,yFinal*. **drawRect()** desenha um retângulo com a cor de desenho atual começando em *esquerda,topo* e com largura e altura determinadas por *largura* e *altura*. **fillRect()** é igual a **drawRect()**, exceto pelo fato de o retângulo ser preenchido com a cor de desenho atual. **drawOval()** desenha uma forma oval com a cor de desenho atual começando em *esquerda,topo* e com largura e altura determinadas por *largura* e *altura*. **fillOval()** é igual a **drawOval()**, exceto pelo fato de a forma oval ser preenchida com a cor de desenho atual. Por conta própria, você deve examinar a classe **Graphics** na documentação Java para conhecer seus outros recursos.

Em todos os componentes, a origem da área de exibição fica no canto superior esquerdo e tem as coordenadas 0,0. Além disso, todos os locais de exibição se baseiam nessa origem. As coordenadas são especificadas em pixels. Portanto, se você gravar uma saída no local 12,14, ela será gravada 12 pixels à direita e 14 pixels à esquerda do canto superior esquerdo.

Calcule a área de desenho

Ao desenhar na superfície de um componente, você pode ter que restringir sua saída à área que fica dentro da borda. Mesmo com o Swing aparando automaticamente qualquer saída que exceda os limites de um componente, é possível a pintura atingir a região da borda, que será então sobreposta quando a borda for desenhada. Para evitar isso, você pode calcular a *área de desenho* do componente. Ela é a área definida pelo tamanho atual do componente menos o espaço usado pela borda. Portanto, antes de pintar em um componente, você deve obter a largura da borda e então ajustar o desenho de acordo.

Para obter a largura da borda, chame o método `getInsets()`, mostrado abaixo:

```
Insets getInsets()
```

Esse método é definido por **Container** e sobreposto por **JComponent**. Ele retorna um objeto **Insets** que contém as dimensões da borda. Os valores dos espaços adicionais podem ser obtidos com o uso destes campos:

```
int top;
int bottom;
int left;
int right;
```

Esses valores são então usados no cálculo da área de desenho dadas a largura e a altura do componente. Você pode obter a largura e a altura do componente chamando nele os métodos `getWidth()` e `getHeight()`, vistos a seguir:

```
int getWidth()
int getHeight()
```

Subtraindo o valor dos espaços adicionais, você estará calculando a largura e a altura gerais da área de desenho de um componente.

Solicite a geração do componente

Como acabamos de explicar, normalmente um componente é pintado somente quando seu método `paint()` é chamado, o que, no caso de um componente do Swing, gera chamadas a `paintComponent()`, `paintBorder()` e `paintChildren()`. Para solicitar pintura, você usará o método `repaint()`, como descrito no Capítulo 15. Sua chamada fará o sistema chamar `paint()` assim que possível. Como explicado, isso gera uma chamada a `paintComponent()`. Portanto, para gerar a saída na superfície de um componente, seu programa a armazenará até `paintComponent()` ser chamado. No método `paintComponent()` sobreposto, você desenhará a saída armazenada.

Um exemplo de geração de componente

Este é um programa que coloca em prática a discussão anterior. Ele cria uma classe chamada **PaintPanel** que estende **JPanel**. Usaremos a superfície de um **PaintPanel** para desenhar a saída. **PaintPanel** especifica uma borda. Assim, podemos ver facil-

mente sua área de pintura. Em seguida, o programa usa um objeto dessa classe para exibir um gráfico de barras que demonstra dados gerados aleatoriamente. Além do gráfico de barras, o programa inclui dois botões. Um nos permite exibir um conjunto de valores diferente; o outro permite a alteração do tamanho da borda ao redor do gráfico. Um exemplo da saída é mostrado aqui:



```
// Pinta linhas em um painel.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

// Esta classe estende JPanel. Ela sobrepõe
// o método paintComponent() para que dados
// aleatórios sejam demonstrados no painel.
class PaintPanel extends JPanel { ←————— PaintPanel estende JPanel.
    Insets ins; // contém os espaços adicionais do painel

    Random rand; // usada para gerar números aleatórios

    PaintPanel(int w, int h) {

        // Usa uma borda de linha vermelha.
        setBorder(BorderFactory.createLineBorder(Color.RED, 1));

        // Define a dimensão ideal como especificado.
        setPreferredSize(new Dimension(w, h));
    }
}
```

```

        rand = new Random();
    }

    // Sobrepõe o método paintComponent().
    protected void paintComponent(Graphics g) { ← Sobrepõe paintComponent()
                                                para pintar na superfície do
                                                componente.

        // Chame sempre o método da superclasse primeiro.
        super.paintComponent(g); ← Lembre-se de chamar a
                                implementação da superclasse.

        // Obtém a altura e largura do componente.
        int height = getHeight();
        int width = getWidth();

        // Obtém os espaços adicionais.
        insets = getInsets();

        // Preenche o painel demonstrando dados aleatórios
        // na forma de um gráfico de barras. ← Desenha o
                                                gráfico de dados
                                                aleatórios.
        for(int i=insets.left+5; i <= width-insets.right-5; i += 4) {
            // Obtém um número aleatório entre 0 e a
            // altura máxima da área de desenho.
            int h = rand.nextInt(height-insets.bottom);

            // Se um valor gerado ficar em cima ou muito próximo da borda,
            // ele é alterado para que fique dentro da borda.
            if(h <= insets.top) h = insets.top+1;

            // Desenha uma linha que representa os dados.
            g.drawLine(i, height-insets.bottom, i, h);
        }
    }

    // Altera o tamanho da borda.
    public void changeBorderSize(int size) {
        setBorder(
            BorderFactory.createLineBorder(Color.RED, size));
    }
}

// Demonstra a pintura diretamente em um painel.
class PaintDemo {

    JButton jbtnMore;
    JButton jbtnSize;
    JLabel jlab;
    PaintPanel pp;

    boolean big; // usa para alternar o tamanho do painel

    PaintDemo() {

```

```
// Cria um contêiner JFrame
JFrame jfrm = new JFrame("Painting Demo");

// Especifica FlowLayout como gerenciador de layout.
jfrm.setLayout(new FlowLayout());

// Fornece um tamanho inicial para o quadro.
jfrm.setSize(240, 260);

// Encerra o programa quando o usuário fecha o aplicativo.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Cria o painel que será pintado.
pp = new PaintPanel(100, 100);

// Cria os botões.
jbtnMore = new JButton("Show More Data");
jbtnSize = new JButton("Change Border Size");

// Descreve o gráfico.
jlab = new JLabel("Bar Graph of Random Data");

// Pinta novamente o painel quando o botão
// Show More Data é clicado.
jbtnMore.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        pp.repaint(); ← Solicita que o painel seja pintado.
    }
});

// Define o tamanho da borda do painel quando
// o botão Change Size Border é clicado.
// A alteração do tamanho da borda resulta
// automaticamente em uma nova pintura.
jbtnSize.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) { ← Altera o tamanho da
        if(!big) pp.changeBorderSize(5);
        else pp.changeBorderSize(1);
        big = !big;
    }
});

// Adiciona os botões, o rótulo e o painel ao painel de conteúdo.
jfrm.add(jlab);
jfrm.add(pp);
jfrm.add(jbtnMore);
jfrm.add(jbtnSize);

big = false;
```

```

    // Exibe o quadro.
    jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Cria o quadro na thread de despacho de evento.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new PaintDemo();
        }
    });
}
}

```

Esse programa é mais complicado do que outros deste capítulo. Ele também introduz alguns elementos novos. Portanto, merece uma verificação mais detalhada. A classe **PaintPanel** estende **JPanel** e sobreporá o método **paintComponent()**. Isso permite que **PaintPanel** grave diretamente na superfície do componente quando a pintura ocorre. O construtor de **PaintPanel** usa dois parâmetros que especificam a largura e altura ideais do painel. Também especifica uma borda vermelha de 1 pixel de largura. Ele faz isso usando a chamada a **setBorder()** a seguir:

```
| setBorder(BorderFactory.createLineBorder(Color.RED, 1));
```

O método **setBorder()** é definido por **JComponent**. Ele anexa uma borda ao componente. Uma borda é um objeto de tipo **Border**, definido em **javax.swing.border**. O Swing fornece várias bordas internas diferentes, que podem ser obtidas com o uso dos métodos estáticos de **BorderFactory** do pacote **javax.swing**. A usada aqui é uma borda de linha simples. Ela é criada com este método:

```
static Border createLineBorder(Color cor, int largura)
```

A cor da borda é especificada por *cor*. Você pode projetar suas próprias cores, porém é mais provável que use uma das constantes definidas pela classe **Color**, como **Color.RED**, **Color.Blue** ou **Color.GREEN**. A classe **Color** fica no pacote **java.awt**. A largura da borda é passada em *largura*.

Dentro da sobreposição de **paintComponent()**, observe que primeiro ela chama **super.paintComponent()**. Como explicado, isso é necessário para assegurar que o componente seja desenhado apropriadamente. Em seguida, a largura e a altura do painel são obtidas junto com os espaços adicionais. Esses valores são usados para restringir a altura e a posição das barras do gráfico dentro da área de desenho do painel. A área de desenho é a largura e a altura gerais de um componente menos as larguras da borda. Os cálculos são projetados para operar com **PaintPanels** e bordas de tamanhos diferentes.

Agora, **paintComponent()** usa a classe **Random** para obter números pseudoaleatórios. (Os números pseudoaleatórios criam uma sequência de valores aparentemente aleatórios.) Essa classe será descrita no Capítulo 24, quando as classes utilitárias Java forem examinadas. Mas, como ela é usada aqui, uma breve descrição

é necessária. **Random** é o gerador Java de números aleatórios. Você pode obter os valores gerados por **Random** de várias maneiras. Nesse caso, seu método **nextInt()** é chamado. Ele tem duas formas. A usada no programa é esta:

```
int nextInt(int max)
```

Nela, *max* especifica um limite superior para o valor. O método retornará um inteiro pseudoaleatório do intervalo zero a *max*-1. O limite usado pelo programa é

```
| height-ins.bottom
```

Esse limite assegura que o valor pseudoaleatório esteja dentro do limite inferior do painel. O limite superior é definido por esta linha:

```
| if(h <= ins.top) h = ins.top+1;
```

Ela assegura que a linha não seja desenhada na borda superior.

A linha é de fato desenhada, usando esta instrução:

```
| g.drawLine(i, height-ins.bottom, i, h);
```

Não esqueça que o sistema de coordenadas começa no canto superior esquerdo. Logo, as coordenadas podem parecer invertidas até você lembrar desse detalhe.

O método **changeBorderSize()** altera o tamanho da borda do painel. Uma vez que **paintComponent()** exibe automaticamente dados que preenchem o tamanho atual da área de desenho do painel, a largura da borda pode ser alterada sem nenhum problema.

A classe **PaintDemo** demonstra **PaintPanel**. Ela cria um **PaintPanel** chamado **pp** com as dimensões 100 por 100. Também cria dois botões chamados **jbtnMore** e **jbtnSize**. Quando **jbtnMore** é clicado, **repaint()** é chamado em **pp**. Isso resulta em uma chamada ao método **paintComponent()** de **PaintPanel**. Logo, sempre que **jbtnMore** é pressionado, novos dados gerados aleatoriamente são exibidos. Quando **jbtnSize** é clicado, a largura da borda é alternada entre 1 e 5. Como **paintComponent()** calcula os limites da área de desenho dinamicamente, a borda pode ser configurada com qualquer valor arbitrário válido.

Verificação do progresso

1. Para gravar a saída diretamente na área de desenho de um componente do Swing, que método você deve sobrepor?
2. Que método obtém a largura da borda atual?
3. Como fazer o método **paintComponent()** ser chamado?

Respostas:

1. **paintComponent()**
2. **getInsets()**
3. Para fazer **paintComponent()** ser chamado, é preciso chamar **repaint()**.

EXERCÍCIOS

1. Qualquer código que afete um componente do Swing deve ser executado na thread _____.
2. Quando usamos **javax.swing.Timer**, que evento é gerado quando o temporizador expira?
3. Que métodos iniciam e interrompem **javax.swing.Timer**?
4. O que é **SwingWorker**?
5. Que método um applet deve usar para criar sua GUI?
6. Seu programa pode pintar diretamente na superfície de um componente?
7. Quais são os três métodos de pintura chamados quando um componente do Swing deve gerar sua exibição?
8. Que métodos obtêm a largura e a altura atuais de um componente?
9. Refaça o applet da seção Tente isto 21-1 de modo que a direção da rolagem se inverta em intervalos periódicos. Por exemplo, faça o texto rolar para a esquerda durante algum tempo e depois para a direita por um determinado período, e assim por diante. A duração vai depender de você, mas a resposta mostrada no Apêndice C inverte a direção da rolagem a cada 20 segundos.
10. Crie um programa que exiba uma janela contendo a figura de um semáforo, semelhante ao mostrado no Capítulo 18, exceto pelo fato de o semáforo exibir todas as três cores ao mesmo tempo. Use um retângulo de 30 pixels de largura por 100 pixels de altura e um círculo de 20 pixels de diâmetro para cada uma das três cores. Você terá que usar outro método definido na classe **Graphics**:

```
void setColor(Color cor)
```

Esse método define a cor que será usada pelo contexto gráfico para desenho futuro. Nesse exercício, use as constantes de cor **Color.BLACK**, **Color.RED**, **Color.YELLOW** e **Color.GREEN**.

11. Estenda o exercício anterior de modo que ele use um **Timer** para executar o semáforo. O semáforo deve exibir só verde por 10 segundos (durante os quais os outros dois círculos são preenchidos com branco), depois só amarelo por 5 segundos e então só vermelho por 15 segundos. Ele deve repetir esse padrão infinitamente.
12. Se **g** referencia um objeto **Graphics**, qual é a diferença, se houver, entre a chamada a **g.drawLine(10, 20, 50, 60)** e a chamada a **g.drawLine(50, 60, 10, 20)**?
13. Se **g** referencia um objeto **Graphics**, qual é a diferença, se houver, entre a chamada a **g.drawRect(10, 20, 50, 60)** e a chamada a **g.drawRect(60, 80, -50, -60)**?
14. Escreva um programa que crie uma janela com um retângulo alto e estreito sendo exibido e uma bola dentro do retângulo. A bola se move em velocidade constante para baixo no retângulo até alcançar o fundo e então inverte a direção e sobe em velocidade constante. Ela se mantém oscilando infinitamente entre a borda superior e inferior do retângulo. Use um **Timer** para mover a bola.

15. Refaça o exercício 10 do Capítulo 15 usando um **JApplet** em vez de um **Applet**. Isto é, crie um **JApplet** que desenhe uma casa. Lembre-se, usando um **JApplet**, você não deve sobrepor o método **paint()** para desenhar no applet, mas sim criar uma subclasse de **JPanel** (ou de algum outro **JComponent**), sobrepor o método **paintComponent()** na subclasse para que ele desenhe uma casa e adicionar uma instância da subclasse ao painel de conteúdo do applet.
16. O exemplo **TimerStopWatch** mostrado no capítulo cria um **JFrame** para conter o cronômetro. Modifique o exemplo para que o cronômetro fique em um **JApplet**.
17. Escreva um programa que crie uma janela contendo um **JTextArea** dentro de um **JScrollPane**. Um **JTextArea** é um componente do Swing que pode exibir várias linhas de texto. Veja um dos construtores:

```
public JTextArea(int linhas, int cols)
```

Esse construtor cria um **JTextArea** vazio com o número dado de linhas e colunas. Use-o para criar um **JTextArea** com 20 linhas e 30 colunas. Para adicionar linhas de texto a ele, use seu método **append()**, mostrado aqui:

```
void append(String texto)
```

O método acrescenta o texto fornecido ao texto que está sendo exibido. Crie uma thread separada que classifique repetidamente (infinitamente) arrays cada vez mais longos usando a classificação de bolha. Primeiro ela cria e classifica um array de tamanho 1.000, em seguida cria e classifica um array de tamanho 1.200, e assim por diante. Cada array deve ter um tamanho 200 vezes maior do que o array anterior e inicialmente deve estar em ordem inversa (do maior para o menor). A classificação de bolha o classificará do menor para o maior. A thread também deve registrar quanto tempo leva classificar o array a cada classificação. Sempre que terminar uma classificação, ela inserirá uma nova linha no fim do **JTextArea** indicando o tamanho do array e quanto tempo levou para classificá-lo.

18. Verdadeiro ou falso?
 - A. Só duas threads são usados no programa **ThreadStopWatch** do começo deste capítulo: a thread de despacho de evento e a thread que chama **updateTime()** a cada décimo de segundo.
 - B. Sempre que criamos um **JApplet**, temos que implementar para ele os métodos **start()**, **stop()**, **init()** e **destroy()**.
 - C. Na especificação da posição da figura a ser pintada em um **JComponent**, a posição (0,0) corresponde ao centro do componente.

PARTE III

Examinando a biblioteca de APIs Java

A Parte III examinará as partes-chave da biblioteca Java de Interface de Programação de Aplicativos (API). A biblioteca de APIs contém as classes e interfaces padrão fornecidas com o JDK. Como era de se esperar, ela é muito grande e uma verificação completa de seu conteúdo não faz parte do escopo deste livro. Em vez disso, enfocaremos vários elementos básicos da biblioteca que são muito usados na criação de programas Java e que não foram descritos em outro local do livro. O Capítulo 22 examinará a manipulação de strings via classe **String**. O Capítulo 23 descreverá as partes-chave do pacote **java.lang**. O Capítulo 24 examinará várias partes de **java.util**. O Capítulo 25 discutirá o **Collections Framework**. O Capítulo 26 examinará os fundamentos da rede. O Capítulo 27 conclui o livro examinando a API de concorrência, inclusive o Framework Fork/Join.

Além dos pacotes descritos aqui, outra parte essencial da API Java fica no pacote **java.io**. Ele foi abordado no Capítulo 11 e não será repetido agora.

***Nota:** O material apresentado aqui e em outros locais do livro oferece visões gerais e resumos de partes da API Java. Uma descrição completa da biblioteca de APIs pode ser encontrada na documentação Java, que está disponível online na Oracle. Além disso, inovação e evolução fazem parte da história de Java. Portanto, é possível que acréscimos ou alterações tenham sido feitos na biblioteca de APIs desde que este livro foi escrito.*

Manipulação de strings

PRINCIPAIS HABILIDADES E CONCEITOS

- Os construtores de **String**
- Três recursos da linguagem relacionados a strings
- Sobrepor **toString()**
- Usar **length()**
- Obter os caracteres de uma string
- Comparar strings
- Usar **indexOf()** e **lastIndexOf()**
- Obter um string modificado
- Mudar maiúsculas e minúsculas em um string
- Entender como **StringBuffer** e **StringBuilder** estão relacionadas a **String**

Uma das classes Java mais usadas é **String**. Isso ocorre porque a manipulação de strings é parte integrante de muitos programas Java. Um resumo da classe **String** foi apresentado no Capítulo 5; agora, ela será examinada com mais detalhes. É importante mencionar desde já que **String** dá suporte a um amplo conjunto de funcionalidades e não é possível examinarmos todos os seus aspectos aqui. Nossa objetivo é fornecer uma abordagem geral e um conhecimento prático sólido de seus principais recursos. **String** é uma classe que você deve reexaminar de tempos em tempos ao avançar em seu estudo de Java.

A classe **String** fica no pacote **java.lang**. Logo, está disponível automaticamente para todos os programas. Embora o próximo capítulo examine outros aspectos de **java.lang**, devido à importância e ao uso disseminado de **String**, ela merece um capítulo inteiro. Começaremos examinando os aspectos básicos de **String**.

ASPECTOS BÁSICOS DOS STRINGS

Como vimos no Capítulo 5, em Java um string é uma sequência de caracteres. Mas diferentemente de outras linguagens que implementam strings como arrays de caracteres, Java implementa strings como objetos de tipo **String**. A implementação de strings como objetos permite a Java fornecer um conjunto de recursos que torna a manipulação de strings conveniente. Por exemplo, Java tem métodos para a comparação de dois strings, a busca de um substring, a concatenação de strings e a alteração

da caixa dentro de um string. Além disso, objetos **String** podem ser construídos de várias maneiras, o que facilita a obtenção de um string quando necessário.

A classe **String** implementa as interfaces a seguir: **Comparable<String>**, **CharSequence** e **Serializable**. A interface **Comparable** especifica como os objetos são comparados. **CharSequence** define um conjunto de métodos que são aplicáveis a uma sequência de caracteres. **Serializable** apenas indica que o estado de um **String** pode ser salvo e restaurado com o uso do mecanismo Java de serialização.

Quando você criar um objeto **String**, estará criando um string que não pode ser alterado. Isto é, uma vez que um objeto **String** for construído, não há como alterar os caracteres que compõem esse string. Isso pode parecer uma restrição grave, mas, como explicado no Capítulo 5, não é esse o caso. Você continua podendo executar todos os tipos de operações com strings. A diferença é que sempre que precisar de uma versão alterada de um string existente, um novo objeto **String** será criado contendo as modificações. O string original é deixado inalterado. Essa abordagem é usada porque strings fixos que não podem ser alterados são implementados com mais eficiência do que os alteráveis.

É importante enfatizar que quando dizemos que os strings existentes dentro de objetos de tipo **String** são inalteráveis, queremos dizer que o conteúdo de uma instância de **String** não pode ser alterado depois de criado. No entanto, uma variável declarada como uma referência **String** pode ser alterada para referenciar um objeto **String** diferente.

***Nota:** Para os casos em que for desejável um string modificável, Java fornece duas opções: **StringBuffer** e **StringBuilder**. Elas são mencionadas brevemente no fim deste capítulo. Ambas contêm strings que podem ser modificados após serem criados.*

A classe **String** é declarada como **final**, ou seja, não pode ter subclasses. Isso permite que ocorram certas otimizações que aumentam o desempenho em operações comuns de strings.

OS CONSTRUTORES DE STRING

A classe **String** dá suporte a um grande número de construtores. Veremos vários exemplos. Para criar um **String** vazio, é só chamar o construtor padrão. Por exemplo,

```
|String str = new String();
```

criará uma instância de **String** sem caracteres.

Geralmente criamos strings com valores iniciais. A classe **String** fornece vários construtores que atendem a essa tarefa. Por exemplo, para criar um **String** inicializado por um array de caracteres, use o construtor abaixo:

```
String(char[ ] cars)
```

Você pode especificar um subconjunto de um array de caracteres como inicializador usando o construtor a seguir:

```
String(char[ ] cars, int índiceInicial, int numCars)
```

Aqui, *índiceInicial* especifica o índice em que o subconjunto começa, e *numCars* especifica o número de caracteres a serem usados.

Na construção de um **String** a partir de um array, o conteúdo do array é copiado no **String**. Em outras palavras, o array não fica subjacente à instância de **String**. Ou seja, os dois ficam separados. Portanto, após o **String** ser construído, a alteração do array não altera o conteúdo do string.

Você pode construir um objeto **String** contendo a mesma sequência de caracteres de outro objeto **String** usando este construtor:

`String(String objStr)`

Aqui, *objStr* é um objeto **String**.

Vejamos um programa que coloca os construtores anteriores em ação. Em cada caso, observe como uma instância de **String** é construída.

```
// Demonstra vários construtores de String.

class StringConsDemo {

    public static void main(String[] args) {

        char[] digits = new char[16];

        // Cria um array contendo os dígitos de 0 a 9
        // mais os valores hexadecimais de A a F.
        for(int i=0; i < 16; i++) {
            if(i < 10) digits[i] = (char) ('0'+i);
            else digits[i] = (char) ('A' + i - 10);
        }

        // Cria um string contendo todo o array.
        String digitsStr = new String(digits); ←
        System.out.println(digitsStr);

        // Cria um string contendo uma parte do array.
        String nineToTwelve = new String(digits, 9, 4); ←
        System.out.println(nineToTwelve);

        // Constrói um string a partir de outro.
        String digitsStr2 = new String(digitsStr); ←
        System.out.println(digitsStr2);

        // Agora, cria um string vazio.
        String empty = new String(); ←

        // Esta linha não exibirá nada:
        System.out.println("Empty string: " + empty);
    }
}
```

Constrói strings de várias maneiras.

A saída é mostrada aqui:

```
0123456789ABCDEF
9ABC
0123456789ABCDEF
Empty string:
```

String também fornece vários construtores que inicializam um string dado um array de **bytes**. Veja dois exemplos:

```
String(byte[ ] cars)
String(byte[ ] cars, int índiceInicial, int numCars)
```

Nesse caso, *cars* especifica o array de bytes. A segunda forma permite a especificação de um subconjunto. Nos dois construtores, a conversão de byte para caractere é feita com o uso do mapeamento de caracteres padrão. (Um mapeamento de caracteres define o relacionamento entre bytes e caracteres.) Também são definidas versões estendidas dos construtores de conversão de bytes em strings em que podemos especificar o mapeamento de caracteres. Como vimos na discussão sobre I/O do Capítulo 11, no nível mais baixo o I/O é orientado a bytes. Logo, os construtores de arrays de **bytes** podem ser especialmente úteis na construção de um string a partir de entradas fornecidas por um fluxo de bytes.

Além de usar os construtores que acabamos de descrever, você pode construir um **String** a partir de um objeto **StringBuilder** ou **StringBuffer**. Também pode construir um **String** a partir de um array de pontos de código Unicode.

Verificação do progresso

1. Quais são as três maneiras de passar caracteres como argumentos para construtores de **String**?
2. Um string Java é mutável. Verdadeiro ou falso?
3. Você precisa importar um pacote explicitamente para usar strings em Java. Verdadeiro ou falso?

TRÊS RECURSOS DA LINGUAGEM RELACIONADOS A STRINGS

Já que os strings são uma parte comum e importante da programação, Java dá suporte a três recursos de strings especialmente úteis diretamente dentro da sintaxe da linguagem. Eles são a criação automática de novas instâncias de **String** a partir de literais de string, a concatenação de vários objetos **String** com o uso do operador + e a conversão de outros tipos de dados em uma representação na forma de string. Há métodos explícitos disponíveis para a execução de todas essas funções, mas o suporte interno de Java as torna mais convenientes.

Respostas:

1. Em arrays de caracteres, arrays de bytes e outros strings.
2. Falso.
3. Falso. O pacote **java.lang**, que contém a classe **String**, é importado automaticamente.

Literais de strings

Como você sabe, um literal de string é criado com a especificação de um string entre aspas. Para cada literal de string de seu programa, Java construirá automaticamente um objeto **String**. Logo, você pode usar um literal de string para inicializar um objeto **String**. Por exemplo, o código a seguir cria um objeto **String** contendo “this is an example string literal”.

```
|String str = "this is an example string literal"; // usa um literal de string
```

Nesse caso, **str** recebe uma referência ao objeto que representa o string entre aspas.

Já que um objeto **String** é criado para cada literal de string, você pode usar um literal de string em qualquer local em que puder usar uma instância de **String**. Por exemplo, você pode passar um literal de string como argumento para um método que esteja esperando um objeto **String**. Portanto, dado um método chamado **myMethod()** declarado desta forma:

```
|void myMethod(String arg) { ...
```

a chamada a seguir é válida:

```
|myMethod("a string literal");
```

Concatenação de strings

Em geral, Java não permite operações de objetos **String** com operadores. Uma exceção a essa regra é o operador **+**, que concatena dois strings, produzindo um objeto **String** como resultado. Isso nos permite encadear uma série de operações **+**. Por exemplo, o fragmento a seguir concatena três strings:

```
|String age = "19";
String str = "He is " + age + " years old.";
System.out.println(str);
```

Esse código exibe o string “He is 19 years old”.

A concatenação de strings pode ser útil na criação de strings muitos longos. Em vez de permitir que strings longos se estendam pelo código-fonte, podemos dividi-los em partes menores, usando **+** para concatená-los. Veja um exemplo:

```
// Usando a concatenação para evitar linhas longas.
class ConCat {
    public static void main(String[] args) {
        String longStr = "This could have been " +
            "a very long line that would have " +
            "wrapped around. But string concatenation " +
            "prevents this.";

        System.out.println(longStr);
    }
}
```

Concatenação de strings com outros tipos de dados

Você pode concatenar strings com outros tipos de dados. Por exemplo, considere esta versão um pouco diferente do exemplo anterior:

```
| int age = 19;
| String str = "He is " + age + " years old.";
| System.out.println(str);
```

Nesse caso, **age** é um **int** em vez de outro **String**, mas a saída produzida é a mesma de antes. Isso ocorre porque o valor **int** de **age** é convertido automaticamente em sua representação na forma de string como um objeto **String**. Esse string é então concatenado como antes. O compilador converterá um operando em seu equivalente na forma de string sempre que o outro operando de + for uma instância de **String**.

No entanto, cuidado ao combinar outros tipos de operações com expressões de concatenação de strings. Você pode obter resultados inesperados. Considere o código a seguir:

```
| String str = "four: " + 2 + 2;
| System.out.println(str);
```

Esse fragmento exibe

```
| four: 22
```

em vez de

```
| four: 4
```

que era o esperado. Veja por quê. A concatenação de “four:” com o string equivalente a 2 ocorre primeiro. Esse resultado é então concatenado com o string equivalente a 2 uma segunda vez. Para concluir a adição de inteiros primeiro, você deve usar parênteses, desta forma:

```
| String str = "four: " + (2 + 2);
```

Agora **str** contém o string “four: 4”.

Sobrepondo **toString()**

Quando Java converte um objeto em sua representação na forma de string durante a concatenação, faz isso chamando o método **toString()** do objeto. Como você sabe, todas as classes implementam o método **toString()** porque ele é definido por **Object**. Sua forma geral é:

String **toString()**

O método retorna um string que descreve o objeto. Embora a classe **Object** forneça uma implementação padrão de **toString()**, ela pode não ser adequada a alguns casos. Na verdade, para as classes mais importantes que criar, você vai querer sobrepor **toString()** e fornecer sua própria representação na forma de string para os objetos da classe. Felizmente, é fácil fazer isso. Ao sobrepor **toString()** para sua classe, simplesmente retorne um objeto **String** contendo um string legível por humanos que descreva apropriadamente um objeto da classe. Quando sobrepuiser **toString()** para as classes que criar, você estará permitindo que elas se integrem totalmente ao ambiente de programação Java. Por exemplo, elas poderão ser usadas significativamente em instruções **print()** e **println()** e em expressões de concatenação. O programa a seguir demonstra isso sobrepondo **toString()** para a classe **Box**:

```
// Sobrepõe toString() para a classe Box.
class Box {
    double width;
    double height;
    double depth;

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // Fornece a representação de Box na forma de string.
    public String toString() { ← Sobrepõe toString() para Box.
        return "Dimensions are " + width + " by " +
               depth + " by " + height + ".";
    }
}

class OverrideToString {
    public static void main(String[] args) {
        Box b = new Box(10, 12, 14);
        String str = "Box b: " + b; // toString() é chamado aqui

        System.out.println(b); // toString() é chamado aqui
        System.out.println(str);
    }
}
```

A saída desse programa é mostrada aqui:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

Como você pode ver, o método **toString()** de **Box** é chamado automaticamente quando um objeto **Box** é usado em uma expressão de concatenação ou em uma chamada a **println()**.

Pergunte ao especialista

P Por que alguém criaria um string vazio? Uma vez que os strings são imutáveis, não podemos adicionar novos caracteres a eles. Ou seja, um string vazio estará sempre vazio. Pode dar um exemplo que use um string vazio?

R Sim. Um string vazio é útil em um laço em que strings sejam concatenados. Nesse caso, com frequência o string vazio é o string inicial apropriado. Veja um exemplo:

```
String[] data = {"abc", "def", "ghi"};
String result = "";
```

```

| for(String s : data)
|   result += s;

```

Observe que a variável de string **result** é inicializada com um string vazio. Assim, após a primeira concatenação, **result** conterá apenas o string “abc” porque ele foi concatenado com o string vazio. Ou seja, o string vazio fornece a condição inicial apropriada.

TENTE ISTO 22-1 Adicione um método `toString()` à classe `GenSimpleStack`

`GenStackToStringDemo.java`
`GenSimpleStack.java`

Como mencionado anteriormente, para muitas classes que criar, você vai querer sobrepor o método `toString()` herdado de `Object` para que ele retorne um string significativo para um objeto da classe. Neste projeto você aplicará esse princípio à classe `GenSimpleStack` criada na seção Tente isto 14-1 do Capítulo 14.

Primeiro verá qual versão de `toString()` herdada de `Object` retorna para `GenSimpleStack`. Em seguida, criará uma sobreposição de `toString()` para `GenSimpleStack` que retorne um resultado mais útil.

PASSO A PASSO

1. A primeira etapa é criar um programa que exiba o resultado da chamada a `toString()` em um objeto `GenSimpleStack`. Insira o código a seguir em um arquivo chamado **GenStackToStringDemo.java** e salve o arquivo no mesmo diretório dos arquivos Java das classes `StackFullException`, `StackEmptyException` e `GenSimpleStack` e da interface `IGenSimpleStack` da seção Tente isto 14-1.

```

class GenStackToStringDemo {
    public static void main(String[] args)
        throws StackFullException, StackEmptyException {
        Integer iStore[] = new Integer[10];
        GenSimpleStack<Integer> stack =
            new GenSimpleStack<Integer>(iStore);

        System.out.println("Empty stack: " + stack);

        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);
        stack.pop();

        System.out.println("Non-empty stack: " + stack);
    }
}

```

2. Antes de sobrepor **toString()** para a classe **GenSimpleStack**, compile o programa da Etapa 1 e execute-o. Você verá uma saída semelhante à mostrada a seguir, embora deva ver caracteres diferentes após o caractere “@”.

```
| Empty stack: GenSimpleStack@7369ca65
| Non-empty stack: GenSimpleStack@7369ca65
```

Esse strings são gerados pelo método **toString** da classe **Object**. Observe que a primeira parte da string é o nome da classe, a última parte é a sequência de caracteres hexadecimais e há um caractere “@” separando-as. Esse string é útil porque nos informa a classe do objeto, mas na maioria dos aplicativos, seria interessante termos um string mais significativo.

3. Adicione o método a seguir à classe **GenSimpleStack** da seção Tente isto 14-1. Esse método sobrepõe **toString()** para que retorne um resultado mais significativo. Compile a classe.

```
// Sobrepõe toString() para GenSimpleStack.
public String toString() {
    String result = "(";

    // Adiciona a representação na forma de string de
    // todos os itens da pilha, separados por vírgulas.
    for(int i = 0; i < tos; i++) {
        result += data[i];
        if(i < tos-1) // se não for o último item,
            // adiciona uma vírgula e um espaço
        result += ", ";
    }

    // Adiciona o parêntese direito e o retorna.
    result += ")";
    return result;
}
```

4. Compile e execute o programa **GenStackToStringDemo** novamente. Você verá a saída a seguir:

```
| Empty stack: ()
| Non-empty stack: (1, 2, 3)
```

Os elementos da pilha são listados dentro de parênteses e separados por vírgulas, com o topo da pilha no lado direito. Esses strings são mais significativos do que os gerados pela versão de **toString()** de **Object**.

Observe que, no laço **for**, o método **toString** que você criou chama implicitamente os métodos **toString()** de todos os itens da pilha. Portanto, se todas as classes fornecerem métodos **toString()** úteis, até mesmo objetos complexos como pilhas contendo objetos de outras classes poderão gerar representações significativas na forma de strings.

5. Para sua conveniência, aqui está a classe **GenSimpleStack** completa, incluindo a sobreposição de **toString()**:

```
|class GenSimpleStack<T> implements IGenSimpleStack<T> {
|    private T[] data; // este array contém a pilha
|    private int tos; // índice do topo da pilha
|
|    // Constrói uma pilha vazia com o array dado como espaço de
|    // armazenamento.
|    GenSimpleStack(T[] arrayRef) {
|        data = arrayRef;
|        tos = 0;
|    }
|
|    // Insere um item na pilha.
|    public void push(T obj) throws StackFullException {
|        if (isFull())
|            throw new StackFullException(data.length);
|
|        data[tos] = obj;
|        tos++;
|    }
|
|    // Extrai um item da pilha.
|    public T pop() throws StackEmptyException {
|        if (isEmpty())
|            throw new StackEmptyException();
|
|        tos--;
|        return data[tos];
|    }
|
|    // Sobrepõe toString() para GenSimpleStack.
|    public String toString() {
|        String result = "(";
|
|        // Adiciona a representação na forma de string de
|        // todos os itens da pilha, separados por vírgulas.
|        for (int i = 0; i < tos; i++) {
|            result += data[i];
|            if (i < tos - 1) // se não for o último item,
|                // adiciona uma vírgula e um espaço
|            result += ", ";
|        }
|
|        // Adiciona o parêntese direito e o retorna.
|        result += ")";
|        return result;
|    }
|}
```

```

    // Retorna true se a pilha estiver vazia.
    public boolean isEmpty() {
        return tos == 0;
    }

    // Retorna true se a pilha estiver cheia.
    public boolean isFull() {
        return tos == data.length;
    }
}

```

O MÉTODO length()

Como os strings não são arrays, eles não têm um campo **length**. No entanto, **String** tem um método chamado **length()** que retorna o tamanho de um string. O tamanho do string é o número de caracteres que ele contém. O método **length()** é mostrado aqui:

```
int length()
```

Por exemplo,

```

String str = "Theta";
System.out.println(str.length());

```

exibe 5, porque há cinco caracteres em **str**. Como demonstrado na próxima seção, **length()** facilita a execução de operações com os caracteres individuais de um string.

Verificação do progresso

1. Para obter o número de caracteres de um string, você deve usar o campo **length** do string, como faria com um array. Verdadeiro ou falso?
2. A maioria das classes deve sobrepor o método _____ para criar representações significativas de suas instâncias na forma de strings.
3. O que o operador + faz quando seus operandos são strings?

OBTENDO OS CARACTERES DE UM STRING

A classe **String** fornece três maneiras pelas quais podem ser obtidos os caracteres (isto é, valores de tipo **char**) de um objeto **String**. Examinaremos cada uma agora. Embora os caracteres que compõem um string não possam ser indexados com se

Respostas:

1. Falso. Você deve usar o método **length()** da classe **String**.
2. **toString()**
3. Concatena os strings.

fossem um array de caracteres, vários métodos de **String** empregam um índice (em outras palavras, um deslocamento) no string em suas operações. Como nos arrays, os índices dos strings começam em zero.

charAt()

Para obter um caractere individual de um **String**, você pode usar o método **charAt()**. Ele tem esta forma geral:

```
char charAt(int onde)
```

Aqui, *onde* é o índice do caractere que você deseja obter. O valor de *onde* deve ser não negativo e especificar um local dentro do string. O método **charAt()** retorna o caractere do local especificado.

O programa a seguir mostra **charAt()** em ação. Ele é usado para exibir um string empregando espaçamento duplo. No processo, também demonstra o método **length()**.

```
// Demonstra charAt() e length().  
  
class CharAtAndLength {  
    public static void main(String[] args) {  
        String str = "Programming is both art and science."  
  
        // Percorre todos os caracteres do string.  
        for(int i=0; i < str.length(); i++)  
            System.out.print(str.charAt(i) + " ");  
  
        System.out.println();  
    }  
}
```

A saída é mostrada abaixo:

```
|P r o g r a m m i n g   i s   b o t h   a r t   a n d   s c i e n c e .
```

No programa, observe como **length()** é usado para controlar o número de iterações e **charAt()** é usado para obter cada caractere em sequência.

getChars()

Se você tiver que obter mais de um caractere de uma só vez, pode usar o método **getChars()**. Ele tem esta forma geral:

```
void getChars(int inícioOrigem, int fimOrigem, char[ ] destino,  
            int inícioDestino)
```

Aqui, *inícioOrigem* especifica o índice do começo do substring a ser obtido, e *fimOrigem* especifica um índice que fica uma unidade após o fim do substring desejado. Logo, o substring contém os caracteres de *inícioOrigem* a *fimOrigem*-1. O array que receberá os caracteres é especificado por *destino*. O índice de *destino* em que o substring será copiado é passado em *inícioDestino*. Tenha o cuidado de assegurar que o array *destino* seja suficientemente grande para conter o número de caracteres do substring especificado.

O programa a seguir demonstra **getChars()**:

```
class GetCharsDemo {
    public static void main(String[] args) {
        String str = "Programming is both art and science.";
        int start = 15;
        int end = 23;
        char[] buf = new char[end - start];

        str.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

Esta é a saída do programa:

```
|both art
```

toCharArray()

Se você quiser converter todos os caracteres de um objeto **String** em um array de caracteres, a maneira mais fácil é chamar **toCharArray()**. Ele retorna um array de caracteres contendo todo o string. Sua forma geral é:

```
char[ ] toCharArray()
```

Por exemplo, veja como obter um array com o conteúdo inteiro de **str**:

```
String str = "Programming is both art and science.";
char[] chrs = str.toCharArray();
System.out.println(chrs);
```

Após essa sequência ser executada, **chrs** conterá os caracteres que estavam em **str**, e a instrução **println()** exibirá

```
Programming is both art and science.
```

Mais uma coisa: o método **toCharArray()** é essencialmente uma conveniência, já que é possível usar **getChars()** para obter o mesmo resultado.

Pergunte ao especialista

P Anteriormente você mencionou que um literal de string pode ser usado em qualquer local em que puder ser usada uma instância de **String**. Ou seja, posso chamar um método de **String** diretamente em um literal de string?

R Sim, você pode chamar métodos diretamente em um string entre aspas. Por exemplo, a instrução a seguir chama **length()** no literal de string “mystring”.

```
| System.out.println("mystring".length());
```

Como esperado, ela exibe 8. Veja outro exemplo:

```
| System.out.println("Code and Data".charAt(9));
```

Essa instrução exibe o caractere D.

Verificação do progresso

1. Para obter o sexto caractere de um string `s`, que chamada de método você deve fazer?
2. O método `getChars()` retorna um novo array de caracteres. Verdadeiro ou falso?
3. Qualquer modificação no array `char` retornado por `s.toCharArray()` causará uma modificação correspondente no string `s`. Verdadeiro ou falso?

COMPARAÇÃO DE STRINGS

A classe **String** tem muitos métodos que comparam strings ou substrings. Vários exemplos serão descritos agora.

`equals()` e `equalsIgnoreCase()`

Para comparar dois strings e ver se são iguais, use o método `equals()`. Ele tem esta forma geral:

```
boolean equals(Object str)
```

Aqui, `str` é o objeto que está sendo comparado com o objeto **String** chamador. Esse método é especificado por **Object** e sobreposto por **String**. O método `equals()` retorna **true** quando os strings têm os mesmos caracteres na mesma ordem. Ele retorna **false** quando os strings diferem ou quando `str` não é um **String**. A comparação diferencia maiúsculas de minúsculas.

Para fazer uma comparação sem considerar maiúsculas e minúsculas (isto é, uma comparação que ignora diferenças na caixa dos caracteres), chame o método `equalsIgnoreCase()`. Quando ele compara dois strings, considera **A-Z** igual a **a-z**, por exemplo. Sua forma geral é:

```
boolean equalsIgnoreCase(String str)
```

Nela, `str` é o objeto que está sendo comparado com o objeto **String** chamador. O método retorna **true** quando, independentemente de diferenças de caixa, os strings contêm os mesmos caracteres na mesma ordem; caso contrário, retorna **false**.

Veja um exemplo que demonstra `equals()` e `equalsIgnoreCase()`:

```
// Demonstra equals() e equalsIgnoreCase().

class EqualityDemo {
    public static void main(String[] args) {
        String str1 = "table";
        String str2 = "table";
        String str3 = "chair";
        String str4 = "TABLE";
```

Respostas:

1. `s.charAt(5)`
2. Falso. Ele insere caracteres em um array fornecido como um de seus argumentos.
3. Falso.

```

if(str1.equals(str2))
    System.out.println(str1 + " equals " + str2);
else
    System.out.println(str1 + " does not equal " + str2);

if(str1.equals(str3))
    System.out.println(str1 + " equals " + str3);
else
    System.out.println(str1 + " does not equal " + str3);

if(str1.equals(str4)) ←
    System.out.println(str1 + " equals " + str4);
else
    System.out.println(str1 + " does not equal " + str4); ←
        Estas duas
        comparações
        produzem
        resultados
        diferentes.

if(str1.equalsIgnoreCase(str4)) ←
    System.out.println("Ignoring case differences, " + str1 +
                       " equals " + str4);
else
    System.out.println(str1 + " does not equal " + str4);
}
}

```

A saída do programa é mostrada abaixo:

```

table equals table
table does not equal chair
table does not equal TABLE
Ignoring case differences, table equals TABLE

```

equals() versus ==

Antes de prosseguir, é importante enfatizar que o método **equals()** e o operador **==** executam duas operações diferentes. Como acabamos de explicar, o método **equals()** compara os caracteres existentes dentro de um objeto **String**. O operador **==** compara duas referências de objeto para ver se elas referenciam a mesma instância. Logo, eles executam duas operações essencialmente diferentes. O programa a seguir mostra como dois objetos **String** diferentes podem conter os mesmos caracteres sem que referências a eles sejam consideradas iguais:

```

// equals vs ==
class EqualsNotEqualTo {
    public static void main(String[] args) {
        String str1 = "Alpha";
        String str2 = new String(str1);

        System.out.println(str1 + " equals " + str2 + " is " +
                           str1.equals(str2));
        System.out.println(str1 + " == " + str2 + " is " + (str1 == str2));
    }
}

```

A variável **str1** referencia a instância de **String** criada pelo literal “**Alpha**”. O objeto referenciado por **str2** é criado com **str1** como inicializador. Logo, o conteúdo dos

dois objetos **String** é idêntico, mas os objetos são distintos. Ou seja, **str1** e **str2** não referenciam os mesmos objetos e, portanto, não são iguais, como mostrado pela saída do programa:

```
| Alpha equals Alpha is true
| Alpha == Alpha is false
```

regionMatches()

O método **regionMatches()** compara um subconjunto (isto é, uma região) de um string com o subconjunto de outro. Há uma forma sobrecarregada que nos permite ignorar a caixa dos caracteres na comparação. Aqui estão as formas gerais desses dois métodos:

```
boolean regionMatches(int índiceInicial, String str2,
                      int índiceInicialStr2, int numCars)

boolean regionMatches(boolean ignorarCaixa,
                      int índiceInicial, String str2,
                      int índiceInicialStr2, int numCars)
```

Nas duas versões, *índiceInicial* especifica o índice em que começa a região a ser comparada dentro do objeto **String** chamador. O objeto **String** que está sendo comparado é especificado por *str2*. O índice em que a comparação começará dentro de *str2* é especificado por *índiceInicialStr2*. O tamanho da região que está sendo comparada é passado em *numCars*. A primeira versão diferencia maiúsculas de minúsculas. Na segunda versão, se *ignorarCaixa* for **true**, a caixa dos caracteres será ignorada. Caso contrário, será considerada.

Veja um exemplo:

```
// Demonstra RegionMatches.

class CompareRegions {
    public static void main(String[] args) {
        String str1 = "Standing at river's edge.";
        String str2 = "Running at river's edge.";

        if(str1.regionMatches(9, str2, 8, 12))
            System.out.println("Regions match.");
        else
            System.out.println("Regions do not match.");
    }
}
```

A saída é

```
| Regions match.
| Regions do not match.
```

startsWith() e endsWith()

String define dois métodos que são, mais ou menos, formas especializadas de **regionMatches()**. O método **startsWith()** determina se um determinado objeto **String** começa com um string especificado. Inversamente, **endsWith()** determina se o obje-

to **String** em questão termina com um string especificado. Eles têm as formas gerais a seguir:

```
boolean startsWith(String str)
boolean endsWith(String str)
```

Aqui, *str* é o string que está sendo verificado. No caso de **startsWith()**, se *str* coincidir com o começo do string chamador, **true** será retornado. No caso de **endsWith()**, **true** será retornado se *str* coincidir com o final do string chamador. Caso não haja correspondência, os dois métodos retornarão **false**. As comparações diferenciam maiúsculas de minúsculas.

Por exemplo, dada a instrução

```
| String str = "Status: Complete";
```

as duas expressões a seguir são verdadeiras:

```
| str.startsWith("Status:")
e
| str.endsWith("Complete")
```

Há uma segunda forma de **startsWith()** que permite a especificação de um ponto de partida. Ela é mostrada abaixo:

```
boolean startsWith(String str, int índiceInicial)
```

Aqui, *índiceInicial* especifica o índice do string chamador em cuja posição a pesquisa começará.

compareTo() e **compareIgnoreCase()**

Geralmente, é suficiente saber apenas se dois strings, ou partes de dois strings, são idênticas. No caso de classificação, você tem que saber qual é *menor que*, *igual a* ou *maior que* a outra. Um string é menor do que outro se vier antes neste na ordem alfabética. Um string é maior do que outro se vier depois na ordem alfabética. O método **compareTo()** serve a essa finalidade. Ele é especificado pela interface **Comparable<T>**, que implementa **String**. Sua forma geral é:

```
int compareTo(String str)
```

Aqui, *str* é o **String** que está sendo comparado com o **String** chamador. A comparação diferencia maiúsculas de minúsculas. O resultado da comparação é retornado e é interpretado, como mostrado abaixo:

Valor	Significado
Menor que zero	O string chamador é menor do que <i>str</i> .
Maior que zero	O string chamador é maior do que <i>str</i> .
Zero	Os dois strings são iguais.

Se quiser fazer uma comparação que não diferencie maiúsculas de minúsculas, use **compareIgnoreCase()**, como mostrado a seguir:

```
int compareIgnoreCase(String str)
```

Esse método retorna os mesmos resultados de `compareTo()`, exceto por ignorar diferenças na caixa das letras.

O próximo exemplo mostra a diferença entre `compareTo()` e `compareToIgnoreCase()`.

```
// Demonstra compareTo() e compareToIgnoreCase().

class CompareStrings {
    public static void main(String[] args) {
        String str1 = "alpha";
        String str2 = "ALPHA";
        String str3 = "Beta";

        int result;

        // Demonstra as diferenças entre compareTo()
        // e compareToIgnoreCase().
        result = str1.compareTo(str2);
        if(result != 0)
            System.out.println("Using compareTo(): " +
                               str1 + " and " + str2 + " differ");

        result = str1.compareToIgnoreCase(str2);
        if(result == 0)
            System.out.println("Using compareToIgnoreCase(): " +
                               str1 + " and " + str2 + " are the same\n");

        // Agora, compara alpha com Beta usando compareTo().
        System.out.println("Using compareTo() to compare " + str1 +
                           " with " + str3);
        result = str1.compareTo(str3); ←
        if(result < 0)
            System.out.println(str1 + " is less than " + str3);
        else if(result == 0)
            System.out.println(str1 + " is equal to " + str3);
        else if(result > 0)
            System.out.println(str1 + " is greater than " + str3);

        System.out.println();

        // Em seguida, compara alpha com Beta usando compareToIgnoreCase().
        System.out.println("Using compareToIgnoreCase() to compare " +
                           str1 + " with " + str3);
        result = str1.compareToIgnoreCase(str3); ←
        if(result < 0)
            System.out.println(str1 + " is less than " + str3);
        else if(result == 0)
            System.out.println(str1 + " is equal to " + str3);
        else if(result > 0)
            System.out.println(str1 + " is greater than " + str3);

    }
}
```

Estas duas comparações produzem uma ordem diferente.

A saída é mostrada aqui:

```
Using compareTo(): alpha and ALPHA differ
Using compareToIgnoreCase(): alpha and ALPHA are the same

Using compareTo() to compare alpha with Beta
alpha is greater than Beta

Using compareToIgnoreCase() to compare alpha with Beta
alpha is less than Beta
```

Vê algo incomum na saída? Observe que, quando “alpha” é comparada com “Beta” com o uso de **compareTo()**, alpha é maior do que Beta. No entanto, quando a comparação é feita com o uso de **compareToIgnoreCase()**, alpha é menor do que Beta. Veja por quê. Ainda que Beta venha após alpha na ordem alfabética, se você usar uma comparação de strings que diferencie maiúsculas de minúsculas, alpha será maior do que Beta porque Beta começa com uma letra maiúscula. Em ASCII/Unicode, letras maiúsculas têm um *valor menor* do que letras minúsculas. No entanto, quando a caixa das letras é ignorada, como ocorre na segunda operação, a ordem esperada é obtida. É por isso que, com frequência, **compareToIgnoreCase()** é uma alternativa útil na classificação de strings.

USANDO `indexOf()` E `lastIndexOf()`

A classe **String** fornece dois métodos que permitem a busca de um caractere ou substring específico em um string e a obtenção de um índice desse item se ele for encontrado. Eles são

- **indexOf()** Procura a primeira ocorrência de um caractere ou substring.
- **lastIndexOf()** Procura a última ocorrência de um caractere ou substring.

Esses dois métodos são sobre carregados de várias maneiras. Em todos os casos, retornam o índice em que o caractere ou substring foi encontrado, ou, caso contrário, -1.

Para procurar a primeira ocorrência de um caractere, use

```
int indexOf(int car)
```

Para procurar a última ocorrência de um caractere, use

```
int lastIndexOf(int car)
```

Aqui, *car* é o caractere que está sendo procurado.

Para procurar a primeira ou a última ocorrência de um substring, use

```
int indexOf(String str)
int lastIndexOf(String str)
```

Aqui, *str* especifica o substring.

Você pode especificar um ponto de partida para a busca usando estas formas:

```
int indexOf(int car, int índiceInicial)
int lastIndexOf(int car, int índiceInicial)
int indexOf(String str, int índiceInicial)
int lastIndexOf(String str, int índiceInicial)
```

Aqui, `índiceInicial` especifica o índice em cuja posição a busca começa. Em `indexOf()`, a busca percorre de `índiceInicial` até o fim do string. Em `lastIndexOf()`, a busca percorre de `índiceInicial` até zero.

O exemplo a seguir mostra como usar vários dos métodos de índices na pesquisa do conteúdo de `Strings`:

```
// Demonstra indexOf() e lastIndexOf().

class IndexOfDemo {
    public static void main(String[] args) {
        String str = "alpha beta gamma theta zeta";

        System.out.println("The string is: " + str);
        System.out.println("The first index of t is " + str.indexOf('t'));

        System.out.println("The last index of t is " + str.lastIndexOf('t'));

        System.out.println("The first index of eta is " + str.indexOf("eta"));

        System.out.println("The last index of eta is " + str.lastIndexOf("eta"));

        System.out.println("The first index of eta after position 10 is " +
                           str.indexOf("eta", 10));
    }
}
```

Esta é a saída do programa:

```
The string is: alpha beta gamma theta zeta
The first index of t is 8
The last index of t is 25
The first index of eta is 7
The last index of eta is 24
The first index of eta after position 10 is 19
```

Verificação do progresso

1. Para verificar se dois strings são iguais sem levar em consideração se as letras são maiúsculas ou minúsculas, use o método _____.
2. Para comparar dois strings e ver qual vem primeiro alfabeticamente, use o método _____.
3. A expressão `s.equals(t)` é igual a `s==t`. Verdadeiro ou falso?
4. Para determinar se um string `t` é substring de um string `s`, chame _____. O resultado será `-1` se e somente se `t` não for substring de `s`.

Respostas:

1. `equalsIgnoreCase()`
2. `compareTo()`
3. Falso.
4. `s.indexOf(t)`

OBTENDO UM STRING MODIFICADO

Como explicado, os objetos **String** são imutáveis. No entanto, é fácil obter uma versão modificada de um objeto **String** com o uso de um ou mais métodos definidos pela classe **String**. Esses métodos retornam uma cópia modificada do string original. Já que o string original nunca é modificado, a regra da imutabilidade não é violada. Nesta seção, examinaremos os métodos **substring()**, **replace()** e **trim()**; todos retornam versões modificadas do string em que são chamados.

substring()

Você pode obter parte de um string usando o método **substring()**. Ele tem duas formas. A primeira é

```
String substring(int índiceInicial)
```

Nela, *índiceInicial* especifica o índice em que o substring desejado começará. Essa forma retorna uma cópia do substring que começa em *índiceInicial* e inclui o resto do string chamador.

A segunda forma de **substring()** permite a especificação do índice tanto inicial quanto final do substring. Essa forma foi introduzida no Capítulo 5, mas será mostrada novamente aqui como complemento:

```
String substring(int índiceInicial, int índiceFinal)
```

Agora, *índiceInicial* é o índice inicial e *índiceFinal* especifica um índice uma unidade além do ponto final. O string retornado contém todos os caracteres de *índiceInicial* até, porém sem incluir, *índiceFinal*.

O programa a seguir demonstra as duas formas de **substring()**. No processo, mostra outro exemplo de **indexOf()** em ação. O programa define um string que contém as duas frases abaixo:

This is a test. This is, too.

Em seguida, ele usa **indexOf()** para encontrar o começo da segunda frase. Depois, obtém a segunda frase usando a forma de **substring()** de argumento único. O programa usa então uma combinação de **indexOf()**, **substring()** e concatenação de strings para substituir progressivamente um substring por outro dentro do string.

```
// Este programa demonstra as duas formas de substring().

class UseSubStrings {
    public static void main(String[] args) {
        String orgStr = "This is a test. This is, too.";
        String searchStr = "is";
        String subStr = "was";
        String resultStr = "";
        int i;

        System.out.println("Original string: " + orgStr);

        // Obtém a segunda frase em orgStr. Isso é
        // feito primeiro com a busca do fim da primeira
        // frase e então com a obtenção do resto do string.
        i = orgStr.indexOf(".") + 2; ← Encontra o fim da primeira frase.
```

```

String str = orgStr.substring(i);           ← Usa substring() para
System.out.println("Second sentence: " + str + "\n"); obter a segunda frase.

// Substitui todas as ocorrências de searchStr por subStr.
System.out.println ("Progressively replacing " +
                    searchStr + " with " + subStr);
do {
    System.out.println(orgStr);

    // encontra as próximas ocorrências de searchStr.
    i = orgStr.indexOf(searchStr);           ← Encontra o índice da
    if(i != -1) {                           próxima substituição.
        // obtém a primeira parte do string
        resultStr = orgStr.substring(0, i);   ← Obtém a primeira parte do string
                                                até o ponto de substituição.
        // adiciona a sequência de substituição
        resultStr = resultStr + subStr;      ← Adiciona o string substituto.

        // adiciona o resto do string, saltando searchStr
        resultStr = resultStr + orgStr.substring(i + searchStr.length());

        // cria o string resultante, o novo orgstr
        orgStr = resultStr;
    }
} while(i != -1);
}
}

A saída é mostrada aqui:

```

Agora, usa
substring() para obter
o resto do string.

```

Original string: This is a test. This is, too.
Second sentence: This is, too.

Progressively replacing is with was
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.

```

No programa, preste atenção em como a modificação progressiva de **orgStr** ocorre. A cada passagem pelo laço **do**, é feita uma tentativa de encontrar **searchStr**. Se ela for bem-sucedida, a primeira parte de **orgStr** será atribuída a **resultStr**. Em seguida, **subStr** é acrescentado a **resultStr**. Para concluir, o resto de **orgStr**, menos os caracteres que coincidem com os de **searchStr**, é acrescentado a **resultStr**. Por fim, **resultStr** é atribuído a **orgStr** e o processo se repete até não serem encontradas mais ocorrências.

replace()

O método **replace()** tem duas formas. A primeira substitui um caractere por outro dentro do string. Ela tem a forma geral a seguir:

String replace(char *original*, char *substituto*)

Aqui, *original* especifica o caractere a ser substituído pelo caractere especificado por *substituto*. A cópia modificada do string chamador é retornada.

A segunda forma de **replace()** substitui uma sequência de caracteres por outra dentro do string. Sua forma geral é:

String replace(CharSequence *original*, CharSequence *substituto*)

As sequências podem ser objetos de tipo **String**, porque **String** implementa a interface **CharSequence**. A cópia modificada do string chamador é retornada.

O programa a seguir demonstra as duas formas de **replace()**. Observe que ambas substituem todas as ocorrências do item original pelo substituto.

```
// Este programa demonstra as duas formas de replace().

class Replace {
    public static void main(String[] args) {
        String orgStr = "alpha beta gamma alpha beta gamma";
        String resultStr;

        System.out.println("Original string: " + orgStr);

        // Primeiro, substitui g por X.
        resultStr = orgStr.replace('g', 'X');
        System.out.println(resultStr);

        // Agora, substitui beta por zeta.
        resultStr = resultStr.replace("beta", "zeta");
        System.out.println(resultStr);
    }
}
```

A saída do programa é mostrada abaixo:

```
Original string: alpha beta gamma alpha beta gamma
alpha beta Xamma alpha beta Xamma
alpha zeta Xamma alpha zeta Xamma
```

trim()

O método **trim()** exclui o espaço em branco inicial e final (normalmente, espaços, tabulações e caracteres de nova linha) de um string. Ele tem esta forma geral:

String trim()

Se o string chamador tiver um espaço em branco inicial e/ou final, **trim()** o removerá e retornará um novo string contendo o resultado. Caso contrário, o string resultante será equivalente ao chamador.

Veja um exemplo. Dado o string:

```
|String str = " Gamma ";
```

Após

```
|str = str.trim();
```

str conterá somente o string “Gamma”. Os espaços em branco inicial e final foram excluídos.

O método **trim()** é muito útil quando processamos comandos do usuário. Por exemplo, o próximo programa solicita ao usuário o nome de um estado e então exibe sua capital. Ele usa **trim()** para remover qualquer espaço em branco inicial ou final que possa ter sido inserido inadvertidamente pelo usuário.

```
// Usando trim() para processar comandos.  
import java.io.*;  
  
class UseTrim {  
    public static void main(String[] args)  
        throws IOException  
{  
    // cria um BufferedReader usando System.in  
    BufferedReader br = new  
        BufferedReader(new InputStreamReader(System.in));  
    String str;  
  
    System.out.println("Enter 'stop' to quit.");  
    System.out.println("Enter State: ");  
    do {  
        str = br.readLine();  
        str = str.trim(); // remove o espaço em branco ← Usa trim() para remover  
                        // o espaço em branco  
        if(str.equals("Illinois"))  
            System.out.println("Capital is Springfield.");  
        else if(str.equals("Missouri"))  
            System.out.println("Capital is Jefferson City.");  
        else if(str.equals("California"))  
            System.out.println("Capital is Sacramento.");  
        else if(str.equals("Washington"))  
            System.out.println("Capital is Olympia.");  
        // ...  
    } while(!str.equals("stop"));  
}  
}
```

Quando esse programa for executado, se o usuário inserir algo como “ Illinois ”, a chamada a **trim()** removerá o espaço em branco inicial e final, permitindo que a entrada seja “ Illinois ”.

***Nota:** Se você estiver usando o JDK 7 ou posterior, poderá usar um **String** para controlar uma instrução **switch**. Se quiser, tente alterar o exemplo anterior para se beneficiar desse novo recurso.*

ALTERANDO A CAIXA DOS CARACTERES DE UM STRING

Às vezes é útil alterar a caixa das letras de uma string. Por exemplo, você pode querer normalizar o string para que seja usado por algum outro processo. O método **toLowerCase()** converte todos os caracteres de um string de maiúsculas para minúsculas. O método **toUpperCase()** converte todos os caracteres de um string de mi-

núsculas para maiúsculas. Caracteres não alfabéticos, como dígitos, não são afetados. Aqui estão as formas mais simples desses métodos:

```
String toLowerCase()
String toUpperCase()
```

Os dois métodos retornam um **String** contendo o equivalente em maiúsculas ou miúsculas do **String** chamador.

Veja um exemplo que usa **toLowerCase()** e **toUpperCase()**:

```
// Demonstra toUpperCase() e toLowerCase().

class ChangeCase {
    public static void main(String[] args)
    {
        String str = "This is a test.";

        System.out.println("Original: " + str);

        String upper = str.toUpperCase();
        String lower = str.toLowerCase();

        System.out.println("Uppercase: " + upper);
        System.out.println("Lowercase: " + lower);
    }
}
```

A saída produzida pelo programa é esta:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

String também fornece versões sobrecarregadas de **toLowerCase()** e **toUpperCase()** que permitem a especificação de um objeto **Locale** para controlar a conversão da caixa. (Consulte o Capítulo 24 para ver uma descrição de **Locale**.) A especificação da região pode ser muito útil em algumas situações e ajudar a internacionalizar seu aplicativo.

Verificação do progresso

1. O método **trim()** elimina _____.
2. Para que um string **s** passe a exibir apenas letras maiúsculas, use _____.
3. A chamada **s.substring(0, 3)** retorna um string de tamanho _____.

Respostas:

1. o espaço em branco no início e no fim do string
2. **s.toUpperCase()**
3. 3 (supondo que **s** tenha pelo menos três caracteres)

Pergunte ao especialista

P Além dos métodos que você discutiu, pode me falar dos outros métodos que **String** fornece?

R Devido à sua importância, **String** é uma classe poderosa e completa. Como tal, oferece ao programador um rico conjunto de métodos. Abaixo estão mais alguns que você deve achar interessantes e úteis.

Método	Descrição
<code>String concat(String str)</code>	Retorna o resultado do acréscimo de <i>str</i> ao fim do string chamador. Logo, desempenha a mesma função de <code>+</code> .
<code>boolean contains(CharSequence str)</code>	Retorna true se o objeto chamador tiver o string especificado por <i>str</i> . Caso contrário, retorna false .
<code>boolean contentEquals(CharSequence str)</code>	Retorna true se o string chamador contiver o mesmo string de <i>str</i> . Caso contrário, retorna false .
<code>static String format(String strfm, Object ... args)</code>	Retorna um string formatado como especificado por <i>strfm</i> . (Consulte o Capítulo 24 para ver detalhes sobre formatação.)
<code>boolean isEmpty()</code>	Retorna true se o string chamador não tiver caracteres. Caso contrário, retorna false .
<code>boolean matches(string expReg)</code>	Retorna true se o string chamador estiver relacionado à expressão regular especificada por <i>expReg</i> . Caso contrário, retorna false .
<code>String replaceAll(String expReg, String novaStr)</code>	Retorna um string em que todos os substrings relacionados à expressão regular especificada por <i>expReg</i> são substituídos por <i>novaStr</i> .
<code>String replaceFirst(String expReg, String novaStr)</code>	Retorna um string em que o primeiro substring que estiver relacionado à expressão regular especificada por <i>expReg</i> é substituído por <i>novaStr</i> .
<code>String[] split(String expReg)</code>	Decompe o string chamador em partes e retorna um array contendo o resultado. Cada parte é delimitada pela expressão regular passada em <i>expReg</i> .

Observe que vários dos métodos operam com expressões regulares. Uma introdução às expressões regulares pode ser encontrada no Apêndice B. Observe ainda que dois dos métodos recebem um **CharSequence** como argumento. Lembre-se, **String** implementa **CharSequence**. Logo, strings podem ser passados para esses métodos. Além disso, qualquer objeto que implemente **CharSequence** pode ser passado para esses métodos.

String fornece várias versões sobrecarregadas do método **static valueOf()**, que é usado para produzir a representação de um objeto, um array de caracteres ou um tipo primitivo na forma de string. **String** também dá suporte a métodos que tratam pontos de código Unicode.

StringBuffer E StringBuilder

Antes de sairmos do tópico da manipulação de strings, é importante mencionar duas alternativas a **String**. Elas são **StringBuffer** e **StringBuilder**. As duas classes oferecem recursos semelhantes aos de **String** com um diferencial importante: contêm strings que podem ser modificados. Por exemplo, você pode inserir caracteres e substrings no meio ou acrescentá-los ao fim. Tanto **StringBuffer** quanto **StringBuilder** crescem automaticamente para dar espaço a esses acréscimos. Você também pode excluir caracteres. Para dar suporte a esses tipos de operações, as duas classes têm métodos como **setCharAt()**, **append()**, **insert()** e **delete()**.

StringBuffer faz parte de Java desde a versão 1.0. **StringBuilder** é uma inclusão mais recente, adicionada pelo JDK 7. Ela é igual a **StringBuffer** exceto por uma diferença importante: não é sincronizada, ou seja, não fornece segurança entre as threads. A vantagem de **StringBuilder** é o desempenho mais rápido. No entanto, em casos em que um string mutável for acessado por várias threads, e não for empregada sincronização externa, você deve usar **StringBuffer** em vez de **StringBuilder**.

EXERCÍCIOS

1. Crie uma instrução que exiba o substring de um string **s** composto por todos os caracteres de **s** exceto o último.
2. Suponhamos que **s** fosse um string e tivéssemos a instrução a seguir:

```
| boolean b = s.isEmpty();
```

Encontre uma instrução ou um conjunto de instruções equivalentes usando **length()** em vez de **isEmpty()**.

3. O que aconteceria se você chamassem **charAt()** e o valor que passasse como índice estivesse fora do intervalo?
4. Suponhamos que você quisesse criar um string composto por duas linhas de texto, cada um em uma linha separada. Você não pode usar uma instrução como

```
| String s = "This is the first line.  
|           This is the second line";
```

porque ela gerará um erro do compilador. Como pode fazê-lo?

5. Qual é o valor da expressão **2+2+"ME"**? É "22ME" ou "4ME"?
6. Suponhamos que você quisesse verificar se uma variável de string **s** contém o string "abcdef". É suficiente chamar **s.startsWith("abc")** e **s.endsWith("def")** e então ver se as duas chamadas de método retornam verdadeiro?
7. No exemplo **UseTrim** deste capítulo, que demonstrou como usar o método **trim()**, a escada **if-else-if** verificou repetidamente se dois strings eram iguais usando o método **equals()**. Por que a escada não fez a verificação repetidamente usando **==**?
8. Explique a diferença entre **startsWith(substring, index)** e **indexOf(substring, index)**.

9. Suponhamos que você tivesse um string **s** contendo um texto em inglês que sempre usasse pronomes do gênero masculino (por exemplo, *he*, *his*, *him*) e quisesse alterá-la para usar pronomes femininos. O programa a seguir lê um string inserido pelo usuário e usa o método **replace()** para tentar fazer essa conversão. Ele converterá corretamente frases como “He went to his house” para “She went to her house”. Infelizmente, o programa não funciona corretamente com outras frases em inglês. Encontre uma frase que o programa não converta corretamente.

```
import java.io.*;  
  
public class ReplaceGender {  
    public static void main(String[] args)  
        throws IOException  
    {  
        // cria um BufferedReader usando System.in  
        BufferedReader br = new  
            BufferedReader(new InputStreamReader(System.in));  
        String str;  
  
        str = br.readLine();  
        str = str.replace(" he ", " she ");  
        str = str.replace("His ", "Her ");  
        str = str.replace("He ", "She ");  
        str = str.replace(" his ", " her ");  
        str = str.replace(" him ", " her ");  
  
        System.out.println(str);  
    }  
}
```

10. Crie um método que use um string como parâmetro e retorne **true** se todos os caracteres do string forem iguais.
11. Suponhamos que **s** fosse um string e tivéssemos a instrução a seguir:

```
| boolean b = s.isEmpty();
```

Encontre três instruções ou conjuntos de instruções diferentes equivalentes.
Não use o método **length()** de **String** como feito no Exercício 2.

12. Ainda que uma instância de **String** seja imutável, há várias maneiras de criar um novo string arbitrário a partir de um string existente. Uma é usar um **StringBuilder** ou um **StringBuffer**. Outra é converter o string em um array **char** usando **toCharArray()**, tratar os caracteres do array e então criar o novo string usando o construtor de **String** que usa um array **char** como argumento. Siga essa última abordagem para implementar um método **replaceChar()** que use três parâmetros: um string, um índice inteiro e um **char**. Ele retorna um novo string idêntico ao primeiro argumento, exceto pelo fato de o caractere do índice fornecido ser substituído pelo caractere que é o terceiro argumento do método. Por exemplo, **replaceChar("abc", 0, 'd')** retorna o string “dbc”.

13. Implemente um método **reverse()** que use um string como parâmetro e retorne um novo string em que a ordem dos caracteres esteja invertida. Use a mesma abordagem descrita no exercício anterior. Isto é, converta o string em um array **char**, trate o array e então converta-o novamente em um string.
14. Um idioma comum visto na programação Java é pegarmos uma variável de qualquer tipo e adicionarmos o string vazio a ela, como em **x + ""**. O que essa inclusão faz?
15. Se **s** e **t** são strings e você quiser saber se **s** começa com o string **t**, pode executar **s.startsWith(t)**. Como mencionado no texto, você também pode usar **regionMatches()** em vez de **startsWith()**. Forneça uma expressão usando **regionMatches()** que seja equivalente a **s.startsWith(t)**.
16. Suponhamos que **s** fosse um string e tivéssemos a instrução a seguir:

```
| char[] cArray = s.toCharArray();
```

Encontre uma instrução ou um conjunto de instruções equivalente usando **getChars()** em vez de **toCharArray()**.

17. Implemente um método **isSubstring()** que use dois strings **s** e **t** como parâmetros e retorne **true** se **t** for substring de **s**. Use a segunda forma de **startsWith()** (que tem dois parâmetros). Não use os métodos **indexOf()** sobrecarregados.
18. Suponhamos que **s** e **t** fossem strings e tivéssemos a instrução a seguir:

```
| boolean b = s.equalsIgnoreCase(t);
```

Crie uma instrução ou um conjunto de instruções equivalente usando **equals()** e **toLowerCase()** em vez de **equalsIgnoreCase()**.

19. Que string vem primeiro na ordem alfabética: "\$*%" ou "&@"? Por quê?
20. Implemente um método **myTrim()** que use um string como parâmetro e retorne um novo string com todos os caracteres de espaço em branco removidos do começo e do fim. Não use o método **trim()** da classe **String**. Em vez disso, chame repetidamente **substring()** para obter strings cada vez mais curtos. Assuma como caracteres de espaço em branco os caracteres de espaço " ", tabulação '\t', nova linha '\n' e retorno '\r'.
21. Implemente um método **isEmailAddress()** que use um string **s** como parâmetro. Ele retorna **true** quando **s** tem a forma "**x@y.com**", em que **x** e **y** podem ser qualquer combinação de uma ou mais letras (minúsculas ou maiúsculas) ou dígitos.
22. Suponhamos que **s** e **t** fossem strings e tivéssemos a instrução a seguir:

```
| String v = s.concat(t);
```

Crie uma instrução ou um conjunto de instruções equivalente sem usar o método **concat()** e sem usar o operador + nos strings.

23

Examinando o pacote `java.lang`

PRINCIPAIS HABILIDADES E CONCEITOS

- Os encapsuladores de tipos primitivos
- A classe **Math**
- A classe **Process**
- A classe **ProcessBuilder**
- A classe **Runtime**
- A classe **System**
- A classe **Object**
- A classe **Class**
- A classe **Enum**
- A interface **Comparable**
- A interface **Appendable**
- A interface **Iterable**
- A interface **Readable**
- A interface **CharSequence**
- A interface **AutoCloseable**

Este capítulo examinará o pacote **java.lang**. Como você sabe, **java.lang** é importado automaticamente para todos os programas. Ele contém classes e interfaces que são fundamentais para praticamente toda a programação Java. É o pacote Java mais amplamente usado e todos os programadores de Java devem ter um conhecimento geral do que ele fornece.

O pacote **java.lang** inclui as seguintes classes de nível superior:

Boolean	Byte	Character
Class	ClassLoader	ClassValue
Compiler	Double	Enum
Float	InheritableThreadLocal	Integer
Long	Math	Number
Object	Package	Process
ProcessBuilder	Runtime	RuntimePermission

<code>SecurityManager</code>	<code>Short</code>	<code>StackTraceElement</code>
<code>StrictMath</code>	<code>String</code>	<code>StringBuffer</code>
<code>StringBuilder</code>	<code>System</code>	<code>Thread</code>
<code>ThreadGroup</code>	<code>ThreadLocal</code>	<code>Throwable</code>
<code>Void</code>		

As interfaces de nível superior definidas por `java.lang` são as mostradas abaixo:

<code>Appendable</code>	<code>Cloneable</code>	<code>Readable</code>
<code>AutoCloseable</code>	<code>Comparable</code>	<code>Runnable</code>
<code>CharSequence</code>	<code>Iterable</code>	

Como podemos ver, as classes e interfaces de `java.lang` definem um grande número de funcionalidades. Algumas partes de `java.lang`, como `String`, `StringBuilder`, `StringBuffer`, `Throwable`, `Thread` e `Runnable`, foram descritas em outros locais do livro. Além disso, nem todas as partes são amplamente usadas, ou usadas por todos os programadores. Por exemplo, algumas classes, como `StrictMath` e `Compiler`, são mais aplicáveis a situações específicas. Portanto, enfocaremos classes aplicáveis a um amplo grupo de situações e não descritas em outro local do livro. O capítulo termina com uma visão geral das interfaces de `java.lang`. Começaremos revendo os encapsuladores de tipos primitivos.

ENCAPSULADORES DE TIPOS PRIMITIVOS

Como explicado na Parte I, Java usa tipos primitivos, como `int` e `char`, por razões de desempenho. Esses tipos de dados não fazem parte da hierarquia de objetos. Eles são passados por valor para os métodos e não podem ser passados diretamente por referência. Além disso, não há como dois métodos referenciarem a *mesma instância* de um tipo primitivo. Haverá momentos em que você terá que criar uma representação para um desses tipos primitivos na forma de objeto. Por exemplo, há classes de coleções discutidas no Capítulo 25 que só lidam com objetos; para armazenar um tipo primitivo em uma delas, você terá que encapsulá-lo em uma classe. Para atender essa necessidade, Java fornece classes que correspondem a cada um dos tipos primitivos. Basicamente, essas classes encapsulam, ou *inserem*, os tipos primitivos dentro de uma classe. Logo, normalmente elas são chamadas de *encapsuladores de tipos*. Embora vários aspectos dos encapsuladores de tipos tenham sido abordados na Parte I, aqui examinaremos algumas de suas outras características e recursos. Começaremos revendo a superclasse dos encapsuladores de tipos numéricos: `Number`.

Number

Como vimos na Parte I, a classe abstrata `Number` define uma superclasse que é estendida (entre outras) pelas classes que encapsulam os tipos numéricos `byte`, `short`, `int`, `long`, `float` e `double`. `Number` declara métodos que retornam o valor do objeto em cada um dos diferentes formatos de número. Por exemplo, `doubleValue()` retorna o valor como um `double`, `floatValue()` como um `float`, e assim por diante. A necessidade de usar esses métodos foi bastante reduzida devido ao *autounboxing*. No entanto, eles ainda estão disponíveis se quisermos usá-los. `Number` implementa a interface `java.io.Serializable`.

Em **java.lang.Number** tem subclasses concretas que armazenam valores explícitos de cada tipo numérico: **Double**, **Float**, **Byte**, **Short**, **Integer** e **Long**.

Double e Float

Double e **Float** são encapsuladores de valores de ponto flutuante de tipo **double** e **float**, respectivamente. Ambas estendem **Number** e implementam a interface **Comparable<T>**. Os construtores de **Float** são mostrados abaixo:

```
Float(double num)
Float(float num)
Float(String str) throws NumberFormatException
```

Como você pode ver, objetos **Float** podem ser construídos com valores de tipo **float** ou **double**. Também podem ser construídos a partir da representação de um número de ponto flutuante na forma de string.

Os construtores de **Double** são os seguintes:

```
Double(double num)
Double(String str) throws NumberFormatException
```

Objetos **Double** podem ser construídos com um valor **double** ou um string contendo um valor de ponto flutuante.

O exemplo a seguir cria dois objetos **Double** – um usando um valor **double** e o outro passando um string que pode ser analisado como um **double**:

```
class DoubleDemo {
    public static void main(String[] args) {
        Double d1 = new Double(3.14159);
        Double d2 = new Double("314159E-5");

        System.out.println(d1 + " " + d2);
    }
}
```

Como você pode ver na saída abaixo, os dois construtores criaram instâncias idênticas de **Double**.

```
| 3.14159 3.14159
```

Tanto **Double** quanto **Float** definem várias constantes que são úteis no trabalho com números. Entre elas, temos:

MAX_VALUE	Valor positivo máximo
MIN_VALUE	Valor positivo mínimo
NaN	Um valor que representa algo que não é um número
POSITIVE_INFINITY	Infinito positivo
NEGATIVE_INFINITY	Infinito negativo

Para **Float**, esses valores são de tipo **float**. Para **Double**, são de tipo **double**.

Além dos métodos herdados de **Number**, tanto **Float** quanto **Double** definem vários outros métodos. Por exemplo, você pode verificar se dois números são iguais usando **equals()**, e seu relacionamento dentro de uma determinada ordem usando **compareTo()**, que é definido por **Comparable<T>**. Como explicado no Capítulo 11, **Float**

define o método **parseFloat()** e **Double** define o método **parseDouble()**. Esses métodos podem ser usados na conversão de strings numéricos em valores **float** ou **double**.

Dois métodos muito interessantes são **isInfinite()** e **isNaN()**. Eles são definidos por **Float** e **Double**. As versões **Double** são mostradas abaixo e as versões **Float** são semelhantes.

```
boolean isInfinite()
boolean isNaN()
```

O método **isInfinite()** retorna **true** quando o objeto chamador contém um valor infinito, que pode ser positivo ou negativo. O método **isNaN()** retorna **true** quando o objeto chamador contém um valor que não é um número.

O próximo exemplo cria dois objetos **Double**; um é infinito e o outro não é um número.

```
// Demonstra isInfinite() e isNaN()
class InfNaN {
    public static void main(String[] args) {
        Double d1 = new Double(1/0.0);
        Double d2 = new Double(0/0.0);

        System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1isNaN());
        System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isnan());
    }
}
```

Esse programa gera a saída a seguir:

```
| Infinity: true, false
| NaN: false, true
```

Nota: Pode interessar o fato de que, se a divisão do programa tivesse usado um inteiro como divisor, teria sido lançada uma exceção de divisão por zero em vez de ter sido gerado um valor infinito ou NaN.

Tanto **Float** quanto **Double** também fornecem versões **static** de **isInfinite()** e **isNaN()**. As versões **Double** são estas:

```
static boolean isInfinite(double num)
static boolean isNaN(double num)
```

Nos dois casos, o número é passado em *num* e o resultado é retornado.

Byte, Short, Integer e Long

As classes **Byte**, **Short**, **Integer** e **Long** são encapsuladores dos tipos inteiros **byte**, **short**, **int** e **long**, respectivamente. Todas estendem **Number** e implementam a interface **Comparable<T>**. Seus construtores são mostrados abaixo:

```
Byte(byte num)
Byte(String str) throws NumberFormatException
```

```
Short(short num)
Short(String str) throws NumberFormatException
```

`Integer(int num)`

`Integer(String str) throws NumberFormatException`

`Long(long num)`

`Long(String str) throws NumberFormatException`

Como se vê, esses objetos podem ser construídos a partir de valores numéricos ou de strings contendo números inteiros válidos.

Entre outros campos, todos os encapsuladores de números inteiros definem as constantes a seguir:

MIN_VALUE	Valor mínimo
MAX_VALUE	Valor máximo

O tipo dos valores será o tipo encapsulado.

Além dos métodos herdados de **Number**, todos os encapsuladores de inteiros definem vários outros métodos. Por exemplo, você pode verificar se dois números são iguais usando **equals()** e seu relacionamento dentro de uma determinada ordem usando **compareTo()**, que é definido por **Comparable<T>**.

Todos os encapsuladores de números inteiros definem métodos para a análise de inteiros a partir de strings e a conversão dos strings novamente em inteiros. Como você sabe, as classes **Byte**, **Short**, **Integer** e **Long** fornecem os métodos **parseByte()**, **parseShort()**, **parseInt()** e **parseLong()**, respectivamente. Esses métodos retornam o **byte**, **short**, **int** ou **long** equivalente ao string numérico com o qual são chamados. No entanto, variantes dos métodos permitem a especificação da *raiz*, ou base numérica, da conversão. As bases comuns são 2 para binário, 8 para octal, 10 para decimal e 16 para hexadecimal. Estas são as versões dos métodos de análise que usam a base numérica:

```
static byte parseByte(String str, int base) throws NumberFormatException
static short parseShort(String str, int base) throws NumberFormatException
static int parseInt(String str, int base) throws NumberFormatException
static long parseLong(String str, int base) throws NumberFormatException
```

Em todos os casos, *str* especifica o string numérico a ser convertido, e *base* especifica a base numérica. O resultado da conversão é retornado. Uma **NumberFormatException** é lançada quando *str* não contém um número válido. Veja um exemplo que analisa um valor binário:

```
// Demonstra parseInt().

class ParseBinary {
    public static void main(String[] args) {
        int num;
        String str = "10011101";

        num = Integer.parseInt(str, 2);

        System.out.println("Here is " + str + " in decimal: " + num);
    }
}
```

A saída é mostrada abaixo:

```
|Here is 10011101 in decimal: 157
```

Para converter um número em um string decimal, use as versões de `toString()` definidas nas classes **Byte**, **Short**, **Integer** ou **Long**. As classes **Integer** e **Long** também fornecem os métodos `toBinaryString()`, `toHexString()` e `toOctalString()`, que convertem um valor em um string binário, hexadecimal ou octal, respectivamente. Estas são as versões de **Integer**:

```
static String toBinaryString(int num)
static String toHexString(int num)
static String toOctalString(int num)
```

Em todos os casos, *num* especifica o valor a ser convertido para sua representação na forma de string.

Nota: Tanto **Float** quanto **Double** também fornecem o método `toHexString()`, que converte um valor de ponto flutuante para seu formato de ponto flutuante hexadecimal.

O programa a seguir demonstra a conversão binária, hexadecimal e octal:

```
// Converte um inteiro em binário, hexadecimal
// e octal.

class StringConversions {
    public static void main(String[] args) {
        int num = 19648;

        System.out.println(num + " in binary: " +
                           Integer.toBinaryString(num));

        System.out.println(num + " in octal: " +
                           Integer.toOctalString(num));

        System.out.println(num + " in hexadecimal: " +
                           Integer.toHexString(num));
    }
}
```

A saída do programa é mostrada abaixo:

```
19648 in binary: 100110011000000
19648 in octal: 46300
19648 in hexadecimal: 4cc0
```

Tanto **Integer** quanto **Long** contêm dois métodos que fazem a rotação dos bits de um valor inteiro. Eles se chamam `rotateLeft()` e `rotateRight()`. Suas formas **Integer** são estas:

```
static int rotateLeft( )(int num, int n)
static int rotateRight( )(int num, int n)
```

Aqui, *num* é o número que sofre a rotação e *n* especifica quantas posições serão percorridas. A rotação é semelhante a um deslocamento, exceto pelo bit que sai de uma extremidade voltar para a outra extremidade. Veja um exemplo:

```
// Demonstra rotateLeft() e rotateRight()

class Rotations {
    public static void main(String[] args) {
        int num = -3356756;

        System.out.println(Integer.toBinaryString(num));

        num = Integer.rotateLeft(num, 2);
        System.out.println(Integer.toBinaryString(num));

        num = Integer.rotateRight(num, 2);
        System.out.println(Integer.toBinaryString(num));
    }
}
```

A saída é esta:

```
111111110011001100011110101100
11111111001100110001111010110011
1111111110011001100011110101100
```

Character

Character é um encapsulador simples que contém um **char**. Ele implementa **java.io.Serializable** e **Comparable<T>**. Só tem um construtor:

```
Character(char car)
```

Aqui, *car* especifica o caractere que será encapsulado pelo objeto **Character** que está sendo criado. Para obter o valor **char** contido em um objeto **Character**, chame o método **charValue()**, mostrado abaixo:

```
char charValue()
```

Ele retorna o caractere.

Character tem vários métodos **static** que categorizam caracteres. Outros alteram a caixa dos caracteres. Um resumo pode ser visto na Tabela 23-1. Esses métodos podem ser muito úteis no trabalho com caracteres. Por exemplo, se um arquivo separar campos com um espaço, você pode usar **isSpaceChar()** para encontrar o fim de um campo e o início do outro. Se estiver analisando uma linha de texto, pode usar **isDigit()** para encontrar o início de um número ou **isLetterOrDigit()** para ler a identificação de um funcionário, como RH534R, que contém tanto letras quanto dígitos. O exemplo a seguir demonstra vários desses métodos:

```
// Demonstra vários métodos is...

class IsDemo {
    public static void main(String[] args) {
```

```

char[] a = {'a', 'b', '5', '?', 'A', ' ', '\t', '\n'};

for(int i=0; i<a.length; i++) {
    if(Character.isDigit(a[i]))
        System.out.println(a[i] + " is a digit.");

    if(Character.isLetter(a[i]))
        System.out.println(a[i] + " is a letter.");

    if(Character.isLetterOrDigit(a[i]))
        System.out.println(a[i] + " is a letter or digit.");

    if(Character.isWhitespace(a[i])) { ←
        String chStr = "";
        if(a[i] == ' ') chStr = "<space>";
        else if(a[i] == '\t') chStr = "<tab>";
        else if(a[i] == '\n') chStr = "<newline>";
        System.out.println(chStr + " is whitespace.");
    }

    if(Character.isSpaceChar(a[i])) { ←
        String chStr = "";
        if(a[i] == ' ') chStr = "<space>";
        else if(a[i] == '\t') chStr = "<tab>";
        else if(a[i] == '\n') chStr = "<newline>";
        System.out.println(chStr + " is space.");
    }

    if(Character.isUpperCase(a[i]))
        System.out.println(a[i] + " is uppercase.");

    if(Character.isLowerCase(a[i]))
        System.out.println(a[i] + " is lowercase.");

    System.out.println();
}
}

```

Na saída, observe que estes métodos exibem conjuntos de caracteres diferentes.

A saída do programa é:

```

a is a letter.
a is a letter or digit.
a is lowercase.

b is a letter.
b is a letter or digit.
b is lowercase.

5 is a digit.
5 is a letter or digit.

```

```

A is a letter.
A is a letter or digit.
A is uppercase.

<space> is whitespace.
<space> is space.

<tab> is whitespace.

<newline> is whitespace.

```

Na saída, observe que há uma diferença entre `isSpaceChar()` e `isWhiteSpace()`. O método `isSpaceChar()` encontra espaços, que também podem incluir separadores de linhas e de parágrafos. O método `isWhiteSpace()` encontra todos os espaços em branco, como tabulações e novas linhas.

Outros dois métodos que você pode achar interessantes são `forDigit()` e `digit()`. Eles permitem a conversão entre os valores inteiros e os dígitos que representam. Podemos vê-los abaixo:

```

static char forDigit(int num, int base)
static int digit(char dígito, int base)

```

Tabela 23-1 Resumo dos métodos de Character

Método	Descrição
<code>static boolean isDigit(char car)</code>	Retorna true quando <i>car</i> é um dígito. Retorna false para todos os outros caracteres.
<code>static boolean isJavaIdentifierPart(char car)</code>	Retorna true quando <i>car</i> é permitido em um identificador Java, só não pode ser o primeiro. Retorna false para todos os outros caracteres.
<code>static boolean isJavaIdentifierStart(char car)</code>	Retorna true quando <i>car</i> é permitido como o primeiro caractere de um identificador Java. Retorna false para todos os outros caracteres.
<code>static boolean isLetter(char car)</code>	Retorna true quando <i>car</i> é uma letra. Retorna false para todos os outros caracteres.
<code>static boolean isLetterOrDigit(char car)</code>	Retorna true quando <i>car</i> é uma letra ou dígito. Retorna false para todos os outros caracteres.
<code>static boolean isLowerCase(char car)</code>	Retorna true quando <i>car</i> é uma letra minúscula. Retorna false para todos os outros caracteres.
<code>static boolean isSpaceChar(char car)</code>	Retorna true quando <i>car</i> é um caractere de espaço, inclusive separadores de linha e de parágrafo. Retorna false para todos os outros caracteres.
<code>static boolean isUpperCase(char car)</code>	Retorna true quando <i>car</i> é uma letra maiúscula. Retorna false para todos os outros caracteres.
<code>static boolean isWhiteSpace(char car)</code>	Retorna true quando <i>car</i> é um espaço em branco, como os espaços, tabulações e novas linhas. Retorna false para todos os outros caracteres.
<code>static char toLowerCase(char car)</code>	Retorna a representação minúscula de <i>car</i> .
<code>static char toUpperCase(char car)</code>	Retorna a representação maiúscula de <i>car</i> .

O método **forDigit()** retorna o caractere numérico associado ao valor de *num*. A base numérica da conversão é especificada por *base*. O método **digit()** retorna o valor inteiro associado a *dígito* (que, presume-se, seja um dígito) de acordo com a base numérica especificada por *base*.

Character define várias constantes e tem outros métodos além dos discutidos aqui. É uma classe que você deve examinar melhor ao avançar em seu estudo de Java.

Antes de sairmos do tópico **Character**, é necessário mencionar uma alteração que ocorreu alguns anos atrás. A partir do JDK 5, a classe **Character** passou a dar suporte a caracteres Unicode de 32 bits. No passado, todos os caracteres Unicode cabiam em 16 bits, que é o tamanho de um **char** (o valor encapsulado por **Character**). No entanto, posteriormente o conjunto de caracteres Unicode foi expandido, demandando mais de 16 bits. Os caracteres adicionais são chamados de *caracteres complementares*. Acréscimos foram feitos a **Character** para permitir o suporte ao conjunto Unicode expandido, incluindo vários métodos sobrecarregados que usam um parâmetro **int** em vez de **char**. Além disso, em um array ou string, quando um caractere cujo valor está fora do intervalo de **char** é necessário, ele é dividido em duas partes, chamadas de *substituto baixo* e *substituto alto*. Java usa o UTF-16 (Unicode Transformation Format) para codificar esses caracteres. Nesse ponto de seu estudo de Java, você não deve precisar de caracteres complementares. Mas eles podem ser importantes em aplicativos que desenvolver em sua carreira profissional.

Boolean

Boolean é um encapsulador de valores **boolean** sem muitos recursos, que pode ser útil na passagem de uma variável **boolean** por referência. Essa classe implementa **java.io.Serializable** e **Comparable<T>** e define os construtores abaixo:

```
Boolean(boolean valorBool)
Boolean(String stringBool)
```

Na primeira versão, *valorBool* dever ser **true** ou **false**. Na segunda, se *stringBool* contiver o string “true” (em maiúsculas ou minúsculas), o novo objeto **Boolean** será **true**. Caso contrário, será **false**.

Duas das constantes especificadas por **Boolean** são **TRUE** e **FALSE**, que definem objetos **Boolean** verdadeiros e falsos.

Boolean fornece alguns métodos próprios. Um particularmente interessante é **parseBoolean()**, que converte um string em seu equivalente **boolean**. Ele é mostrado aqui:

```
static boolean parseBoolean(String str)
```

Independentemente de diferenças na caixa das letras, se *str* contiver o string “true”, **true** será retornado. Qualquer outro valor em *str* fará **parseBoolean()** retornar **false**.

O método **toString()** de **Boolean** retorna “true” quando o valor chamador é **true** e “false” quando ele é **false**.

O autoboxing e os encapsuladores de tipos

Como explicado na Parte I, devido ao *autoboxing*, você não precisa construir uma instância encapsuladora de tipos chamando explicitamente um construtor. Por exemplo, a variável **hypot** pode ser construída com o uso desta linha:

```
|Double hypot = 170.54;
```

em vez de

```
| Double hypot = new Double(170.54);
```

O *autoboxing* pode ser usado com qualquer um dos encapsuladores de tipos.

Verificação do progresso

1. As classes encapsuladoras **Byte**, **Short**, **Integer**, **Long**, **Float** e **Double** são todas subclasses de _____.
2. Devido ao recurso chamado _____, raramente precisamos construir explicitamente um encapsulador de valores primitivos.
3. As classes encapsuladoras de tipos primitivos têm duas grandes vantagens. A primeira é que permitem a criação de um valor primitivo na forma de objeto. Qual é a segunda?

A CLASSE Math

Para alguns tipos de aplicativos, uma das classes mais importantes de **java.lang** é **Math**. A classe **Math** contém um grande número de métodos **static** que executam funções matemáticas comuns, como calcular o seno de um ângulo ou obter um logaritmo. Portanto, você se pretende fazer cálculos em alta velocidade, provavelmente usará métodos definidos por **Math**. Embora **Math** defina métodos demais para examinarmos cada um, faremos um resumo aqui para lhe dar uma ideia do que a classe tem a oferecer.

Nota: Se a análise numérica estiver em seus planos, examine a classe **Math** com atenção.

Math define vários métodos trigonométricos. Estes são os de seno, cosseno e tangente:

```
static double sin(double arg)
static double cos(double arg)
static double tan(double arg)
```

Em todos eles, *arg* especifica o ângulo em radianos.

As funções de seno do arco, cosseno do arco e tangente do arco são suportadas por estes métodos:

```
static double asin(double arg)
static double acos(double arg)
static double atan(double arg)
```

Eles retornam o resultado indicado.

Respostas:

1. **Number**
2. *autoboxing*
3. Fornecem um conjunto útil de métodos e constantes para o trabalho com os valores primitivos.

Você pode obter o logaritmo natural de um valor usando **log()**. Para obter o logaritmo na base 10, use **log10()**. Esses métodos são mostrados abaixo:

```
static double log(double arg)
static double log10(double arg)
```

Aqui, *arg* especifica o valor cujo logaritmo está sendo obtido.

Para calcular o resultado da elevação de um valor a uma potência especificada, use **pow()**:

```
static double pow(double v, double x)
```

Ele retorna *v* elevado a *x*; por exemplo, **pow(2.0, 3.0)** retorna 8,0.

Como você já deve ter visto anteriormente neste livro, a raiz quadrada de um valor pode ser obtida com uma chamada a **sqrt()**. O que pode ser novidade é que **Math** também fornece o método **cbrt()**, que retorna a raiz cúbica. Veja os dois métodos:

```
static double sqrt(double arg)
static double cbrt(double arg)
```

O valor para o qual a raiz quadrada ou cúbica está sendo obtida é passado em *arg*. A raiz é retornada.

Às vezes é útil obter o *piso* ou o *teto* de um valor. O piso é o maior número inteiro menor ou igual ao valor. O teto é o menor número inteiro maior ou igual ao valor. Para calcular essa funções, **Math** tem os métodos **floor()** e **ceil()**, mostrados abaixo:

```
static double floor(double arg)
static double ceil(double arg)
```

O piso ou o teto de *arg* é retornado.

Math define várias versões sobrecarregadas de métodos de valor absoluto. Aqui estão elas:

```
static int abs(int arg)
static long abs(long arg)
static float abs(float arg)
static double abs(double arg)
```

Em todos os casos, o valor absoluto de *arg* é retornado.

Costuma ser útil a conversão de radianos em graus e vice-versa. Para executar essas tarefas, **Math** fornece os métodos **toDegree()** e **toRadians()**, mostrados a seguir:

```
static double toDegrees(double ângulo)
static double toRadians(double ângulo)
```

Em **toDegrees()**, o ângulo passado para *ângulo* deve ser especificado em radianos. O resultado em graus é retornado. Em **toRadians()**, o ângulo passado para *ângulo* deve ser especificado em graus. O resultado em radianos é retornado.

Outro método interessante é **hypot()**:

```
static double hypot(double lado1, double lado2)
```

Dado o comprimento dos dois lados opostos, **hypot()** retorna o comprimento da hipotenusa de um triângulo retângulo. Em outras palavras, ele retorna a raiz quadrada da soma dos quadrados de *lado1* e *lado2*.

Uma maneira de obter um número pseudoaleatório é usando o método **random()** de **Math** mostrado aqui:

```
static double random()
```

Ele retorna um número aleatório entre 0 e 1.

Além dos métodos definidos por **Math**, essa classe também declara duas constantes **static final**. A primeira é **E**, que é aproximadamente 2,72. A segunda é **PI**, aproximadamente 3,14. Você pode usar essas constantes em vez de inserir os valores manualmente.

O programa a seguir demonstra várias funções de **Math**.

```
// Demonstra várias funções de Math.

class MathDemo {
    public static void main(String[] args) {

        // faz a conversão entre radianos e graus
        double theta = 120.0;

        System.out.println(theta + " degrees is " +
                           Math.toRadians(theta) + " radians.");

        theta = 1.312;
        System.out.println(theta + " radians is " +
                           Math.toDegrees(theta) + " degrees\n");

        // demonstra sin() e asin()
        theta = 1.0;
        double sine = Math.sin(theta);
        double asine = Math.asin(sine);
        System.out.println("Sine of " + theta + " is " + sine);
        System.out.println("Arc sine of " + sine + " is " + asine + "\n");

        // encontra a hipotenusa de um triângulo retângulo
        double h = Math.hypot(3.0, 4.0);
        System.out.println("Hypotenuse is " + h + "\n");

        // calcula uma potência
        double p = Math.pow(3.0, 3.0);
        System.out.println("pow(3.0, 3.0) is " + p + "\n");

        // usa log10()
        double lg = Math.log10(100.0);
        System.out.println("log10(100.0) is " + lg + "\n");

        // exibe E e PI
        System.out.println("PI: " + Math.PI + "\n E: " + Math.E);
    }
}
```

A saída é mostrada aqui:

```
120.0 degrees is 2.0943951023931953 radians.
1.312 radians is 75.17206272116401 degrees

Sine of 1.0 is 0.8414709848078965
Arc sine of 0.8414709848078965 is 1.0

Hypotenuse is 5.0

pow(3.0, 3.0) is 27.0

log10(100.0) is 2.0

PI: 3.141592653589793
E: 2.718281828459045
```

Pergunte ao especialista

P Cite alguns dos outros métodos da classe **Math**.

R Um método interessante é **copySign()**, que tem a forma a seguir:

```
static double copySign(double valor, double valorSinal)
```

Esse método altera o sinal do primeiro argumento para que fique igual ao do segundo e retorna o resultado. Por exemplo, **copySign(3.14, -4)** retorna -3,14 e **copySign(-3.14, 5.3)** retorna 3,14.

Outro método interessante é **exp()**, que tem a forma:

```
static double exp(double arg)
```

Ele calcula e^{arg} , em que e é a constante matemática representada em Java pelo valor **Math.E**. Em outras palavras, **exp(arg)** é apenas um método de conveniência que faz o mesmo que **pow(Math.E, arg)**.

Verificação do progresso

- Para elevar um número a uma potência, usamos o método _____ de **Math**.
- Para converter radianos em graus, usamos o método _____ de **Math**.

Respostas:

1. **pow()**
2. **toDegrees()**

A CLASSE Process

A classe abstrata **Process** encapsula um *processo* – isto é, um programa em execução. É usada principalmente como superclasse para o tipo de objetos criados por **start()** na classe **ProcessBuilder** ou por **exec()** na classe **Runtime**. **Process** contém os métodos abstratos mostrados abaixo:

Método	Descrição
<code>void destroy()</code>	Encerra o processo.
<code>int exitValue()</code>	Retorna um código de saída obtido no processo.
<code>InputStream getErrorStream()</code>	Retorna um fluxo de entrada que lê entradas no fluxo de saída err do processo.
<code>InputStream getInputStream()</code>	Retorna um fluxo de entrada que lê entradas no fluxo de saída out do processo.
<code>OutputStream getOutputStream()</code>	Retorna um fluxo de saída que grava saídas no fluxo de entrada in do processo.
<code>int waitFor() throws InterruptedException</code>	Retorna o código de saída obtido no processo. Esse método não retorna até o processo em que é chamado terminar.

Preste atenção em **exitValue()**, **destroy()** e **waitFor()**. Usaremos esses métodos nos exemplos da próxima seção.

A CLASSE ProcessBuilder

Dependendo das restrições de segurança, Java nos permite iniciar a execução de outro processo pesado (isto é, outro programa) com o uso da classe **ProcessBuilder**. Acabamos de explicar que todos os processos são representados pela classe **Process**. **ProcessBuilder** cria um novo processo e dá acesso a ele por intermédio de uma instância de **Process**. Como veremos, criar um novo processo é uma tarefa fácil. Os exemplos a seguir demonstram o procedimento. No entanto, lembre-se de que applets não assinados (não confiáveis) não podem iniciar um novo processo devido a restrições de segurança.

ProcessBuilder define dois construtores. O que usaremos é:

```
ProcessBuilder(String ... args)
```

Aqui, *args* é uma lista de argumentos que especifica o nome do programa a ser executado, além de outros argumentos de linha de comando que forem necessários. Observe que *args* é um parâmetro varargs. Ou seja, você pode passar para ele uma lista de argumentos, ou um array de argumentos.

A criação de uma instância de **ProcessBuilder** não inicia a execução do processo. Em vez disso, o processo deve ser iniciado com o uso do método **start()**, mostrado abaixo:

```
Process start() throws IOException
```

O método **start()** inicia o processo especificado pelo objeto chamador. Em outras palavras, executa o programa especificado. Ele retorna um objeto **Process** que en-

capsula o processo. Logo, o novo processo pode ser gerenciado por intermédio da instância de **Process**.

O exemplo a seguir executa o editor de texto **Bloco de Notas** do Windows. Observe que ele especifica o nome do arquivo a ser editado como argumento. (Se não estiver usando o Windows, faça a substituição por um programa apropriado fornecido por seu sistema operacional.)

```
// Demonstra ProcessBuilder.

import java.io.IOException;

class PBDemo {
    public static void main(String[] args) {

        try {
            ProcessBuilder procBldr =
                new ProcessBuilder("notepad", "testfile");
            procBldr.start(); ← Inicia o processo.
        } catch (IOException exc) {
            System.out.println("Error executing notepad.\n" + exc);
        }
    }
}
```

O objeto **Process** retornado por **ProcessBuilder** pode ser gerenciado por métodos de **Process** após o novo programa começar a execução. Por exemplo, o método **waitFor()** faz o programa esperar até o processo filho terminar, e então, retorna o valor retornado pelo processo filho. Normalmente esse valor é 0 quando não ocorre nenhum problema. Agora vejamos o exemplo anterior modificado para esperar o processo que está sendo executado terminar:

```
// Espera até o processo ser encerrado.

import java.io.IOException;

class PBDemo2 {
    public static void main(String[] args) {
        try {
            ProcessBuilder procBldr =
                new ProcessBuilder("notepad", "testfile");

            Process p = procBldr.start();
            p.waitFor(); ← Espera o processo terminar.
            System.out.println("Notepad returned " + p.exitValue());
        } catch (IOException exc) {
            System.out.println("Error executing notepad.\n" + exc);
        } catch (InterruptedException exc) {
            System.out.println("Wait interrupted\n" + exc);
        }
    }
}
```

Você pode encerrar o processo filho com o método **destroy()**. O próximo programa ilustra isso iniciando o **Bloco de Notas**, esperando dois segundos e encerrando-o via **destroy()**.

```
// Demonstra destroy().  
  
import java.io.IOException;  
  
class PBDemo3 {  
    public static void main(String[] args) {  
        try {  
            ProcessBuilder procBldr =  
                new ProcessBuilder("notepad", "testfile");  
  
            Process p = procBldr.start();  
  
            try {  
                Thread.sleep(2000);  
            } catch (InterruptedException exc) {  
                System.out.println("Sleep interrupted\n" + exc);  
            }  
  
            // encerra o processo.  
            p.destroy();  
            ← Encerra o processo.  
  
        } catch (IOException exc) {  
            System.out.println("Error executing notepad.\n" + exc);  
        }  
    }  
}
```

ProcessBuilder fornece vários outros recursos, como a possibilidade de acessar o ambiente de tempo de execução, configurar o diretório e redirecionar os fluxos padrão de entrada, saída e erro de um processo. Se tiver interesse, examine-os.

A CLASSE Runtime

A classe **Runtime** encapsula o ambiente de tempo de execução. Usando **Runtime**, você pode chamar vários métodos relacionados ao estado e ao comportamento da Máquina Virtual Java. Por exemplo, pode executar outros processos (embora com frequência **ProcessBuilder** seja a melhor opção), obter o número de processadores disponíveis, verificar níveis de memória, solicitar coleta de lixo e sair do programa, entre outras coisas.

Runtime não define nenhum construtor. Em vez disso, você pode obter uma referência ao ambiente de tempo de execução chamando o método **static Runtime.getRuntime()**, mostrado a seguir:

```
static Runtime getRuntime()
```

Assim, poderá acessar o ambiente de tempo de execução por intermédio da referência retornada.

Um exemplo que usa **Runtime** será examinado aqui. Embora Java forneça coleta de lixo automática, poderíamos querer saber quanta memória está disponível para o programa usar e quanto dela sobrará. Podemos usar essas informações, por exemplo, para verificar a eficiência do código ou saber aproximadamente quantos objetos adicionais de um determinado tipo podem ser instanciados. Para obter esses valores, use os métodos **totalMemory()** e **freeMemory()**, mostrados abaixo:

```
long totalMemory()
long freeMemory()
```

O método **totalMemory()** retorna o número total de bytes de memória disponíveis para a JVM. O método **freeMemory()** retorna o número aproximado de bytes de memória que não estão sendo usados atualmente.

Como mencionado na Parte I, a coleta de lixo Java é executada periodicamente para reciclar objetos não usados. No entanto, você também pode solicitar coleta de lixo. Uma maneira de fazer isso é chamando o método **gc()** de **Runtime**, mostrado abaixo:

```
void gc()
```

É preciso entender que, normalmente, não há necessidade de o programa chamar **gc()**, porque a JVM iniciará automaticamente a coleta de lixo quando considerar apropriado. Mas o método **gc()** permite que você solicite coleta de lixo em um momento de sua preferência.

Com o uso de **gc()** e **freeMemory()**, você pode obter informações sobre como seu programa usa memória. Por exemplo, para verificar o uso de memória de uma parte de seu programa, primeiro chame **gc()** seguido de uma chamada a **freeMemory()**. Isso fornecerá um padrão do uso de memória. Em seguida, execute a parte de seu código na qual está interessado e então faça outra chamada a **freeMemory()**. A diferença entre as duas chamadas a **freeMemory()** indica (aproximadamente) quanta memória foi alocada pelos objetos que estão sendo criados. O programa a seguir ilustra esse procedimento e também demonstra **totalMemory()**.

```
// Demonstra totalMemory(), freeMemory() e gc().

class MemoryDemo {
    public static void main(String[] args) {
        Runtime r = Runtime.getRuntime();
        long mem1, mem2;
        Integer[] someints = new Integer[1000];

        System.out.println("Total memory is: " +
                           r.totalMemory());
        mem1 = r.freeMemory();
        System.out.println("Initial free memory: " + mem1);
        r.gc();
        mem1 = r.freeMemory();
        System.out.println("Free memory after garbage collection: "
                           + mem1);
```

```

for(int i=0; i<1000; i++)
    someints[i] = new Integer(i); // aloca inteiros

mem2 = r.freeMemory();
System.out.println("Free memory after allocation: "
+ mem2);
System.out.println("Memory used by allocation: "
+ (mem1-mem2));

// descarta inteiros
for(int i=0; i<1000; i++) someints[i] = null;

r.gc(); // solicita coleta de lixo

mem2 = r.freeMemory();
System.out.println("Free memory after collecting" +
" discarded Integers: " + mem2);
}
}

```

Um exemplo da saída desse programa é mostrado aqui (é claro que seus resultados podem ser diferentes):

```

Total memory is: 16252928
Initial free memory: 15956816
Free memory after garbage collection: 16173656
Free memory after allocation: 15989984
Memory used by allocation: 183672
Free memory after collecting discarded Integers: 16173216

```

Verificação do progresso

1. Que classe é usada para iniciar um processo externo?
2. Quais os recursos da classe **Runtime**?

A CLASSE System

A classe **System** contém um conjunto de métodos e variáveis estáticas. Como você sabe, a entrada, a saída e a saída de erro padrão do tempo de execução Java são armazenadas nas variáveis **in**, **out** e **err** de **System**. Um resumo dos métodos definidos por **System** é mostrado na Tabela 23-2. Vários desses métodos podem lançar uma **SecurityException** se a operação solicitada não for permitida devido a restrições de segurança. Observe que **System** também fornece um método **gc()**, que é um membro **static**. Logo, pode ser chamado diretamente sem criar um objeto **System**. As próximas seções examinarão quatro usos comuns de **System**.

Respostas

1. **ProcessBuilder**
2. Os recursos de modificar e de dar informações sobre o ambiente de tempo de execução.

Tabela 23-2 Resumo dos métodos definidos por System

Método	Descrição
static void arraycopy(Object <i>origem</i> , int <i>índiceOrigem</i> , Object <i>destino</i> , int <i>índiceDestino</i> , int <i>tamanho</i>)	Copia um array. O array a ser copiado é passado em <i>origem</i> e o índice em cujo ponto a cópia começará dentro de <i>origem</i> é passado em <i>índiceOrigem</i> . O array que receberá a cópia é passado em <i>destino</i> e o índice em cujo ponto a cópia começará dentro de <i>destino</i> é passado em <i>índiceDestino</i> . <i>tamanho</i> é o número de elementos a serem copiados.
static long currentTimeMillis()	Retorna a hora atual em milissegundos desde meia-noite de primeiro de janeiro de 1970.
static void exit(int <i>códigoSaída</i>)	Interrompe a execução e retorna o valor de <i>códigoSaída</i> para o processo pai. Por convenção, 0 indica encerramento normal. Todos os outros valores indicam algum tipo de erro.
static void gc()	Solicita coleta de lixo.
static String getProperty(String <i>qual</i>)	Retorna a propriedade associada a <i>qual</i> . Um objeto null é retornado quando a propriedade desejada não é encontrada.
static long nanoTime()	Retorna um valor em nanossegundos desde algum ponto inicial não especificado.
static void setErr(PrintStream <i>fluxoErro</i>)	Define o fluxo padrão err para <i>fluxoErro</i> .
static void setIn(InputStream <i>fluxoEntrada</i>)	Define o fluxo padrão in para <i>fluxoEntrada</i> .
static void setOut(PrintStream <i>fluxoSaída</i>)	Define o fluxo padrão out para <i>fluxoSaída</i> .

Usando currentTimeMillis() para marcar o tempo de execução do programa

Uma das aplicações da classe **System** que você pode achar interessante é o uso do método **currentTimeMillis()** para marcar quanto tempo várias partes de seu programa demoram para ser executadas. O método **currentTimeMillis()** retorna a hora atual em milissegundos desde meia-noite de primeiro de janeiro de 1970. Para marcar o tempo de uma seção de seu programa, armazene esse valor imediatamente antes do começo da seção em questão. Logo após a conclusão, chame **currentTimeMillis()** novamente. O tempo decorrido será o tempo final menos o tempo inicial (mais a sobre carga das instruções que chamam **currentTimeMillis()**). O programa a seguir demonstra isso:

```
// Marcando o tempo de execução do programa.

class Elapsed {
    public static void main(String[] args) {
        long start, end;

        System.out.println("Timing a for loop from 0 to 100,000,000");

        // marca o tempo de um laço de 0 a 100.000.000
        start = System.currentTimeMillis(); // obtém a hora inicial
```

```

    for(long i=0; i < 100000000L; i++) ;
    end = System.currentTimeMillis(); // obtém a hora final

    System.out.println("Elapsed time: " + (end-start));
}
}

```

Veja um exemplo da saída (lembre-se de que provavelmente seus resultados serão diferentes):

```

Timing a for loop from 0 to 100,000,000
Elapsed time: 62

```

Se seu sistema tiver um relógio que ofereça precisão em nanosegundos, você poderá reescrever o programa anterior para usar **nanoTime()** em vez de **currentTimeMillis()**. Por exemplo, esta é a parte principal do programa reescrita para usar **nanoTime()**:

```

start = System.nanoTime(); // obtém a hora inicial
for(long i=0; i < 100000000L; i++) ;
end = System.nanoTime(); // obtém a hora final

```

Usando **arraycopy()**

O método **arraycopy()** pode ser usado para copiar rapidamente um array de qualquer tipo de um local para outro. Isso é muito mais rápido do que o laço equivalente escrito manualmente em Java. Veja um exemplo de dois arrays sendo copiados pelo método **arraycopy()**. Primeiro, **a** é copiado para **b**. Em seguida, todos os elementos de **a** são deslocados uma unidade para *baixo*. Então, **b** é deslocado uma unidade para *cima*.

```

// Usando arraycopy().

class ACDemo {
    static byte[] a = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
    static byte[] b = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };

    public static void main(String[] args) {
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
        System.arraycopy(a, 0, b, 0, a.length);
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
        System.arraycopy(a, 0, a, 1, a.length - 1);
        System.arraycopy(b, 1, b, 0, b.length - 1);
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
    }
}

```

Como você pode ver na saída a seguir, é possível copiar usando a mesma origem e destino nas duas direções:

```

a = ABCDEFGHIJ
b = MMMMMMM

```

```
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Obtendo valores de propriedades

Java define várias propriedades, chamadas de *propriedades do sistema*, que contêm informações úteis sobre o ambiente de execução. Os nomes das propriedades são mostrados abaixo:

file.separator	java.specification.version	java.vm.version
java.class.path	java.vendor	line.separator
java.class.version	java.vendor.url	os.arch
java.compiler	java.version	os.name
java.ext.dirs	java.vm.name	os.version
java.home	java.vm.specification.name	path.separator
java.io.tmpdir	java.vm.specification.vendor	user.dir
java.library.path	java.vm.specification.version	user.home
java.specification.name	java.vm.vendor	user.name
java.specification.vendor		

Você pode obter os valores dessas propriedades chamando o método **System.getProperty()**. Por exemplo, o programa a seguir exibe o caminho do diretório do usuário atual, a versão de Java, o nome do sistema operacional e o caractere usado como separador de arquivos:

```
class ShowProperties {
    public static void main(String[] args) {
        System.out.println(System.getProperty("user.dir"));
        System.out.println(System.getProperty("java.version"));
        System.out.println(System.getProperty("os.name"));
        System.out.println(System.getProperty("file.separator"));
    }
}
```

Redirecionando fluxos de I/O padrão

A classe **System** define os métodos **setIn()**, **setErr()** e **setOut()**, que permitem o redirecionamento de um fluxo padrão. Por exemplo, em vez de gravar no console, **System.out** poderia gravar em um arquivo. O próximo programa mostra um caso. Ele redireciona **System.out** para o arquivo cujo nome é especificado como argumento de linha de comando.

```
// Redireciona System.out para um arquivo.

import java.io.*;
```

```
class RedirectOut {
    public static void main(String[] args) throws IOException
    {

        // Primeiro, confirma se um arquivo foi especificado.
        if(args.length != 1) {
            System.out.println("RedirectOut: to");
            return;
        }

        // Cria um PrintStream vinculado ao arquivo especificado.
        try (PrintStream fout = new PrintStream(args[0]))
        {
            // salva o System.out original
            PrintStream orgOut = System.out;

            // redireciona System.out para o arquivo.
            System.setOut(fout); ← Redireciona System.out para fout.

            // observe que System.out é usado aqui
            System.out.println("This goes in the file.");

            // restaura o System.out original
            System.setOut(orgOut);

            System.out.println("This is shown on the screen.");
        } catch(IOException exc) {
            System.out.println("I/O Error: " + exc);
        }
    }
}
```

Há muitos pontos interessantes nesse programa. Primeiro, observe como um novo **PrintStream** é criado. **PrintStream** define vários construtores. O usado pelo programa foi:

`PrintStream(String nomearquivo) throws FileNotFoundException`

Ele abre (ou cria, se necessário) os arquivos especificados por *nomearquivo* para a saída. Se o arquivo não puder ser aberto, uma **FileNotFoundException** será lançada. Lembre-se, **FileNotFoundException** é subclasse de **IOException**. Como é aberto em uma instrução **try-with-resources**, o arquivo é fechado automaticamente quando saímos do bloco **try**.

Em seguida, o programa salva a referência ao fluxo **System.out** atual. Depois, usando **setOut()**, redireciona **System.out** para o arquivo e grava uma linha de texto. Essa saída será gravada no arquivo. Agora o **System.out** original é restaurado e uma linha de texto é gravada no console.

Verificação do progresso

1. Em um programa Java, você pode determinar se o programa está sendo executado em um computador Windows ou Mac. Verdadeiro ou falso?
2. O método da classe **System** para a cópia de um array se chama _____.
3. O que o método **currentTimeMillis()** da classe **System** retorna?

A CLASSE Object

Object é uma superclasse de todas as outras classes. Ela foi introduzida no Capítulo 7. Por conveniência, estamos mostrando os métodos definidos por **Object** na Tabela 23-3. A maioria já foi descrita em outros locais deste livro. No entanto, dois merecem uma discussão rápida. O primeiro é **clone()**, que retorna uma cópia do objeto em que é chamado. A classe do objeto chamador deve implementar a interface **Cloneable**. Caso contrário, uma **CloneNotSupportedException** será lançada. Essa interface não tem membros; ela apenas indica que a classe permite que instâncias sejam clonadas. A clonagem é arriscada porque pode causar efeitos colaterais indesejados. Por exemplo, a versão de **Object** para **clone()** cria um objeto cujos membros são iguais aos do objeto original. Logo, se um membro for uma referência a algum outro objeto, tanto o original quanto o clone referenciarão o mesmo objeto e os dois poderão alterar seu estado. Portanto, como regra geral, **clone()** precisa ser sobreposto para que esses problemas sejam evitados. Já que a clonagem é uma técnica especializada, ela não será mais discutida neste livro.

O segundo método é **hashCode()**. Esse método retorna um *código de hash* para o objeto chamador. Em termos mais simples, um código de hash é um valor derivado de algum aspecto do objeto. Os códigos de hash são especialmente úteis no armazenamento de objetos em alguns tipos de coleções. (Consulte o Capítulo 25.)

A CLASSE Class

Class encapsula informações sobre uma classe ou interface. Objetos de tipo **Class** são criados automaticamente quando as classes são carregadas. Não podemos declarar um objeto **Class** de maneira explícita. Geralmente, um objeto **Class** é obtido com uma chamada ao método **getClass()** definido por **Object**. **Class** é um tipo genérico declarado como mostrado abaixo:

```
class Class<T>
```

Aqui, **T** é o tipo da classe ou interface representada.

Com frequência os métodos definidos por **Class** são úteis em situações em que informações de tipo de um objeto são necessárias no tempo de execução. São forneci-

Respostas:

1. Verdadeiro. Você pode fazer isso chamando **System.getProperty("os.name")**.
2. **arraycopy**
3. Um valor **long** informando o tempo em milissegundos desde meia-noite de primeiro de janeiro de 1970.

Tabela 23-3 Métodos definidos por Object

Método	Descrição
Object clone() throws CloneNotSupportedException	Cria um novo objeto igual ao objeto chamador.
boolean equals(Object <i>objeto</i>)	Retorna true se o objeto chamador for equivalente a <i>objeto</i> .
void finalize() throws Throwable	Método finalize() padrão. É chamado antes de um objeto não usado ser reciclado.
final Class<?> getClass()	Obtém um objeto Class que descreve o objeto chamador.
int hashCode()	Retorna o código de hash associado ao objeto chamador.
final void notify()	Retoma a execução de uma thread que está esperando no objeto chamador.
final void notifyAll()	Retoma a execução de todas as threads que estão esperando no objeto chamador.
String toString()	Retorna um string que descreve o objeto.
final void wait() throws InterruptedException	Espera outra thread de execução.
final void wait(long <i>milissegundos</i>) throws InterruptedException	Espera outra thread de execução durante os <i>milissegundos</i> especificados.
final void wait(long <i>milissegundos</i> , int <i>nanossegundos</i>) throws InterruptedException	Espera outra thread de execução durante os <i>milissegundos e nanossegundos</i> especificados.

dos métodos que nos permitem saber informações adicionais sobre uma classe específica, como seus campos, métodos e construtores públicos. Também podemos obter informações sobre interfaces. Entre outras coisas, isso é importante para os recursos Java de reflexão.

Embora um uso mais sofisticado de **Class** não faça parte do escopo deste livro, o programa a seguir demonstrará dois de seus recursos. Ele usa **getClass()** (herdado de **Object**) para obter uma referência **Class**. Em seguida, demonstra os métodos **getName()** e **getSuperclass()** definidos por **Class**. Esses métodos são mostrados aqui:

```
String getName()
Class<? super T> getSuperclass()
```

O método **getName()** retorna o nome da classe representada pela instância de **Class** que o chamou. O método **getSuperclass()** retorna uma referência a um objeto **Class** que representa a superclasse do objeto **Class** que o chamou.

```
// Demonstra a obtenção de informações de tipo no tempo de execução.

class X {
    int a;
    float b;
}

class Y extends X {
    double c;
```

```

}

class RTTI {
    public static void main(String[] args) {
        X x = new X();
        Y y = new Y();
        Class<?> clObj;

        clObj = x.getClass(); // obtém referência Class
        System.out.println("x is object of type: " +
                           clObj.getName());

        clObj = y.getClass(); // obtém referência Class
        System.out.println("y is object of type: " +
                           clObj.getName());
        clObj = clObj.getSuperclass(); ← Observe como a superclasse é obtida.
        System.out.println("y's superclass is " +
                           clObj.getName());
    }
}

```

A saída desse programa é:

```

x is object of type: X
y is object of type: Y
y's superclass is X

```

A CLASSE Enum

Como descrito no Capítulo 13, as enumerações são integradas à linguagem Java pela palavra-chave **enum**. Todas herdam automaticamente **Enum**. **Enum** é uma classe genérica declarada dessa forma:

```
abstract class Enum<E extends Enum<E>>
```

Aqui, **E** representa o tipo de enumeração. **Enum** não tem construtores públicos. Ela foi apresentada no Capítulo 13.

Além dos métodos descritos no Capítulo 13, há mais um que você pode achar interessante:

```
static <T extends Enum<T>> T valueOf(Class<T> tipo-e, String nome)
```

O método **valueOf()** retorna a constante associada a *nome* do tipo de enumeração especificado por *tipo-e*.

CLASSES RELACIONADAS A THREADS E A INTERFACE Runnable

As principais classes Java relacionadas a threads ficam no pacote **java.lang**. Elas são:

Runnable

Thread

ThreadGroup
ThreadLocal
InheritableThreadLocal

Duas delas, a interface **Runnable** e a classe **Thread**, já foram descritas detalhadamente no Capítulo 12. Um **ThreadGroup** cria um grupo de threads. Os grupos de threads oferecem uma maneira conveniente de gerenciarmos várias threads como uma unidade. **ThreadLocal** é usada na criação de variáveis locais de thread. Cada thread terá sua própria cópia de uma variável local de thread. **InheritableThreadLocal** cria variáveis locais de thread que podem ser herdadas.

OUTRAS CLASSES

Além das classes que acabamos de descrever e das descritas em outros locais deste livro, **java.lang** inclui várias outras. Elas estão resumidas aqui:

ClassLoader	Uma classe abstrata que determina como as classes são carregadas.
ClassValue<T>	Essa é uma classe de uso especial adicionada pelo JDK 7 que associa um valor a um tipo.
Compiler	Dá suporte a ambientes em que o bytecode Java é compilado para código executado em vez de ser interpretado. Não é usada na programação comum.
Package	Encapsula informações sobre um pacote. Entre outras coisas, essas informações incluem elementos como seu nome, versão, fornecedor e anotações.
RuntimePermission	Relacionada ao mecanismo Java de segurança. Encapsula uma permissão disponível no tempo de execução.
SecurityManager	Define um gerenciador de segurança.
StackTraceElement	Descreve um quadro de pilha, que é um elemento individual de um rastreamento de pilha quando ocorre uma exceção.
StrictMath	Define um conjunto completo de métodos matemáticos equivalentes aos de Math . A diferença é que a versão de StrictMath garante a geração de resultados idênticos em todas as implementações, enquanto os métodos de Math têm uma amplitude maior que visa a melhoria do desempenho.
Void	Representa o tipo void .

AS INTERFACES DE **java.lang**

O pacote **java.lang** define oito interfaces de nível superior. A interface **Runnable** foi descrita anteriormente neste livro, no Capítulo 12, quando o uso de várias threads foi descrito. A interface **Cloneable** foi mencionada quando a classe **Object** foi descrita. As outras serão examinadas agora.

A interface Comparable

Objetos de classes que implementam **Comparable** podem ser ordenados. Em outras palavras, classes que implementam **Comparable** contêm objetos que podem ser comparados de alguma maneira significativa. **Comparable** é genérica e é declarada assim:

```
interface Comparable<T>
```

Aqui, **T** representa o tipo dos objetos que estão sendo comparados.

A interface **Comparable** declara um método que é usado para determinar que Java chama de *ordem natural* das instâncias de uma classe. A forma geral do método é esta:

```
int compareTo(T obj)
```

Esse método compara o objeto chamador com *obj*. Ele retorna 0 se os valores forem iguais. Um valor negativo é retornado quando o objeto chamador tem um valor menor. Caso contrário, um valor positivo é retornado.

Como regra geral, o resultado da comparação de dois objetos com **compareTo()** deve ser compatível com o resultado da comparação desses dois objetos com **equals()**. Portanto, se os dois objetos forem iguais de acordo com **equals()**, então **compareTo()** deve retornar zero e vice-versa.

Essa interface é implementada por várias das classes já discutidas neste livro, como **Byte**, **Character**, **Double**, **Float**, **Long**, **Short**, **String** e **Integer**.

TENTE ISTO 23-1 Crie um método genérico para encontrar o menor valor em um array

`Finder.java`
`MinElementDemo.java`

Suponhamos que você tivesse um array e quisesse encontrar o menor elemento. Isso é fácil quando os elementos são inteiros. (Consulte o exemplo **MinMax** do Capítulo 5.) Também é fácil quando os elementos são strings onde o “menor” string é o que vem primeiro em ordem alfabética. Por exemplo, veja uma maneira de criar um método que encontra o menor elemento de um array de strings não vazio:

```
String minStr(String[] data) {
    String currMin = data[0];

    for(String s : data)
        if(s.compareTo(currMin) < 0) currMin = s;

    return currMin;
}
```

Esse método funciona percorrendo o array com um laço, comparando os valores e registrando o menor valor encontrado. Após o término do laço, o valor de **currMin** será o menor valor, que será retornado. Observe que os strings são comparados com o uso do método **compareTo()**, que é especificado pela interface **Comparable** e implementado por **String**. (Consulte o Capítulo 22.)

Podemos usar a mesma abordagem básica para outros tipos de arrays, como arrays **Integer** ou arrays de tipos definidos pelo usuário? Mais precisamente, podemos adaptar a lógica de **minStr()** para criar um método genérico que encontre o menor valor em diferentes tipos de arrays? A resposta é sim se o tipo dos elementos do array implementar a interface **Comparable**. Se uma classe implementar essa interface, poderemos usar o método **compareTo()** para comparar objetos dessa classe.

Nesse projeto, criaremos um método genérico chamado **minElement()** para encontrar o menor elemento de qualquer tipo de array, contanto que o tipo do elemento implemente a interface **Comparable**.

PASSO A PASSO

- Crie um novo arquivo chamado **Finder.java**. O método genérico **minElement()** ficará contido em uma classe chamada **Finder**. Além disso, **minElement()** será **static**, para que nenhum objeto de tipo **Finder** precise ser instanciado. Isso tornará seu uso mais conveniente. Insira o código a seguir em **Finder.java**.

```
class Finder {

    public static <T extends Comparable<T>> T minElement(T[] data) {
        T currMin = data[0];

        for(T x : data) {
            if (x.compareTo(currMin) < 0)
                currMin = x;
        }
        return currMin;
    }
}
```

- Examinemos o método **minElement()** detalhadamente. Ele se parece muito com o método **minStr()** que vimos acima, exceto por ser um método genérico com parâmetro de tipo **T**. O tipo **T** pode ser qualquer tipo, desde que implemente a interface **Comparable<T>**. Ou seja, elementos de tipo **T** têm que poder ser comparados com o uso do método **compareTo()**.

Já que a classe **String** implementa **Comparable**, o método **minElement()** pode substituir o método **minStr()** mostrado acima. Os encapsuladores de tipos, como **Integer** e **Double**, também implementam **Comparable**. Portanto, você pode usar **minElement()** para encontrar o menor valor em arrays de objetos **Integer**, por exemplo. No geral, **minElement()** pode ser usado para encontrar o menor elemento de qualquer array não vazio, inclusive os de tipos definidos pelo usuário, contanto que o tipo do elemento implemente **Comparable**.

3. Para ver como **minElement()** funciona com arrays de inteiros, arrays de strings e arrays de **Doubles**, crie um arquivo chamado **MinElementDemo.java** contendo o código a seguir:

```
class MinElementDemo {
    public static void main(String[] args) {
        Integer[] intArray = { 3, 1, 4, 3, 6, 5 };
        int intMin = Finder.minElement(intArray);
        System.out.println(intMin);

        String[] strArray = {"every", "good", "boy", "does", "fine"};
        String strMin = Finder.minElement(strArray);
        System.out.println(strMin);

        Double[] doubleArray = {3.14, 2.8, 6.023, 1.414};
        Double doubleMin = Finder.minElement(doubleArray);
        System.out.println(doubleMin);
    }
}
```

Observe que o método **minElement()** é chamado três vezes, cada uma com um array de tipo diferente. Quando você executar o método **main()** na classe **MinDemo**, verá a saída abaixo:

```
1
boy
1.414
```

4. Você pode usar a mesma abordagem básica usada por **minElement()** sempre que criar um método genérico que precise comparar objetos. Apenas especifique que o parâmetro de tipo estende **Comparable**. Em seguida, use **compareTo()** para comparar os objetos.

A interface Appendable

Objetos de uma classe que implemente **Appendable** podem ter um caractere ou uma sequência de caracteres acrescida a eles. **Appendable** define estes três métodos:

```
Appendable append(char car) throws IOException
Appendable append(CharSequence cars) throws IOException
Appendable append(CharSequence cars, int índice, int fim)
    throws IOException
```

Na primeira forma, o caractere *car* é acrescido ao objeto chamador. Na segunda, a sequência de caracteres *cars* é acrescida ao objeto. A terceira forma permite a indicação de uma parte (os caracteres compreendidos entre *índice* e *fim*-1) da sequência especificada por *cars*. Em todos os casos, uma referência ao objeto chamador é retornada.

A interface Iterable

Iterable deve ser implementada por qualquer classe cujos objetos sejam usados pela versão for-each do laço **for**. Em outras palavras, para um objeto ser usado dentro de

um laço **for** de estilo for-each, sua classe deve implementar **Iterable**. **Iterable** é uma interface genérica que tem esta declaração:

```
interface Iterable<T>
```

Aqui, **T** é o tipo dos elementos que estão sendo percorridos. A interface define um método, **iterator()**, que pode ser visto abaixo:

```
Iterator<T> iterator()
```

Ele retorna um iterador para os elementos contidos no objeto chamador.

Nota: Os iteradores são descritos com detalhes no Capítulo 25.

A interface Readable

A interface **Readable** indica que um objeto pode ser usado como fonte de caracteres. Ela define um método chamado **read()**, que é mostrado aqui:

```
int read(CharBuffer buf) throws IOException
```

Esse método lê caracteres em *buf*. Ele retorna o número de caracteres lidos, ou -1 se o objeto chamador estiver vazio.

A interface CharSequence

A interface **CharSequence** define métodos que lêem uma sequência de caracteres. Esses métodos são mostrados abaixo:

```
char charAt(int índice)
int length()
CharSequence subSequence(int índiceInicial, int índiceFinal)
String toString()
```

O método **charAt()** retorna o caractere do índice especificado por *índice*. O número de caracteres da sequência chamadora é retornado por **length()**. Um subconjunto da sequência chamadora começando em *índiceInicial* e terminando em *índiceFinal* – 1 é retornado por **subSequence()**. Para concluir, **toString()** retorna um objeto **String** equivalente à sequência chamadora.

Pergunte ao especialista

P A interface **CharSequence** foi mencionada no último capítulo, junto com a classe **String**. Que outras classes implementam essa interface e por que ela é importante?

R Ela é implementada pelas classes **CharBuffer**, **Segment**, **String**, **StringBuffer** e **StringBuilder** e usada em vários pacotes do JDK. A importância dessa interface, assim como de todas as outras, é que ela ajuda a reduzir a vinculação, como descrito no Capítulo 16. Isto é, classes que usam essa interface são menos interligadas, logo, são mais flexíveis e reutilizáveis.

Por exemplo, um dos métodos `replace()` da classe `String` mencionado no capítulo anterior tem a forma geral

```
String replace(CharSequence original, CharSequence substituto)
```

Além de usar objetos da classe `String` como seus parâmetros, esse método usará objetos de qualquer classe que implemente a interface, como `StringBuffer` e `StringBuilder`. Portanto, o método é muito mais útil do que seria se não tivesse usado a interface `CharSequence`.

Mais uma coisa: o método `subSequence()` da interface `CharSequence` retorna um `CharSequence` em vez de um objeto da classe `String` ou de outra classe. A vantagem disso é, novamente, a flexibilidade obtida. Nesse caso, a flexibilidade beneficia o implementador do método, que pode optar por criar e retornar qualquer tipo de objeto que deseje, contanto que implemente a interface.

A interface AutoCloseable

`AutoCloseable` foi adicionada pelo JDK 7 e dá suporte à instrução `try-with-resources`, que implementa o que às vezes é chamado de *gerenciamento automático de recursos* (ARM, *automatic resource management*). A instrução `try-with-resources` automatiza o processo de liberação de um recurso (como um fluxo) quando ele não é mais necessário. (Consulte o Capítulo 11 para ver detalhes.) Só objetos de classes que implementam `AutoCloseable` podem ser usados com `try-with-resources`. A interface `AutoCloseable` só define o método `close()`, mostrado abaixo:

```
void close() throws Exception
```

Esse método fecha o objeto chamador, liberando qualquer recurso que ele possa conter. Ele é chamado automaticamente no fim de uma instrução `try-with-resources`, eliminando assim a necessidade de chamarmos `close()` explicitamente. `AutoCloseable` é implementada por várias classes, inclusive todas as classes de I/O que abrem um fluxo que pode ser fechado.

Verificação do progresso

1. Que métodos você chama para obter a classe e a superclasse de um objeto?
2. Que interface declara um método usado para determinar a ordem “natural” dos objetos de uma classe?
3. Que interface uma classe deve implementar se quiser permitir o uso de laços `for-each` para o acesso a dados de objetos dessa classe?

Respostas:

1. `getClass()` e `getSuperclass()`
2. `Comparable<T>`
3. `Iterable<T>`

EXERCÍCIOS

1. As classes usadas para encapsular valores primitivos para que eles possam ser usados onde apenas objetos são permitidos são chamadas de classes _____.
 2. O infinito não é um número. Mais precisamente, para um **double d**, se **isInfinite(d)** for **true**, então **isNaN(d)** será **true**. Verdadeiro ou falso?
 3. **Double.MIN_VALUE** é o menor valor inteiro positivo (a saber, 1). Verdadeiro ou falso?
 4. Determine os valores das expressões a seguir. As constantes são todas definidas na classe **Double**:
 - A. MAX_VALUE / MIN_VALUE
 - B. MAX_VALUE * MAX_VALUE
 - C. POSITIVE_INFINITY / POSITIVE_INFINITY
 - D. POSITIVE_INFINITY / NEGATIVE_INFINITY
 - E. POSITIVE_INFINITY – POSITIVE_INFINITY
 - F. POSITIVE_INFINITY – NEGATIVE_INFINITY
 - G. NaN – NaN
 5. Para qualquer caractere **ch**, a expressão
- ```
| Character.isUpperCase(Character.toUpperCase(ch))
```
- produz **true**. Verdadeiro ou falso?
6. Para qualquer **double d**, a expressão **Math.ceil(Math.floor(d))** é igual à expressão **Math.floor(Math.ceil(d))**. Verdadeiro ou falso?
  7. Para iniciar outro programa (outro processo pesado) em um programa Java, você deve chamar o construtor da classe **Process**. Verdadeiro ou falso?
  8. O método **nanoTime()** da classe **System** retorna o tempo em nanossegundos tomando como base meia-noite de primeiro de janeiro de 1970. Verdadeiro ou falso?
  9. O método **getName()** de **Class** retorna apenas o nome da classe sem o pacote ao qual ela pertence. Verdadeiro ou falso?
  10. Crie um método **numWords()** que use um string como seu parâmetro. Ele calcula e retorna o número de palavras do string. No caso em questão, uma *palavra* é um substring composto por um ou mais caracteres que não sejam espaços em branco separado de outras palavras por um ou mais caracteres de espaço em branco. Percorra os caracteres do string com um laço e use o método **isWhiteSpace()** da classe **Character**.
  11. Crie um método **toAllUpperCase()** que use um string como seu parâmetro. Ele retorna uma cópia do string em que todas as letras são maiúsculas. Use o método **toUpperCase()** da classe **Character** e não o da classe **String**.
  12. A expressão a seguir será compilada? Em caso negativo, por que não? Se for, qual é o valor da expressão?

```
| new Boolean(Boolean.parseBoolean(new
| | Boolean("false").toString())).equals(Boolean.FALSE);
```

13. Modifique a classe **PBDemo2** para que seu método **main()** inicie um navegador Web em seu computador e abra sua página favorita.
14. Crie um programa semelhante a **RedirectOut** que, em vez de redirecionar **System.out**, redirecione **System.in** para obter dados em um arquivo cujo nome é especificado como argumento de linha de comando.
15. Crie um método **getLineage()** que use qualquer objeto como seu parâmetro. Ele retorna um array de strings contendo o nome da classe do objeto, o nome de sua superclasse, o nome da superclasse da superclasse e assim por diante, indo até o topo da hierarquia e incluindo a classe **Object**.
16. Escreva um programa que crie dois arrays com 10 milhões de inteiros ou mais (supondo que seu sistema possa tratar arrays tão grandes). Em seguida, meça quanto tempo (em milissegundos) ele leva para copiar os dados do primeiro array para o segundo usando duas abordagens diferentes: usando um laço **for** e usando o método **arraycopy()** da classe **System**. Agora exiba quanto tempo cada abordagem demora.
17. Adicione um método **maxElement()** à classe **Finder** definida na seção Tente isto 23-1. Ele é idêntico ao método **minElement()** dessa classe, exceto por retornar o maior elemento do array e não o menor.
18. O programa **MemoryDemo** mostrou que um array de 1.000 **Integers** ocupa cerca de 183.672 bytes de memória. No entanto, uma variável **int** só ocupa 4 bytes. Logo, o array não deveria ter ocupado apenas 4.000 bytes? Por que ocupou mais?
19. Use a classe **Math** para criar um programa que demonstre as igualdades a seguir. Mais precisamente, o programa deve calcular e exibir os valores do lado esquerdo de cada igualdade. Para fins de comparação, ele também deve exibir os valores do lado direito da igualdade. Lembre-se, a constante matemática  $e$  é representada por **Math.E** e  $\pi$  é representado por **Math.PI** em Java.
  - A.  $e^{\log(2.0)} = 2,0$
  - B.  $\log(e^{2.0}) = 2,0$
  - C.  $\sin^2(\pi/3) + \cos^2(\pi/3) = 1,0$
  - D.  $\tan(\arctan(\pi/3)) = \pi/3$
20. Implemente uma função **roundDownTenth()** que use um valor **double** como seu parâmetro. Ela arredonda o valor para baixo até a dezena mais próxima e o retorna. Por exemplo, **roundDownTenth(3.16)** retorna 3,1 e **roundDownTenth(-3.51)** retorna -3,6.
21. Como mencionado neste capítulo, a classe **Math** declara duas constantes **static final**: **E** e **PI**. Se você tivesse que calcular a área de um círculo dado seu raio **r**, qual das expressões a seguir seria melhor e por quê?
  - a. **Math.PI \* r \* r**
  - b. **3.141592653589793238462643 \* r \* r**

# 24

## Examinando o pacote `java.util`

### PRINCIPAIS HABILIDADES E CONCEITOS

- Trabalhar com localidades usando a classe **Locale**
- Trabalhar com data e hora com as classes **Date**, **Calendar** e **GregorianCalendar**
- Formatar saídas com a classe **Formatter**
- Usar o método **printf( )** para formatar
- Ler entradas formatadas com a classe **Scanner**
- Gerar números pseudoaleatórios com a classe **Random**
- Usar a classe **Observable** e a interface **Observer**
- Agendar tarefas usando as classes **Timer** e **TimerTask**

O pacote **java.util** define classes e interfaces que atendem várias tarefas comumente encontradas quando se programa. Por exemplo, ele contém o *Collections Framework*. O Collections Framework é um dos subsistemas mais poderosos de Java e dá suporte a uma sofisticada hierarquia de interfaces e classes que gerenciam grupos de objetos. O pacote **java.util** também fornece várias classes utilitárias, inclusive as que formatam dados para saída, leem dados formatados e trabalham com data e hora.

Como **java.util** dá suporte a um conjunto tão amplo de funcionalidades, é um pacote muito extenso. Ele inclui as seguintes classes de primeiro nível:

|                        |                   |                        |
|------------------------|-------------------|------------------------|
| AbstractCollection     | EventObject       | PropertyResourceBundle |
| AbstractList           | FormattableFlags  | Random                 |
| AbstractMap            | Formatter         | ResourceBundle         |
| AbstractQueue          | GregorianCalendar | Scanner                |
| AbstractSequentialList | HashMap           | ServiceLoader          |
| AbstractSet            | HashSet           | SimpleTimeZone         |
| ArrayQueue             | Hashtable         | Stack                  |
| ArrayList              | IdentityHashMap   | StringTokenizer        |
| Arrays                 | LinkedHashMap     | Timer                  |

|                     |                    |             |
|---------------------|--------------------|-------------|
| BitSet              | LinkedHashSet      | TimerTask   |
| Calendar            | LinkedList         | TimeZone    |
| Collections         | ListResourceBundle | TreeMap     |
| Currency            | Locale             | TreeSet     |
| Date                | Objects            | UUID        |
| Dictionary          | Observable         | Vector      |
| EnumMap             | PriorityQueue      | WeakHashMap |
| EnumSet             | Properties         |             |
| EventyListenerProxy | PropertyPermission |             |

As interfaces de primeiro nível definidas por **java.util** são as mostradas abaixo:

|             |               |              |
|-------------|---------------|--------------|
| Collection  | Comparator    | Deque        |
| Enumeration | EventListener | Formattable  |
| Iterator    | List          | ListIterator |
| Map         | NavigableMap  | NavigableSet |
| Observer    | Queue         | RandomAccess |
| Set         | SortedMap     | SortedSet    |

O pacote **java.util** é muito grande, mas seu tamanho é gerenciável porque as classes e interfaces que compõem o Collections Framework podem ser examinadas separadamente, como um grupo. Portanto, a descrição de **java.util** será dividida em dois capítulos. Este abordará as partes que não pertencem ao Collections Framework, o qual será abordado no próximo capítulo.

O assunto principal deste capítulo serão as classes a seguir:

- **Locale**
- **Date, Calendar e GregorianCalendar**
- **Formatter e Scanner**

A classe **Locale** encapsula informações relacionadas a uma região geográfica, política ou cultural. Ela ajuda na internacionalização. **Date, Calendar e GregorianCalendar** gerenciam data e hora. A classe **Formatter** formata dados e a classe **Scanner** lê dados formatados. Também serão abordadas as classes

- **Random**
- **Observer e Observable**
- **Timer e TimerTask**

**Random** é usada para produzir uma sequência de números pseudoaleatórios. **Observer e Observable** dão suporte ao padrão Observer descrito no Capítulo 16. **Timer e TimerTask** permitem a execução de uma thread em algum momento futuro. As classes e interfaces restantes que não fazem parte do Collections Framework se encontram em um resumo no fim deste capítulo.

## A CLASSE Locale

A classe **Locale** encapsula informações relacionadas a convenções geopolíticas ou culturais. É uma das várias classes que nos ajudam a criar programas que podem ser internacionalizados. A internacionalização é importante para muitos programas comerciais porque, com frequência, eles têm aspectos que dependem da região. É claro que uma das dependências regionais é o idioma usado nas mensagens. No entanto, outros aspectos de um programa também dependem da região. Por exemplo, os formatos usados na exibição de datas, horas e números às vezes diferem entre países ou idiomas. A classe **Locale** ajuda a gerenciar as diferenças entre regiões. Embora a internacionalização seja um tópico muito extenso que não faz parte do escopo deste livro, essa classe fornece um ponto de partida.

A classe **Locale** implementa as interfaces **Serializable** e **Cloneable**. Aqui temos dois construtores de **Locale**:

```
Locale(String idioma)
Locale(String idioma, String país)
```

Eles constroem um objeto **Locale** para representar um idioma específico. O parâmetro *idioma* recebe um *código de idioma*, que deve estar de acordo com a ISO 639 (Códigos para a Representação de Nomes de Idiomas). Exemplos são en, de e fr, que correspondem a inglês, alemão e francês, respectivamente. O parâmetro *país* especifica um código de país, que pode ser um dos códigos definidos pela ISO 3166 (Códigos para a Representação de Nomes de Países e suas Subdivisões). Exemplos são US, DE e FR para Estados Unidos, Alemanha e França, respectivamente.

*Nota:* ISO significa International Organization for Standardization.

Por exemplo, a linha a seguir constrói um objeto **Locale** que representa as convenções do inglês dos Estados Unidos.

```
| Locale usLocale = new Locale("en", "US");
```

Embora o construtor de **Locale** não seja difícil de usar, a classe fornece uma alternativa conveniente para muitas regiões comuns. Ela define as constantes abaixo, que são objetos **Locale static final**. Você pode usar uma quando um objeto regional correspondente ao nome for necessário.

|               |          |                     |
|---------------|----------|---------------------|
| CANADA        | GERMAN   | KOREAN              |
| CANADA_FRENCH | GERMANY  | PRC                 |
| CHINA         | ITALIAN  | SIMPLIFIED_CHINESE  |
| CHINESE       | ITALY    | TAIWAN              |
| ENGLISH       | JAPAN    | TRADITIONAL_CHINESE |
| FRANCE        | JAPANESE | UK                  |
| FRENCH        | KOREA    | US                  |

Por exemplo, a constante **Locale.CANADA** representa o objeto **Locale** do Canadá, a constante **Locale.GERMAN** representa o objeto **Locale** do idioma alemão e a constante **Locale.GERMANY** representa o objeto **Locale** da Alemanha.

Você pode obter o **Locale** padrão usado pela JVM chamando o método **Locale.getDefault()** mostrado aqui:

```
static Locale getDefault()
```

A localidade padrão reflete as configurações usadas pelo sistema em que a JVM está sendo executada. É a localidade usada quando nenhuma outra é especificada explicitamente.

A classe **Locale** define vários métodos que obtêm informações sobre uma instância de **Locale**. Abaixo temos três exemplos:

```
final String getDisplayCountry()
final String getDisplayLanguage()
final String getDisplayName()
```

Eles retornam strings da localidade que podem ser usados para exibir o nome do país, o nome do idioma e o nome completo da localidade.

Agora veremos um exemplo que demonstra **Locale**. Primeiro, ele obtém a localidade atual, que nesse caso são os Estados Unidos. Em seguida, exibe o país, o idioma e o nome. Depois, o programa cria manualmente uma localidade para a Alemanha e exibe o país, o idioma e o nome. Para concluir, exibe o país, o idioma e o nome associados à localidade padrão, **Locale.FRANCE**.

```
// Demonstra Locale.
import java.util.*;

class LocaleDemo {
 public static void main(String[] args) { ← Obtém a localidade padrão.

 // obtém a localidade padrão.
 Locale defLocale = Locale.getDefault();

 // exibe o nome, o país e o idioma
 System.out.println("Default locale: ");
 System.out.println("Name: " + defLocale.getDisplayName());
 System.out.println("Country: " + defLocale.getDisplayCountry());
 System.out.println("Language: " + defLocale.getDisplayLanguage());

 System.out.println();

 // cria manualmente uma localidade para a Alemanha Cria uma localidade
 Locale germanLocale = new Locale("de", "DE"); ← para a Alemanha.
 System.out.println("German locale: ");
 System.out.println("Name: " + germanLocale.getDisplayName());
 System.out.println("Country: " +
 germanLocale.getDisplayCountry());
 System.out.println("Language: " +
 germanLocale.getDisplayLanguage());

 System.out.println();
 }
}
```

```
// agora, usa a localidade padrão para a França.
System.out.println("Locale.FRANCE: ");
System.out.println("Name: " +
 Locale.FRANCE.getDisplayName()); ← Usa a localidade
System.out.println("Country: " +
 Locale.FRANCE.getDisplayCountry());
System.out.println("Language: " +
 Locale.FRANCE.getDisplayLanguage());
}
}
```

Um exemplo da saída é mostrado aqui:

```
Default locale:
Name: English (United States)
Country: United States
Language: English

German locale:
Name: German (Germany)
Country: Germany
Language: German

Locale.FRANCE:
Name: French (France)
Country: France
Language: French
```

Várias classes da biblioteca Java de APIs dependem da localidade. Elas incluem **Calendar** e **GregorianCalendar**, descritas na próxima seção, e **Formatter**, descrita um pouco depois.

## Pergunte ao especialista

**P** Estava examinando a documentação Java online sobre **Locale** e notei que ela menciona as abreviações BCP 47 e UTS 35. O que são elas?

**R** O JDK 7 adicionou atualizações significativas à classe **Locale**. Entre elas está o suporte ao BCP 47 da Internet Engineering Task Force (IETF), que define tags para a identificação de idiomas, e o Unicode Technical Standard (UTS) 35, que define a Locale Data Markup Language (LDML). Como consequência, vários métodos novos foram adicionados a **Locale**. Também foi adicionada a classe **Locale.Builder**, que constrói instâncias de **Locale**. Ela assegura que uma especificação de localidade seja bem formada como definido pelo BCP 47. (Os construtores de **Locale** não fornecem essa verificação.)

## Verificação do progresso

1. A constante de **Locale** que representa os Estados Unidos é **Locale.\_\_\_\_\_**.
2. Para obter a localidade padrão usada pela JVM em um sistema, use o método **static \_\_\_\_\_** de **Locale**.

## TRABALHANDO COM DATA E HORA

É comum um programa ter que usar data e hora. Vejamos alguns exemplos. Um aplicativo de inserção de pedidos poderia fornecer um *time stamp* que indicasse quando um pedido foi inserido. Um programa de controle de estoque poderia incluir um campo que indicasse para quando é esperada uma nova remessa de itens. Você poderia querer exibir a data e hora atuais como cortesia para o usuário. Qualquer que seja a razão, Java dá um suporte significativo para data e hora por intermédio de várias classes. As discutidas aqui serão **Date**, **Calendar** e **GregorianCalendar**.

### Date

A classe **Date** encapsula data e hora e fazia parte da versão 1.0 de Java. Ela é menos usada do que se poderia supor. Isso porque, quando Java 1.1 foi lançada, muitas das funções executadas pela classe **Date** original foram integradas a **Calendar** e **Date-Format** (que faz parte de `java.text`). Como resultado, muitos dos métodos da classe **Date** da versão 1.0 foram substituídos. Na linguagem Java, substituir significa ainda com suporte, mas obsoleto. Já que os métodos substituídos da versão 1.0 não devem ser usados em códigos novos, eles não serão descritos aqui.

**Date** dá suporte aos construtores não obsoletos a seguir:

```
Date()
Date(long milisseg)
```

O primeiro construtor inicializa o objeto com a data e a hora atuais fornecidas pela máquina. O segundo aceita um argumento que é igual ao número de milissegundos decorridos desde meia-noite de primeiro de janeiro de 1970. **Date** implementa as interfaces **Comparable**, **Serializable** e **Cloneable**.

**Date** dá suporte a alguns métodos obsoletos. Por exemplo, você pode obter data e hora em milissegundos a partir de primeiro de janeiro de 1970 chamando o método **getTime()**. Também pode definir data e hora chamando **setTime()** e passando o número de milissegundos desde primeiro de janeiro de 1970. Você pode comparar objetos **Date** usando **equals()**, **compareTo()**, **after()** e **before()**. O método **toString()** foi sobreposto para exibir uma forma simples de data e hora.

Como explicado, muitos dos métodos definidos por **Date** foram substituídos. Esses métodos davam acesso aos componentes individuais de data e hora, como hora, dia ou ano. No entanto, já que foram substituídos, não devem ser usados. Em vez

---

Resposta:

1. US
2. `getDefault()`

disso, para obter esse tipo de informação, você usará a classe **Calendar**, descrita a seguir.

## Calendar e GregorianCalendar

A maneira moderna de trabalhar com data e hora em Java é usando a classe abstrata **Calendar** e sua subclasse concreta **GregorianCalendar**. **Calendar** fornece um conjunto de métodos que permitem que data e hora em milissegundos sejam convertidas em vários componentes úteis. Por exemplo, você pode obter ano, mês, dia, hora, minuto e segundo. **GregorianCalendar** é uma subclasse de **Calendar** que dá suporte ao calendário gregoriano que conhecemos.

**Calendar** implementa as interfaces **Serializable**, **Comparable** e **Cloneable**. Ela não fornece construtores públicos e define as constantes **static int** a seguir:

|                      |             |               |
|----------------------|-------------|---------------|
| ALL_STYLES           | FRIDAY      | PM            |
| AM                   | HOUR        | SATURDAY      |
| AM_PM                | HOUR_OF_DAY | SECOND        |
| APRIL                | JANUARY     | SEPTEMBER     |
| AUGUST               | JULY        | SHORT         |
| DATE                 | JUNE        | SUNDAY        |
| DAY_OF_MONTH         | LONG        | THURSDAY      |
| DAY_OF_WEEK          | MARCH       | TUESDAY       |
| DAY_OF_WEEK_IN_MONTH | MAY         | UNDECIMBER    |
| DAY_OF_YEAR          | MILLISECOND | WEDNESDAY     |
| DECEMBER             | MINUTE      | WEEK_OF_MONTH |
| DST_OFFSET           | MONDAY      | WEEK_OF_YEAR  |
| ERA                  | MONTH       | YEAR          |
| FEBRUARY             | NOVEMBER    | ZONE-OFFSET   |
| FIELD_COUNT          | OCTOBER     |               |

Como você verá, várias delas são úteis na definição de ou acesso a um componente de data ou hora, como a hora, o minuto ou o ano.

Embora **Calendar** seja abstrata, você pode obter uma referência a um objeto **Calendar** chamando o método **getInstance()**. Ele tem várias formas. A mais simples é:

```
static Calendar getInstance()
```

Ela retorna uma referência a um objeto **Calendar** contendo a data e a hora atuais do sistema. A localidade e o fuso horário padrão são usados. Outras formas permitem a especificação de uma localidade e de um fuso horário. Um ponto importante que é preciso entender é que um objeto **Calendar** representa uma instância específica no tempo.

**Calendar** define muitos métodos. Um particularmente interessante é **get()**, que obtém o valor de um componente de hora ou data, como hora, ano, mês ou dia da semana. Ele é mostrado aqui:

```
int get(int componente)
```

Esse método retorna o valor do componente de hora ou data indicado por *componente*. O valor passado para *componente* deve ser uma das constantes inteiras que acabamos de mostrar. Por exemplo, para obter o ano, passe **Calendar.YEAR**; para obter o minuto, passe **Calendar.MINUTE**; e assim por diante.

Outro método que pode ser muito útil é **getDisplayName( )**, mostrado a seguir:

```
String getDisplayName(int componente, int estilo, Locale loc)
```

Aqui, *componente* especifica o componente de data ou hora a ser obtido. Ele deve ser um dos valores mostrados anteriormente definidos por **Calendar**. O parâmetro *estilo* deve ser **Calendar.SHORT** ou **Calendar.LONG**. Alguns nomes, como o mês ou o dia da semana, têm uma forma longa e uma curta. Esse parâmetro nos permite selecionar qual deve ser obtida. A localidade que determina a representação específica do nome é passada em *loc*. Nem todos os componentes de data e hora têm representações na forma de string. Se nenhuma representação em string estiver disponível, **null** será retornado.

**GregorianCalendar** é uma implementação de **Calendar** que dá suporte ao calendário gregoriano padrão normalmente usado. Essa classe adiciona dois campos **static int**: **AD** e **BC**. Eles representam as duas eras definidas pelo calendário gregoriano.

Há vários construtores para **GregorianCalendar**. O construtor padrão inicializa o objeto com a data e a hora atuais na localidade e o fuso horário padrão. Outros construtores permitem a construção de um **GregorianCalendar** para uma data e uma hora específicas. Veja três exemplos:

```
GregorianCalendar(int ano, int mês, int dia)
```

```
GregorianCalendar(int ano, int mês, int dia, int horas,
 int minutos)
```

```
GregorianCalendar(int ano, int mês, int dia, int horas,
 int minutos, int segundos)
```

Todas as três versões definem dia, mês e ano. Aqui, *ano* especifica o ano. O mês é especificado por *mês*, com zero indicando janeiro. O dia do mês é especificado por *dia*. O primeiro dia do mês é 1. O primeiro construtor configura a hora com meia-noite. A segunda versão configura as horas e os minutos. A terceira adiciona segundos.

Para obter os componentes individuais de data e hora de um **GregorianCalendar**, você pode usar o método **get( )**, herdado de **Calendar**. É fácil obter a maioria dos componentes de data e hora. No entanto, na obtenção da hora, você pode usar um formato de relógio de 12 ou de 24 horas. Se especificar a constante **Calendar.HOUR**, um relógio de 12 horas será usado. A especificação de **Calendar.HOUR\_OF\_DAY** usa um relógio de 24 horas. Em um relógio de 12 horas, você pode determinar se está no período AM ou PM obtendo o valor **Calendar.AM\_PM**. Se o valor for igual a **Calendar.AM**, o período do dia é AM; caso contrário, é PM.

O programa a seguir demonstra **GregorianCalendar**. Observe que ele formata manualmente a data e a hora. Posteriormente neste capítulo, quando **Formatter** for descrita, você verá uma maneira melhor e mais fácil de formatar data e hora.

```

// Demonstra GregorianCalendar
import java.util.*;

class CalendarDemo {
 public static void main(String[] args) {

 // Cria um calendário inicializado
 // com a data e a hora atuais.
 GregorianCalendar calendar = new GregorianCalendar(); ← Obtém a data e a hora atuais.

 // Exibe informações de data e hora atuais.
 System.out.print("Date: ");
 System.out.print(calendar.getDisplayName(Calendar.DAY_OF_WEEK, —
 Calendar.LONG,
 Locale.getDefault())); — Exibe a data.

 System.out.print(" " + calendar.getDisplayName(Calendar.MONTH,
 Calendar.LONG,
 Locale.getDefault())); — Exibe a data.

 System.out.print(" " + calendar.get(Calendar.DATE) + ", ");
 System.out.println(calendar.get(Calendar.YEAR)); — Exibe a data.

 System.out.print("Time: ");
 System.out.print(calendar.get(Calendar.HOUR) + ":"); — Exibe a hora.
 System.out.print(calendar.get(Calendar.MINUTE) + ":"); — Exibe a hora.
 System.out.print(calendar.get(Calendar.SECOND)); — Exibe a hora.

 // determina se é AM ou PM
 int am = calendar.get(Calendar.AM_PM); —
 if(am == Calendar.AM)
 System.out.println(" AM"); — Exibe AM ou PM.
 else
 System.out.println(" PM"); — Exibe AM ou PM.

 // Define as informações de hora e as exibe.
 calendar.set(Calendar.HOUR, 10); —
 calendar.set(Calendar.AM_PM, Calendar.PM); — Define a hora.
 calendar.set(Calendar.MINUTE, 29);
 calendar.set(Calendar.SECOND, 22); — Define a hora.

 System.out.print("Updated time: ");
 System.out.print(calendar.get(Calendar.HOUR) + ":");
 System.out.print(calendar.get(Calendar.MINUTE) + ":");
 System.out.print(calendar.get(Calendar.SECOND)); — Define a hora.

 am = calendar.get(Calendar.AM_PM);
 if(am == Calendar.AM)
 System.out.println(" AM");
 else
 System.out.println(" PM");
 }
}

```

Um exemplo da saída é mostrado aqui:

```
| Date: Monday July 11, 2011
| Time: 10:53:53 AM
| Updated time: 10:29:22 PM
```

Além dos métodos que a classe **GregorianCalendar** herda de **Calendar**, ela também fornece alguns métodos próprios. Talvez o mais interessante seja **isLeapYear()**, que verifica se o ano é bissexto. Sua forma é:

```
boolean isLeapYear(int ano)
```

Esse método retorna **true** quando *ano* é um ano bissexto; caso contrário, retorna **false**. O próximo programa mostra **isLeapYear()** em ação. Ele determina se o ano atual é um ano bissexto.

```
// Demonstra isLeapYear().

import java.util.*;

class LeapYearDemo {
 public static void main(String[] args) {

 // obtém a data e a hora atuais do sistema
 GregorianCalendar calendar = new GregorianCalendar();

 // obtém o ano atual
 int year = calendar.get(Calendar.YEAR);

 System.out.print(year);

 // verifica se o ano atual é bissexto
 if(calendar.isLeapYear(year)) { ← Verifica se o ano é bissexto.
 System.out.println(" is a leap year");
 }
 else {
 System.out.println(" is not a leap year");
 }
 }
}
```

Um exemplo da saída seria:

```
| 2011 is not a leap year
```

**Calendar** (e, portanto, **GregorianCalendar**) dá suporte a muitos outros recursos. Antes de sairmos do tópico de data e hora, seria útil mencionar alguns. Você pode definir um componente de hora ou data usando esta versão de **set()**:

```
void set(int componente, int novoValor)
```

A constante que indica o componente de hora ou data a ser definido é passada em *componente*. O novo valor é passado em *novoValor*. Por exemplo, dado um objeto **Calendar** chamado **myCalendar**, a linha a seguir define o ano como sendo 2025:

```
| myCalendar.set(Calendar.YEAR, 2025);
```

Para determinar se a data e a hora de um objeto **Calendar** são anteriores às de outro objeto **Calendar**, use o método **before()**. Para saber se são posteriores, use o método **after()**. Eles são mostrados aqui:

```
boolean after(Object objCalendar)
boolean before(Object objCalendar)
```

Por exemplo, dadas as linhas

```
GregorianCalendar myCalendarA = new GregorianCalendar(2025, 0, 1);
GregorianCalendar myCalendarB = new GregorianCalendar(2025, 0, 2);
então
```

```
| myCalendarA.before(myCalendarB)
```

é verdadeira, mas

```
| myCalendarA.after(myCalendarB)
```

é falsa.

Você pode somar um valor a um componente de hora ou data usando o método **add()**, mostrado abaixo:

```
void add(int componente, int valor)
```

Ele soma *valor* ao componente de hora ou data especificado por *componente*. Para subtrair, adicione um valor negativo. O método **add()** é muito útil em casos em que a soma de um valor pode fazer outro componente de hora ou data mudar. Por exemplo, a linha a seguir cria um objeto **Calendar** com a data 29 de janeiro de 2025:

```
| GregorianCalendar myCalendar = new GregorianCalendar(2025, 0, 29);
```

Após esta chamada a **add()**,

```
| myCalendar.add(Calendar.DAY_OF_MONTH, 3);
```

**myCalendar** conterá a data 1 de fevereiro de 2025. Isso ocorre porque a soma de 3 dias a 29 de janeiro faz o mês passar para fevereiro.

### Pergunte ao especialista

**P** Na discussão anterior, você mencionou fusos horários. Há classes que encapsulem fusos horários?

**R** Sim. Duas outras classes relacionadas às horas fornecidas por **java.util** são **TimeZone** e **SimpleTimeZone**. Essas classes nos permitem trabalhar com diferenças de fuso horário baseadas na Hora Universal Coordenada (UTC, Coordinated Universal Time), que normalmente também é chamada de Hora de Greenwich (GMT, Greenwich Mean Time). **Time Zone** é uma classe abstrata que define a funcionalidade básica. **SimpleTimeZone** é uma subclasse concreta de **TimeZone**.

## Verificação do progresso

1. Para instanciar um objeto **Calendar**, você deve usar \_\_\_\_\_.
2. E para instanciar um objeto **GregorianCalendar**, usamos \_\_\_\_\_.
3. Um objeto **Calendar** é imutável. Verdadeiro ou falso?

## FORMATANDO A SAÍDA COM Formatter

Como vários dos exemplos dos capítulos anteriores mostraram, o formato padrão usado por Java para exibir valores numéricos nem sempre é ideal. Por exemplo, esta instrução:

```
| System.out.println(10.0/3.0) ;
```

exibe a saída a seguir:

```
| 3.33333333333335
```

Podemos ter casos em que seria interessante ter esse nível de precisão, mas isso não ocorre com frequência. Podemos querer exibir apenas duas casas decimais, por exemplo. Geralmente, é preferível especificar o formato exato em que os dados serão exibidos em vez de usar o padrão fornecido por Java.

Uma maneira de obter saída formatada é com a classe **Formatter**. Ela permite a exibição de números, strings, datas e horas em praticamente qualquer formato desejado. Embora **Formatter** dê suporte a um rico conjunto de recursos, é muito fácil usá-la. **Formatter** já é importante por si só, mas oferece uma segunda vantagem. O mesmo mecanismo de formatação usado por **Formatter** também é usado por algumas outras partes da biblioteca Java, inclusive o método **printf( )** fornecido por **PrintStream**, que será descrito neste capítulo.

### Os construtores de Formatter

Antes de poder usar **Formatter** para formatar saídas, você deve criar uma instância dessa classe. Em geral, **Formatter** funciona convertendo a forma binária dos dados usada por um programa em texto formatado legível por humanos. Em seguida, ela envia esse texto formatado para um destino, que pode ser um buffer na memória, um arquivo ou algum outro tipo de fluxo. Se você usar um buffer para a saída, poderá deixar **Formatter** fornecê-lo automaticamente.

A classe **Formatter** define muitos construtores. Esta é a forma que usaremos:

```
Formatter()
```

---

Respostas:

1. o método estático **getInstance( )** da classe **Calendar**
2. um dos construtores da classe **GregorianCalendar**
3. Falso. Podemos usar os métodos **set( )** e **add( )** para alterar o momento no tempo que ele representa.

Esse é o construtor padrão. Ele usa um **StringBuilder** para armazenar a saída formatada e usa a localidade padrão. Lembre-se, **StringBuilder** é semelhante a **String**, exceto por seu conteúdo poder ser modificado.

Outros construtores nos permitem controlar o destino da saída formatada e a localidade que será usada. Por exemplo:

Formatter(Locale *loc*)

Esse construtor usa um **StringBuilder** para a saída, mas permite a especificação da localidade. Como explicado anteriormente, a localidade determina a forma exata de vários itens, como o ponto decimal ou o símbolo da moeda. O próximo construtor direciona a saída formatada para um arquivo:

Formatter(String *nomearquivo*)  
throws FileNotFoundException

Aqui, *nomearquivo* especifica o nome de um arquivo que receberá a saída formatada. Você pode direcionar a saída para qualquer fluxo de saída usando este construtor:

Formatter(OutputStream *fluxoSaida*)

O fluxo que receberá o texto formatado é especificado por *fluxoSaida*.

Os dois construtores a seguir permitem o envio da saída para qualquer objeto que implemente a interface **Appendable**, que fica no pacote **java.lang** e é implementada por várias classes, inclusive **StringBuilder**, **java.io.PrintStream** e **java.io.Writer**.

Formatter(Appendable *buffer*)  
Formatter(Appendable *buffer*, Locale *loc*)

Aqui, *buffer* especifica o destino da saída, e *loc*, a localidade. Além dos construtores que acabamos de mostrar, **Formatter** fornece muitos outros. Você pode examiná-los por conta própria.

**Formatter** implementa as interfaces **Closeable** e **Flushable** definidas em **java.io** e a interface **Autocloseable** definida em **java.lang**. Os métodos definidos pela classe estão em um resumo na Tabela 24-1. As seções a seguir descrevem o uso de **Formatter**.

## Aspectos básicos da formatação

Após criar um **Formatter**, você poderá usá-lo para gerar uma saída formatada. Para fazê-lo, empregará o método **format()**. A versão que usaremos é:

Formatter format(String *stringFmt*, Object ... *args*)

O parâmetro *stringFmt* compreende dois tipos de itens. O primeiro tipo é composto por caracteres que são simplesmente copiados para o buffer de saída. O segundo tipo contém *especificadores de formato* que definem a maneira como os argumentos subsequentes serão exibidos.

**Tabela 24-1** Métodos definidos por Formatter

| Método                                                                                  | Descrição                                                                                                                                                                                                  |
|-----------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void close()                                                                            | Fecha o <b>Formatter</b> chamador. Isso faz os recursos usados pelo objeto serem liberados. Após um <b>Formatter</b> ser fechado, ele não pode ser reutilizado.                                            |
| void flush()                                                                            | Aplicável principalmente a objetos <b>Formatter</b> vinculados a um fluxo. Faz a saída contida no buffer ser gravada no fluxo.                                                                             |
| Formatter format(String <i>stringFmt</i> , Object ... <i>args</i> )                     | Formata os argumentos passados via <i>args</i> usando os especificadores de formato contidos em <i>stringFmt</i> . Retorna o objeto chamador.                                                              |
| Formatter format(Locale <i>loc</i> , String <i>stringFmt</i> , Object ... <i>args</i> ) | Formata os argumentos passados via <i>args</i> usando os especificadores de formato contidos em <i>stringFmt</i> . A localidade especificada por <i>loc</i> é usada no formato. Retorna o objeto chamador. |
| IOException ioException()                                                               | Se o objeto subjacente que for o destino da saída lançar uma <b>IOException</b> , essa exceção será retornada. Caso contrário, <b>null</b> será retornado.                                                 |
| Locale locale()                                                                         | Retorna a localidade do objeto chamador.                                                                                                                                                                   |
| Appendable out()                                                                        | Retorna uma referência ao objeto subjacente que é o destino da saída.                                                                                                                                      |
| String toString()                                                                       | Para buffers baseados em <b>StringBuilder</b> , esse método retorna um <b>String</b> contendo a saída formatada. Para buffers de outros tipos, a saída formatada pode ou não ser retornada.                |

Em sua forma mais simples, um especificador de formato começa com um símbolo de porcentagem seguido pelo *especificador de conversão de formato*, ou, abreviando, *especificador de formato*. Todos os especificadores de formato são compostos por um único caractere. Por exemplo, o especificador de formato para dados de ponto flutuante é **%f**. Em geral, deve haver um número de argumentos correspondente aos especificadores de formato, e a associação entre eles ocorre da esquerda para a direita. Portanto, considere este fragmento:

```
| Formatter fmt = new Formatter();
| fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);
```

Essa sequência cria um **Formatter** contendo o string a seguir:

```
| Formatting with Java is easy 10 98.600000
```

Nesse exemplo, os especificadores de formato **%s**, **%d** e **%f** são substituídos pelos argumentos que vêm após o string de formato. Logo, **%s** é substituído por "with Java", **%d** é substituído por 10 e **%f** é substituído por 98.6. Todos os outros caracteres são usados na forma em que se encontram. Conforme esperado, o especificador de formato **%s** especifica um string e **%d** especifica um valor inteiro. O especificador **%f** especifica um valor de ponto flutuante, como já mencionado.

**Tabela 24-2** Os especificadores de formato

| Especificador de formato | Formato                           |
|--------------------------|-----------------------------------|
| %a                       | Hexadecimal de ponto flutuante    |
| %A                       |                                   |
| %b                       | Booleano                          |
| %B                       |                                   |
| %c                       | Caractere                         |
| %C                       |                                   |
| %d                       | Inteiro decimal                   |
| %h                       | Código de hash do argumento       |
| %H                       |                                   |
| %e                       | Notação científica                |
| %E                       |                                   |
| %f                       | Ponto flutuante decimal           |
| %g                       | Usa notação decimal ou científica |
| %G                       |                                   |
| %o                       | Inteiro octal                     |
| %n                       | Insere um caractere de nova linha |
| %s                       | String                            |
| %S                       |                                   |
| %t                       | Data e hora                       |
| %T                       |                                   |
| %x                       | Hexadecimal inteiro               |
| %X                       |                                   |
| %%                       | Insere um símbolo de porcentagem  |

O método **format()** aceita uma grande variedade de especificadores de formato, que são mostrados na Tabela 24-2. Observe que muitos especificadores têm uma forma maiúscula e uma minúscula. Quando um especificador maiúsculo é usado, são exibidas letras maiúsculas. Caso contrário, os especificadores de maiúsculas e minúsculas produzem o mesmo formato. É importante saber que Java verifica o tipo de cada especificador de formato em relação ao argumento correspondente. Se o argumento não coincidir, uma **IllegalFormatException** será lançada.

Para **Formatters** que usam um **StringBuilder** como buffer, uma vez que você tenha gerado a saída formatada, poderá obtê-la chamando **toString()**. Por exemplo, continuando com o caso anterior, a instrução a seguir obtém o string formatado contido em **fmt**:

```
|String str = fmt.toString();
```

É claro que, se você quiser apenas exibir o string formatado, não há razão para atribuí-lo primeiro a um objeto **String**. Quando um objeto **Formatter** é passado para **println()**, por exemplo, seu método **toString()** é chamado automaticamente.

Quando não precisar mais de um **Formatter**, você deve fechá-lo chamando **close()**. Isso é especialmente importante para **Formatters** vinculados a arquivos. O fechamento do **Formatter** também fecha o arquivo, liberando assim os recursos usados por ele.

Vejamos um programa curto que reúne tudo o que vimos, mostrando como criar e exibir um string formatado:

```
// Um exemplo muito simples que usa Formatter.
import java.util.*;

class FormatDemo {
 public static void main(String[] args) {
 Formatter fmt = new Formatter(); Cria a saída formatada.

 fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6); ←

 System.out.println(fmt); ← Exibe a saída formatada.
 fmt.close();
 }
}
```

Mais uma coisa: você pode obter uma referência ao objeto no qual a saída formatada será gravada chamando o método `out()`. Ele retorna uma referência a um objeto **Appendable**. Como explicado, por padrão trata-se de um **StringBuilder**. No entanto, você pode usar qualquer objeto **Appendable** como destino da saída. Mas é bom ressaltar que alguns destinos não preservam a saída em buffer. Por exemplo, se você usar um arquivo, a saída será gravada nele. Você não poderá obtê-la usando, digamos, `toString()`.

Agora que você conhece o mecanismo geral usado para formatar saídas, o resto desta seção discutirá vários formatos mostrando muitos exemplos. Também descreverá algumas opções, como a justificação, a largura mínima do campo e a precisão.

## Formatando strings e caracteres

Para formatar um caractere individual, use `%c`. Isso fará o argumento de caractere correspondente ser exibido, sem alteração. Para formatar um string, use `%s`.

## Formatando números

Para formatar um inteiro no formato decimal, você deve usar `%d`. Para formatar um valor de ponto flutuante no formato decimal, use `%f`. E para formatar um valor de ponto flutuante em notação científica, use `%e`.

O especificador de formato `%g` faz **Formatter** usar a notação decimal ou a notação científica se o número de dígitos significativos estiver excedendo a precisão, que é 6 por padrão. O programa a seguir demonstra o efeito do especificador `%g`:

```
// Demonstra o especificador de formato %g.
import java.util.*;

class GFormatDemo {
 public static void main(String[] args) {
 Formatter fmt = new Formatter();

 for(double i=1000; i < 1.0e+10; i *= 100) {
 fmt.format("%g ", i);
 System.out.println(fmt);
 }
 fmt.close();
 }
}
```

Ele produz a saída abaixo:

```
1000.00
1000.00 100000
1000.00 100000 1.00000e+07
1000.00 100000 1.00000e+07 1.00000e+09
```

Você pode exibir inteiros em formato octal ou hexadecimal usando **%o** e **%x**, respectivamente. Por exemplo, este fragmento

```
| fmt.format("Hex: %x, Octal: %o", 196, 196);
```

produz esta saída:

```
| Hex: c4, Octal: 304
```

## Formatando data e hora

Um dos especificadores de conversão mais poderosos é **%t**. Ele permite a formatação de informações de data e hora. O especificador **%t** funciona de maneira um pouco diferente dos outros porque requer o uso de um sufixo para a descrição do componente e do formato exato desejado para a hora ou a data. Os sufixos são mostrados na Tabela 24-3. Por exemplo, para exibir minutos, você usaria **%tM**, em que **M** indica minutos em um campo de dois caracteres. O argumento correspondente ao especificador **%t** deve ser de tipo **Calendar**, **GregorianCalendar**, **Date**, **Long** ou **long**.

Vejamos um programa que demonstra vários formatos de data e hora. Observe que ele usa a classe **GregorianCalendar** para obter a data e a hora atuais. Observe também que é muito mais fácil formatar data e hora usando **Formatter** do que formatá-las manualmente, como fizemos no exemplo de **GregorianCalendar** já mostrado.

```
// Formatando data e hora.

import java.util.*;

class TimeDateFormat {
 public static void main(String[] args) {
 Formatter fmt = new Formatter();
 GregorianCalendar calendar = new GregorianCalendar();

 // formato padrão de 12 horas
 fmt.format("%tr\n", calendar);

 // informações completas de data e hora
 fmt.format("%tc\n", calendar);

 // somente hora e minuto
 fmt.format("%tl:%tM\n", calendar, calendar);

 // mês por nome e número
 fmt.format("%tB %tb %tm", calendar, calendar, calendar);
```

**Tabela 24-3** Sufixos do formato de data e hora

| Sufixo | Substituído por                                                                    |
|--------|------------------------------------------------------------------------------------|
| a      | Nome do dia da semana abreviado                                                    |
| A      | Nome do dia da semana completo                                                     |
| b      | Nome do mês abreviado                                                              |
| B      | Nome do mês completo                                                               |
| c      | String de data e hora formatado como <i>dia mês data hh:mm:ss fuso horário ano</i> |
| C      | Primeiros dois dígitos do ano                                                      |
| d      | Dia do mês como um decimal (01-31)                                                 |
| D      | Mês/dia/ano                                                                        |
| e      | Dia do mês como um decimal (1-31)                                                  |
| F      | Ano-mês-dia                                                                        |
| h      | Nome do mês abreviado                                                              |
| H      | Hora (00 a 23)                                                                     |
| I      | Hora (01 a 12)                                                                     |
| j      | Dia do ano como um decimal (001 a 366)                                             |
| k      | Hora (0 a 23)                                                                      |
| l      | Hora (1 a 12)                                                                      |
| L      | Milissegundo                                                                       |
| m      | Mês como decimal (01 a 13)                                                         |
| M      | Minuto como decimal (00 a 59)                                                      |
| N      | Nanosegundo                                                                        |
| p      | Equivalente de AM ou PM de acordo com a localidade em minúsculas                   |
| Q      | Milissegundos a partir de 1/1/1970                                                 |
| r      | <i>hh:mm:ss</i> (formato de 12 horas)                                              |
| R      | <i>hh:mm</i> (formato de 24 horas)                                                 |
| S      | Segundos                                                                           |
| s      | Segundos a partir de 1/1/1970 UTC                                                  |
| T      | <i>hh:mm:ss</i> (formato de 24 horas)                                              |
| y      | Ano em decimal sem século (00 a 99)                                                |
| Y      | Ano em decimal incluindo o século (0001 a 9999)                                    |
| z      | Diferença do UTC                                                                   |
| Z      | Nome do fuso horário                                                               |

```
// exibe os formatos
System.out.println(fmt);
fmt.close();
}
```

Um exemplo da saída é mostrado abaixo:

```
11:11:27 AM
Mon Jul 11 11:11:27 CDT 2011
11:11
July Jul 07
```

Antes de prosseguir, faça testes com vários formatos de hora e data. Por exemplo, tente exibir as formas longas de mês e dia ou usar milissegundos.

### Os especificadores %n e %%

Os especificadores de formato **%n** e **%%** diferem dos outros porque não afetam um argumento. Em vez disso, são apenas sequências de escape que inserem um caractere na sequência de saída. O especificador **%n** insere uma nova linha, e o **%%** insere um símbolo de porcentagem. É claro que você também pode usar a sequência de escape padrão **\n** para inserir um caractere de nova linha, como fez o exemplo anterior.

### Especificando uma largura de campo mínima

Um inteiro inserido entre o símbolo **%** e o especificador de formato age como um *especificador de largura de campo mínima*. Ele preenche a saída para assegurar que atinja um determinado tamanho mínimo. Por exemplo, **%5f** formata um valor de ponto flutuante em um campo com pelo menos cinco caracteres de largura. Mesmo se um string ou número for maior do que o mínimo, ele será exibido em sua totalidade. O preenchimento padrão é feito com espaços.

O programa a seguir demonstra o especificador de largura de campo mínima aplicando-o às conversões de **%s** e **%f**:

```
// Demonstra um especificador de largura de campo.
import java.util.*;

class FieldWidthDemo {
 public static void main(String[] args) {
 Formatter fmt = new Formatter();

 fmt.format("|%f|\n|%12f|\n",
 10.12345, 10.12345);

 fmt.format("|%s|\n|%10s|",
 "Java", "Java");

 System.out.println(fmt);
 fmt.close();
 }
}
```

Esse programa produz a saída abaixo:

```
|10.123450|
| 10.123450|
|Java|
| Java|
```

A primeira linha exibe o número 10.12345 em seu tamanho padrão. A segunda exibe o valor na largura de 12 caracteres. A terceira e quarta linhas exibem o string "Java" em sua largura padrão e em um campo de 10 caracteres de largura.

O modificador de largura de campo mínima costuma ser usado para produzir tabelas em que as colunas ficam alinhadas. Por exemplo, o próximo programa produz uma tabela de quadrados e cubos dos números entre 1 e 10:

```
// Cria uma tabela de quadrados e cubos.
import java.util.*;

class TableFormatDemo {
 public static void main(String[] args) {
 Formatter fmt = new Formatter();

 for(int i=1; i <= 10; i++) {
 fmt.format("%4d %4d %4d\n", i, i*i, i*i*i);
 }

 System.out.println(fmt);
 fmt.close();
 }
}
```

Sua saída é:

```
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
```

## Especificando precisão

Um *especificador de precisão* pode ser usado com vários especificadores de formato. Ele vem após o especificador de largura de campo mínima (se houver um) e é composto por um inteiro. Seu significado exato depende do especificador de formato ao qual é aplicado. Vejamos alguns exemplos.

Quando o especificador de precisão é aplicado aos especificadores **%f** ou **%e**, ele determina o número de casas decimais exibidas. Por exemplo, **%10.4f** exibe um número de pelo menos dez caracteres de largura com quatro casas decimais. Quando

**%g** é usado, a precisão determina o número de dígitos significativos exibidos. (Pode ocorrer arredondamento.) A precisão padrão é 6.

Quando aplicado a strings, o especificador determina a largura de campo máxima. Por exemplo, **%5.7s** exibe um string com pelo menos cinco e não excedendo sete caracteres. Se o string for maior do que a largura de campo máxima, os caracteres finais serão truncados.

O programa a seguir ilustra o especificador de precisão:

```
// Demonstra o modificador de precisão.
import java.util.*;

class PrecisionDemo {
 public static void main(String[] args) {
 Formatter fmt = new Formatter();

 // Formata 4 casas decimais.
 fmt.format("%.4f\n", 123.1234567);

 // Formata 2 casas decimais em um campo de 16 caracteres
 fmt.format("%16.2e\n", 123.1234567);

 // Exibe no máximo 15 caracteres em um string.
 fmt.format("%.15s\n", "Formatting with Java is now easy.");

 System.out.println(fmt);
 fmt.close();
 }
}
```

Ele produz esta saída:

```
123.1235
 1.23e+02
Formatting with
```

O especificador de precisão também pode ser aplicado aos formatos **%h** e **%b**. Nesses casos, ele especifica uma largura de campo máxima.

## Usando os flags de formatação

**Formatter** reconhece um conjunto de *flags* de formato que nos permitem controlar vários aspectos de uma conversão. Todos os flags de formato são caracteres individuais e vêm após % em uma especificação de formato. Eles podem ser vistos abaixo:

| Flag          | Efeito                                                       |
|---------------|--------------------------------------------------------------|
| -             | Justificação à esquerda                                      |
| #             | Formato de conversão alternativo                             |
| 0             | A saída é preenchida com zeros em vez de com espaços         |
| <i>espaço</i> | A saída numérica positiva é precedida por um espaço          |
| +             | A saída numérica positiva é precedida por um sinal de adição |
| ,             | Os valores numéricos incluem separadores de agrupamento.     |
| (             | Os valores numéricos negativos são inseridos em parênteses   |

Nem todos os flags podem ser aplicados a qualquer caso. Por exemplo, o flag + não pode ser usado com o especificador %s. As seções a seguir mostram vários exemplos de uso dos flags.

### Alinhando a saída

Por padrão, todas as saídas são alinhadas à direita. Isto é, se a largura do campo for maior do que os dados exibidos, eles serão inseridos no lado direito do campo. Você pode forçar o alinhamento da saída à esquerda inserindo um sinal de subtração diretamente após %. Por exemplo, %-10.2f alinha um número de ponto flutuante à esquerda com duas casas decimais em um campo de 10 caracteres. Considere este programa:

```
// Demonstra o alinhamento à esquerda.
import java.util.*;

class LeftJustify {
 public static void main(String[] args) {
 Formatter fmt = new Formatter();

 // Alinha à direita por padrão
 fmt.format("%10.2f\n", 123.123);

 // Agora, alinha à esquerda.
 fmt.format("%-10.2f", 123.123);

 System.out.println(fmt);
 fmt.close();
 }
}
```

Ele produz a saída a seguir:

```
| 123.12 |
|123.12 |
```

Como você pode ver, a primeira linha usa o alinhamento padrão à direita. A segunda é alinhada à esquerda dentro de um campo de 10 caracteres.

### Os flags +, espaço, 0 e (

Para fazer um sinal de adição ser exibido antes de valores numéricos positivos, adicione o flag +. Por exemplo,

```
|fmt.format("%+.2f", 100.25);
```

cria esta saída:

```
+100.25
```

Você pode adicionar um espaço em vez de um sinal de adição antes de valores numéricos positivos usando o flag espaço. Por exemplo,

```
|fmt.format("%.2f\n", -100.25);
|fmt.format(" % .2f", 100.25);
```

cria a saída a seguir:

```
| -100.25
| 100.25
```

Observe que o valor positivo é precedido por um espaço.

Para exibir uma saída numérica negativa entre parênteses, e não com um sinal de subtração precedendo-a, use o flag (. Por exemplo,

```
| fmt.format("%(.2f", -100.25);
```

cria este string:

```
| (100.25)
```

O flag 0 faz a saída ser preenchida com zeros em vez de com espaços. Ele pode ser aplicado a valores numéricos. Por exemplo,

```
| fmt.format("%07.2f\n", 1.23);
```

produz a saída abaixo:

```
| 0001.23
```

Em geral, os flags espaço, +, 0 e ( podem ser usados com os especificadores %d, %f, %e e %g. O flag 0 também pode ser usado com os especificadores %x e %o.

**Nota:** Os flags espaço, + e ( também podem ser usados com %x e %o na formatação de um **BigInteger**, uma classe de inteiros especial que dá suporte a inteiros muito grandes. Ela pertence a **java.math**.

### O flag vírgula

Na exibição de números decimais grandes, pode ser útil adicionar separadores de agrupamento, que em inglês são as vírgulas. Por exemplo, o valor 1234567 pode ser lido mais facilmente se formatado como 1,234,567. Para adicionar especificadores de agrupamento, use o flag vírgula (,). Por exemplo,

```
| fmt.format("%.2f", 4356783497.34);
```

cria este string:

```
| 4,356,783,497.34
```

### O flag #

O caractere # especifica um formato alternativo. Vejamos alguns exemplos. Para %f, # assegura que haja um ponto decimal mesmo se não houver dígitos decimais. Se você usar o especificador de formato %x com um #, o número hexadecimal será exibido com um prefixo 0x. O uso do especificador %o com # faz o número ser exibido com um zero precedendo-o.

## A opção de uso de maiúsculas

Como mencionado anteriormente, vários especificadores de formatos têm versões maiúsculas que fazem a saída formatada usar maiúscula onde apropriado. Veja alguns

exemplos. **%C** gera o caractere correspondente em letra maiúscula. **%S** gera o string correspondente em letras maiúsculas. **%E** e **%G** geram o símbolo *e* em maiúsculas. **%X** faz os dígitos hexadecimais de *a* a *f* serem exibidos em maiúsculas na forma *A* a *F*. Além disso, o prefixo opcional **0x** é exibido como **0X**. **%T** faz todos os itens alfabéticos de data e hora ficarem em maiúsculas.

A sequência a seguir mostra os efeitos de **%X** e **%E**:

```
fmt.format("%#x\n", 250);
fmt.format("%#X\n", 250);
fmt.format("%e\n", 123.1234);
fmt.format("%E", 123.1234);
```

A saída é esta:

```
0xfa
0XFA
1.231234e+02
1.231234E+02
```

## Usando um índice de argumento

**Formatter** tem um recurso muito útil que permite a determinação do argumento ao qual um especificador de formato será aplicado. Como vimos, normalmente os especificadores de formato e os argumentos têm correspondência ordenada, da esquerda para a direita. Isto é, o primeiro especificador de formato é aplicado ao primeiro argumento, o segundo especificador é aplicado ao segundo argumento, e assim por diante. Isso permite que os argumentos e os especificadores de formato se relacionem estritamente da esquerda para a direita.

O índice do argumento vem imediatamente após o símbolo de porcentagem em um especificador de formato. Ele tem a forma a seguir:

*n\$*

em que *n* é o índice do argumento desejado, começando em 1. Por exemplo, considere este exemplo:

```
| fmt.format("%3$d %1$d %2$d", 10, 20, 30);
```

Ele produz o string:

```
| 30 10 20
```

Nesse exemplo, o primeiro especificador de formato é aplicado a 30, o segundo é aplicado a 10 e o terceiro a 20. Logo, os argumentos são usados em uma ordem diferente do que rigorosamente da esquerda para a direita.

Uma vantagem dos índices de argumentos é que eles permitem a reutilização de um argumento sem ser preciso especificá-lo duas vezes. Por exemplo, considere esta linha:

```
| fmt.format("%d in hex is %1$x", 255);
```

Ela produz o string a seguir:

```
| 255 in hex is ff
```

Como você pode ver, o argumento 255 é usado pelos dois especificadores de formato.

Há uma simplificação conveniente chamada *índice relativo* que nos permite reutilizar o argumento afetado pelo especificador de formato anterior. Simplesmente especifique < como índice do argumento. Por exemplo, a chamada a **format( )** a seguir produz os mesmos resultados do último exemplo:

```
| fmt.format("%d in hex is %<x", 255);
```

Os índices relativos são especialmente úteis na criação de formatos de data e hora personalizados. Considere o exemplo abaixo:

```
// Usa índices relativos para simplificar a criação
// de um formato de data e hora personalizado.
import java.util.*;

class RelativeIndexDemo {
 public static void main(String[] args) {
 Formatter fmt = new Formatter();
 GregorianCalendar calendar = new GregorianCalendar();

 fmt.format("Today is day %te of %<tB, %<tY", calendar);
 System.out.println(fmt);
 fmt.close();
 }
}
```

Aqui está um exemplo da saída:

```
| Today is day 11 of July, 2011
```

Graças à indexação relativa, o argumento **calendar** só precisa ser passado uma vez, e não três vezes.

## Formatação para um local diferente

Até agora, todos os exemplos formataram saídas para a localidade padrão. Geralmente é isso que queremos. No entanto, se especificarmos um argumento **Locale** ao criar um **Formatter**, a saída será formatada em relação à localidade. Por exemplo, o programa a seguir mostra a diferença no uso da vírgula e do ponto decimal entre convenções de formatação no inglês dos Estados Unidos e no alemão.

```
// Demonstra uma diferença de formatação baseada na localidade.
import java.util.*;

class LocaleFormat {
 public static void main(String[] args) {
 Formatter usFmt = new Formatter(Locale.US);
 Formatter germanFmt = new Formatter(Locale.GERMAN);
 double n = 1234567.24;
```

```

usFmt.format("English: %,.2f", n);
System.out.println(usFmt);
usFmt.close();

germanFmt.format("German: %,.2f", n);
System.out.println(germanFmt);
germanFmt.close();
}
}

```

A saída é esta:

```

English: 1,234,567.24
German: 1.234.567,24

```

Observe que, em inglês, o ponto decimal é um ponto e o separador de agrupamento é uma vírgula. Isso se inverte na versão alemã.

## Fechando um Formatter

Em geral, devemos fechar um **Formatter** quando não precisamos mais usá-lo. Assim, os recursos que ele estava usando são liberados. Isso é particularmente importante na formatação para um arquivo, mas também pode ser importante em outros casos. A título de ilustração, todos os exemplos anteriores fecharam explicitamente o formattador, até mesmo quando o programa estava terminando. Como esses exemplos mostraram, uma maneira de fechar um **Formatter** é chamando `close()` explicitamente. No entanto, a partir do JDK 7, **Formatter** passou a implementar a interface **AutoCloseable**. Ou seja, passou a dar suporte à nova instrução **try-with-resources**. Com essa abordagem, o **Formatter** é fechado automaticamente quando não é mais necessário.

Por exemplo, veja a primeira demonstração de **Formatter** retrabalhada para usar **try-with-resources**:

```

// Usa o gerenciamento automático de recursos com Formatter.
import java.util.*;

class FormatDemo {
 public static void main(String[] args) {

 try (Formatter fmt = new Formatter()) {
 fmt.format("Formatting %s is easy %d %f", "with Java",
 10, 98.6);
 System.out.println(fmt);
 }
 }
}

```

A saída é a mesma de antes.

## Verificação do progresso

1. Se tivermos duas chamadas a `format()` usando o mesmo **Formatter**, o string gerada pela segunda chamada substituirá o gerado pela primeira. Verdadeiro ou falso?
2. O string "%z" é um especificador de formato válido para uma chamada ao método `format()`. Verdadeiro ou falso?
3. Um **Formatter** deve ser \_\_\_\_\_ ao terminar a formatação de todo o texto.

### TENTE ISTO 24-1 Formatando uma tabela de cidades e populações

`CitiesDemo.java`

Neste projeto, você exibirá dados de uma cidade em uma tabela formatada. Os dados serão compostos por duas informações sobre cada cidade: seu nome e sua população atual.

A formatação será feita por um método que usa **Formatter** para gerar um string contendo uma tabela onde cada linha exibirá os dados de uma cidade. A primeira coluna exibirá o nome da cidade, e a segunda, a população. As colunas serão separadas por barras verticais alinhadas e com espaços em seus dois lados. Também haverá uma coluna de barras verticais ao longo das bordas esquerda e direita da tabela. A coluna de nomes será alinhada à esquerda e terá a largura exata do nome de cidade mais longo. A coluna de populações será alinhada à direita e terá a largura exata do maior valor. Presumiremos o uso de uma fonte de espaçamento uniforme para facilitar o alinhamento.

#### PASSO A PASSO

1. As informações da cidade serão armazenadas em um array de objetos **City**. Para definir a classe **City**, crie um novo arquivo chamado **CitiesDemo.java** e insira o código a seguir:

```
import java.util.*;

class City {
 String name;
 int pop; // população atual

 City(String n, int p) {
```

#### Respostas:

1. Falso. Os dois strings serão anexados.
2. Falso.
3. fechado

```

 name = n;
 pop = p;
 }
}

```

2. Em seguida, comece a inserir a classe **CitiesDemo** como mostrado aqui. Essa é a classe que usaremos para demonstrar a formatação:

```

public class CitiesDemo {

 public static void main(String[] args) {
 City[] cities = {
 new City("Dallas", 1197816),
 new City("Portland", 583776),
 new City("Frostbite Falls", 6424),
 new City("New York", 8175133)
 };

 String result = formatCities(cities);

 System.out.println(result);
 }
}

```

3. Termine a classe **CitiesDemo** adicionando o método **static formatCities()**. Ele é mostrado abaixo:

```

static String formatCities(City[] cities) {
 Formatter formatter = new Formatter();
 String result;
 int col1 = 0, col2 = 0;

 // encontra a largura das colunas
 for(City city : cities) {
 if(city.name.length() > col1)
 col1 = city.name.length();
 if(Integer.toString(city.pop).length() > col2)
 col2 = Integer.toString(city.pop).length();
 }

 // constrói o string de formato
 String frmt = "| %- " + col1 + "s | %" + col2 + "d |\n";
 for(City city : cities) {
 formatter.format(frmt, city.name, city.pop);
 }

 result = formatter.toString();
 formatter.close();
 return result;
}
}

```

Observe que **formatCities()** usa um array de objetos **City** e retorna um string contendo a tabela formatada. Para fazê-lo, primeiro o método tem que encontrar a largura de cada coluna. Ele faz isso percorrendo os nomes e populações e registrando a largura dos maiores valores. Em seguida, usa um **Formatter** para formatar cada linha da tabela.

Preste atenção no string **frmt** que é usado para formatar cada linha. Ele é um pouco complicado porque tem que incluir as larguras das colunas, que são especificadas por **col1** e **col2**. Para facilitar a discussão, suponhamos que a primeira coluna tivesse largura igual a 15 e a segunda igual a 10. Com esses valores, o string atribuído a **frmt** seria:

```
"| %‐15s | %10d |\n"
```

A primeira coluna é alinhada à esquerda, precisa de 15 espaços e especifica um string. A segunda é alinhada à direita, precisa de 10 espaços e especifica um inteiro. Observe as barras verticais no começo, no fim e entre as duas colunas, separadas dos dados por espaços. Elas delineiam as colunas na tabela.

4. Execute o método **main()** na classe **CitiesDemo**. Você verá a saída a seguir:

|                 |         |
|-----------------|---------|
| Dallas          | 1197816 |
| Portland        | 583776  |
| Frostbite Falls | 6424    |
| New York        | 8175133 |

5. Embora agora a tabela esteja formatada apropriadamente, duas melhorias podem ser feitas. A primeira é a inclusão de vírgulas nos números das populações. A segunda é o fechamento da caixa ao redor da tabela. Elas serão deixadas como exercícios.

## A FORMATAÇÃO E O MÉTODO **printf()**

Mesmo não havendo nada tecnicamente errado com o uso de **Formatter** diretamente (como os exemplos anteriores fizeram) na criação de saídas que serão gravadas em **System.out**, há uma alternativa mais conveniente: o método **printf()**. Esse método é definido tanto por **PrintStream** quanto por **PrintWriter**. Ele usa os mesmos especificadores de formato e o mesmo mecanismo da classe **Formatter**, que acabamos de descrever. Estas são suas duas formas para **PrintStream**:

```
PrintStream printf(String stringFmt, Object ... args)
PrintStream printf(Locale localidade, String stringFmt, Object ... args)
```

A primeira versão grava *args* na saída padrão com o formato especificado por *stringFmt* usando a localidade padrão. A segunda forma usa a localidade especificada por *localidade*. As duas retornam uma referência ao **PrintStream** chamador. As versões de **PrintWriter** para **printf()** são semelhantes, exceto por um **PrintWriter** ser retornado.

Já que **System.out** é um **PrintStream**, você pode usar **printf()** para exibir valores formatados no console. Isso facilita a exibição de saída formatada porque

você não precisa construir um **Formatter**, fechando-o quando tiver terminado. Pode simplesmente chamar **printf()**. Por exemplo, o programa a seguir usa **printf()** para exibir valores numéricos em vários formatos:

```
// Demonstra printf().

class PrintfDemo {
 public static void main(String[] args) {
 System.out.println("Here are some numeric values " +
 "in different formats.\n");

 System.out.printf("Various integer formats: ");
 System.out.printf("%d %d %+d %05d\n", 3, -3, 3, 3);

 System.out.println();
 System.out.printf("Default floating-point format: %f\n",
 1234567.123);
 System.out.printf("Floating-point with commas: %,f\n",
 1234567.123);
 System.out.printf("Negative floating-point default: %,f\n",
 -1234567.123);
 System.out.printf("Negative floating-point option: %,(f\n",
 -1234567.123);

 System.out.println();

 System.out.printf("Line up positive and negative values:\n");
 System.out.printf("%,.2f\n%,.2f\n",
 1234567.123, -1234567.123);
 }
}
```

A saída é mostrada abaixo:

```
Here are some numeric values in different formats.

Various integer formats: 3 (3) +3 00003

Default floating-point format: 1234567.123000
Floating-point with commas: 1,234,567.123000
Negative floating-point default: -1,234,567.123000
Negative floating-point option: (1,234,567.123000)

Line up positive and negative values:
 1,234,567.12
 -1,234,567.12
```

Devido à comodidade que **printf()** oferece, normalmente ele é a opção preferida para a exibição de saídas com **System.out**.

**Nota:** **PrintStream** e **PrintWriter** também definem um método chamado **format()**, que funciona como **printf()**. Os dois podem ser usados na exibição de dados formatados.

## A CLASSE Scanner

**Scanner** complementa **Formatter**. Ela lê entradas formatadas e as converte em sua forma binária. Pode ser usada na leitura de entradas a partir da entrada padrão (normalmente o teclado), de um arquivo ou de um string, entre outras. Por exemplo, podemos usar **Scanner** para ler um número a partir do teclado e atribuir seu valor a uma variável. Como veremos, devido à sua eficácia, **Scanner** é muito fácil de usar. A classe **Scanner** foi brevemente introduzida no Capítulo 11. Aqui, ela será examinada com mais detalhes.

### Os construtores de Scanner

**Scanner** define vários construtores que nos permitem criar instâncias de diferentes maneiras. Por exemplo, podemos construir um **Scanner** para fazer leituras em um objeto **String**, **InputStream**, **File** ou qualquer objeto que implemente a interface **Readable** (pertencente ao pacote **java.lang**), entre outros. Estes são os construtores usados no capítulo:

```
Scanner(InputStream origem)
Scanner(String origem)
Scanner(Readable origem)
```

O primeiro construtor cria um **Scanner** que usa o fluxo especificado por *origem* como fonte da entrada. O segundo lê entradas no **String** especificado. O terceiro obtém sua entrada em uma fonte que implementa **Readable**.

Vejamos alguns exemplos. A sequência a seguir cria um **Scanner** que lê o arquivo **Test.txt**:

```
| FileReader fin = new FileReader("Test.txt");
| Scanner src = new Scanner(fin);
```

Esse código funciona porque **FileReader** implementa a interface **Readable**. Logo, a chamada ao construtor passa a ser **Scanner(Readable)**.

A linha abaixo cria um **Scanner** que faz leituras na entrada padrão, ou seja, o teclado:

```
| Scanner conin = new Scanner(System.in);
```

Esse código funciona porque **System.in** é um objeto de tipo **InputStream**. Logo, a chamada ao construtor é mapeada para **Scanner(InputStream)**.

A sequência a seguir cria um **Scanner** que lê um string.

```
| String instr = "10 99.88 scanning is easy.";
| Scanner conin = new Scanner(instr);
```

### Aspectos básicos da varredura

Após criar um **Scanner**, basta usá-lo para ler entradas formatadas. Em geral, um **Scanner** lê *tokens* na fonte subjacente que você especificou quando o **Scanner** foi criado. Quando se trata de **Scanner**, um token é uma parte da entrada delineada por um conjunto de delimitadores, que, por padrão, são espaços em branco. Um token é lido pela sua comparação com uma *expressão regular* específica, que define o formato dos dados. (Uma introdução às expressões regulares pode ser encontrada no

Apêndice B.) Embora a classe **Scanner** permita a definição do tipo específico de expressão que sua próxima operação de entrada comparará, ela inclui muitos padrões predefinidos, vários dos quais comparam os tipos primitivos, como **int** e **double**, e strings arbitrários. Logo, geralmente não precisamos especificar um padrão de comparação. Como as expressões regulares serão introduzidas no Apêndice B, só usaremos padrões predefinidos neste capítulo.

Para usar **Scanner**, siga este procedimento:

1. Determine se um tipo específico de entrada está disponível chamando um dos métodos **hasNextX** de **Scanner**, em que *X* é o tipo de dado desejado.
2. Se a entrada estiver disponível, leia-a chamando um dos métodos **nextX** de **Scanner**.
3. Repita o processo até percorrer toda a entrada.
4. Feche o **Scanner** quando não precisar mais dele.

Como o procedimento sugere, **Scanner** define dois conjuntos de métodos que permitem a leitura de entradas. Os primeiros são os métodos **hasNextX**, cujo resumo é mostrado na Tabela 24-4. Esses métodos determinam se o tipo de entrada especificado está disponível. Por exemplo, uma chamada a **hasNextInt()** só retorna **true** quando o próximo token a ser lido é um inteiro, de onde deduzimos que **hasNext()** retorna **true** quando qualquer tipo de token está disponível. Se os dados desejados estiverem disponíveis, você poderá lê-los chamando um dos métodos **nextX** de **Scanner**. Um resumo é mostrado na Tabela 24-5. Por exemplo, para ler o próximo inteiro, chame **nextInt()**. Logo, **next()** pode ler qualquer tipo de token, retornando-o como um **String**.

A sequência a seguir mostra como ler uma lista de inteiros a partir do teclado:

```
Scanner conin = new Scanner(System.in);
int i;

// Lê uma lista de inteiros.
while(conin.hasNextInt()) {
 i = conin.nextInt();
 // ...
}
```

O laço **while** irá parar quando o próximo token não for um inteiro. Logo, o laço para de ler inteiros quando um não inteiro é encontrado no fluxo de entrada (ou quando não há mais tokens disponíveis).

Se um método **next** não puder encontrar o tipo de dado que está procurando, lançará uma **InputMismatchException**. Uma **NoSuchElementException** será lançada se não houver mais entradas disponíveis. Portanto, é melhor confirmar se o tipo de dado desejado está disponível chamando um método **hasNext** antes de chamar o método **next** correspondente.

Como mencionado, quando não precisarmos mais de um **Scanner**, ele deve ser fechado. No entanto, se um **Scanner** estiver vinculado a **System.in**, fechá-lo também fechará **System.in!** Ou seja, não poderemos usar **System.in** novamente. Como resultado, é comum vermos um **Scanner** vinculado a **System.in** aberto, ou sem ser fechado até o fim do programa. A título de ilustração, os próximos exemplos fecharão o **Scanner** explicitamente no fim do programa.

**Tabela 24-4** Resumo dos métodos hasNext de Scanner

| Método                   | Descrição                                                                                                                   |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| boolean hasNext()        | Retorna <b>true</b> se outro token de qualquer tipo estiver disponível para leitura. Caso contrário, retorna <b>false</b> . |
| boolean hasNextBoolean() | Retorna <b>true</b> se um valor <b>boolean</b> estiver disponível para leitura. Caso contrário, retorna <b>false</b> .      |
| boolean hasNextByte()    | Retorna <b>true</b> se um valor <b>byte</b> estiver disponível para leitura. Caso contrário, retorna <b>false</b> .         |
| boolean hasNextDouble()  | Retorna <b>true</b> se um valor <b>double</b> estiver disponível para leitura. Caso contrário, retorna <b>false</b> .       |
| boolean hasNextFloat()   | Retorna <b>true</b> se um valor <b>float</b> estiver disponível para leitura. Caso contrário, retorna <b>false</b> .        |
| boolean hasNextInt()     | Retorna <b>true</b> se um valor <b>int</b> estiver disponível para leitura. Caso contrário, retorna <b>false</b> .          |
| boolean hasNextLong()    | Retorna <b>true</b> se um valor <b>long</b> estiver disponível para leitura. Caso contrário, retorna <b>false</b> .         |
| boolean hasNextShort()   | Retorna <b>true</b> se um valor <b>short</b> estiver disponível para leitura. Caso contrário, retorna <b>false</b> .        |

**Tabela 24-5** Resumo dos métodos next de Scanner

| Método                | Descrição                                                                   |
|-----------------------|-----------------------------------------------------------------------------|
| String next()         | Retorna o próximo token de qualquer tipo da fonte de entrada.               |
| boolean nextBoolean() | Retorna o próximo token como um valor <b>boolean</b> .                      |
| byte nextByte()       | Retorna o próximo token como um valor <b>byte</b> . A base padrão é usada.  |
| double nextDouble()   | Retorna o próximo token como um valor <b>double</b> .                       |
| float nextFloat()     | Retorna o próximo token como um valor <b>float</b> .                        |
| int nextInt()         | Retorna o próximo token como um valor <b>int</b> . A base padrão é usada.   |
| long nextLong()       | Retorna o próximo token como um valor <b>long</b> . A base padrão é usada.  |
| short nextShort()     | Retorna o próximo token como um valor <b>short</b> . A base padrão é usada. |

## Alguns exemplos com Scanner

**Scanner** torna fácil o que seria uma tarefa tediosa. Para entender por quê, examinemos alguns exemplos. O programa a seguir calcula a média de uma lista de números inseridos no teclado. Primeiro ele cria um **Scanner** vinculado a **System.in**. Em seguida, lê os números, somando-os no processo, até o usuário inserir o string "done". Então ele para de ler a entrada e exibe a média dos números.

```
// Usa Scanner para calcular a média dos valores.
import java.util.*;

class AvgNums {
```

```

public static void main(String[] args) {
 Scanner conin = new Scanner(System.in); ← Cria um Scanner
 que lê de System.in.
 int count = 0;
 double sum = 0.0;

 System.out.println("Enter numbers to average.");

 // Lê e soma os números.
 while(conin.hasNext()) { ← Percorre enquanto houver tokens para ler.
 if(conin.hasNextDouble()) { ← Verifica se o próximo token pode ser lido
 sum += conin.nextDouble(); ← como um double. Se puder, lê o token.
 count++;
 }
 else {
 String str = conin.next();
 if(str.equals("done")) break; ← Caso contrário, procura o string "done".
 else {
 System.out.println("Data format error.");
 return;
 }
 }
 }

 System.out.println("Average is " + sum / count);
 conin.close();
}
}

```

Aqui está um exemplo de execução:

```

Enter numbers to average.
1.2
2
3.4
4
done
Average is 2.65

```

Examinemos detalhadamente como esse programa funciona. Primeiro, observe que ele usa `hasNext()` para determinar se há mais algum token de qualquer tipo no fluxo de entrada. Se houver, ele verifica se o próximo token representa um **double**. Se representar, esse número será lido. Se o próximo token não for um **double**, ele será lido com uma chamada a `next()`, que pode ler qualquer tipo de token. Quando isso ocorre, o programa verifica se o token é o string "done". Se for, ele termina normalmente. Caso contrário, exibe um erro.

Observe que os números são lidos pela chamada a `nextDouble()`. Esse método lê qualquer número que possa ser convertido em um valor **double**, inclusive um valor

inteiro, como 2, e um valor de ponto flutuante, como 3,4. Logo, um número lido por **nextDouble()** não precisa especificar um ponto decimal. Esse mesmo princípio geral se aplica a todos os métodos **next( )**. Eles irão comparar e ler qualquer formato de dado que possa representar o tipo de valor que estiver sendo solicitado.

Uma coisa especialmente interessante em **Scanner** é que a mesma técnica usada para ler em uma fonte pode ser usada para ler em outra. Por exemplo, este é o programa anterior retrabalhado para calcular a média de uma lista de números contidos em um arquivo de texto:

```
// Usa Scanner para calcular a média dos valores de um arquivo.
import java.util.*;
import java.io.*;

class AvgFile {
 public static void main(String[] args)
 throws IOException {

 int count = 0;
 double sum = 0.0;

 // Grava saídas em um arquivo.
 FileWriter fout = new FileWriter("test.txt");
 fout.write("2 3.4 5 6 7.4 9.1 10.5 done");
 fout.close();

 FileReader fin = new FileReader("Test.txt");

 Scanner src = new Scanner(fin); ← Cria um Scanner que lê em um arquivo.

 // Lê e soma os números.
 while(src.hasNext()) {
 if(src.hasNextDouble()) {
 sum += src.nextDouble();
 count++;
 }
 else {
 String str = src.next();
 if(str.equals("done")) break;
 else {
 System.out.println("File format error.");
 return;
 }
 }
 }

 src.close();

 System.out.println("Average is " + sum / count);
 }
}
```

Aqui está a saída:

```
|Average is 6.2
```

Esse programa ilustra outro recurso importante de **Scanner**. Observe que o leitor de arquivo referenciado por **fin** não é fechado diretamente. Em vez disso, ele é fechado automaticamente quando **src** chama **close()**. Quando fechamos um **Scanner**, o objeto **Readable** associado a ele também é fechado (se implementar a interface **Closeable**). Portanto, nesse caso, o arquivo referenciado por **fin** é fechado automaticamente quando **src** é fechado.

Outra coisa: para mantermos esse exemplo e o próximo compactos, as exceções de I/O são lançadas por **main()**. No entanto, normalmente códigos do mundo real tratam eles mesmos as exceções de I/O.

Você pode usar **Scanner** para ler entradas com vários tipos de dados – mesmo se a ordem dos dados for desconhecida. Deve apenas verificar que tipo de dado está disponível antes de lê-lo. Por exemplo, considere este programa:

```
// Usa Scanner para ler vários tipos de dados em um arquivo.
import java.util.*;
import java.io.*;

class ScanMixed {
 public static void main(String[] args)
 throws IOException {

 int i;
 double d;
 boolean b;
 String str;

 // Grava saídas em um arquivo.
 FileWriter fout = new FileWriter("test.txt");
 fout.write("Testing Scanner 10 12.2 one true two false");
 fout.close();

 FileReader fin = new FileReader("Test.txt");

 Scanner src = new Scanner(fin);

 // Lê até o fim.
 while(src.hasNext()) {
 if(src.hasNextInt()) { ←
 i = src.nextInt();
 System.out.println("int: " + i);
 }
 else if(src.hasNextDouble()) { ←
 d = src.nextDouble();
 System.out.println("double: " + d);
 }
 else if(src.hasNextBoolean()) { ←
 b = src.nextBoolean();
 System.out.println("boolean: " + b);
 }
 }
 }
}
```

```
 else {
 str = src.next();
 System.out.println("String: " + str);
 }
 }

 src.close();
}
```

Esta é a saída:

```
String: Testing
String: Scanner
int: 10
double: 12.2
String: one
boolean: true
String: two
boolean: false
```

Ao ler tipos de dados diferentes, como feito no programa anterior, você precisa ter cuidado com a ordem de chamada dos métodos **next**. Por exemplo, se o laço invertesse a ordem das chamadas a **nextInt()** e **nextDouble()**, os dois valores numéricos teriam sido lidos como **doubles**, porque **nextDouble()** procura qualquer string numérico que possa ser representado como um **double**.

A partir do JDK 7, **Scanner** também implementa a interface **AutoCloseable**. Ou seja, pode ser gerenciada por um bloco **try-with-resources**. Quando **try-with-resources** é usado, o scanner é fechado automaticamente quando o bloco termina. Por exemplo, o objeto **src** do programa anterior poderia ser gerenciado assim:

```
try (Scanner src = new Scanner(fin))
{
 // Lê até o fim.
 while(src.hasNext()) {
 if(src.hasNextInt()) {
 i = src.nextInt();
 System.out.println("int: " + i);
 }
 else if(src.hasNextDouble()) {
 d = src.nextDouble();
 System.out.println("double: " + d);
 }
 else if(src.hasNextBoolean()) {
 b = src.nextBoolean();
 System.out.println("boolean: " + b);
 }
 else {
 str = src.next();
 System.out.println("String: " + str);
 }
 }
}
```

Para demonstrar claramente o fechamento de um **Scanner**, os exemplos anteriores não usaram **try-with-resources**. A chamada explícita a **close()** também permite que eles sejam compilados por versões de Java anteriores ao JDK 7. No entanto, a abordagem que usa **try-with-resources** é mais otimizada e pode ajudar a evitar erros. Seu uso é recomendado para códigos novos.

## Mais alguns recursos de Scanner

**Scanner** tem vários outros recursos além dos já discutidos. Alguns serão mencionados aqui para que você tenha uma ideia do que oferecem. Como dito anteriormente, você pode usar um **Scanner** para ler qualquer sequência que coincida com uma expressão regular. Após conhecer as expressões regulares, tente usar as versões de **hasNext()** e **next()** que recebam uma delas como argumento.

Você pode definir os *delimitadores* usados por **Scanner** para determinar onde um token começa e termina. Os delimitadores padrão são os caracteres de espaço em branco e esse é o conjunto de delimitadores que os exemplos anteriores usaram. No entanto, é possível alterar os delimitadores com uma chamada ao método **useDelimiter()**. Você pode passar para ele qualquer expressão regular, mas também pode passar algo tão simples quanto uma vírgula, se quiser usá-la como delimitador.

Outro método particularmente útil em algumas circunstâncias é **findInLine()**, que procura a expressão regular especificada dentro de uma linha de texto. Se uma ocorrência for encontrada, o token (e qualquer caractere não coincidente anterior) será usado e retornado. Trata-se de um método útil se você quiser localizar um padrão específico dentro de uma sequência maior.

Um método que tem relação com **findInLine()** é **findWithinHorizon()**. Esse método tenta encontrar uma ocorrência do padrão especificado dentro de um número de caracteres especificado. Se for bem-sucedido, ele retornará o token coincidente.

Você pode ignorar um padrão usando o método **skip()**. Ele procura uma sequência que coincide com uma expressão regular. Se uma ocorrência for encontrada, **skip()** apenas avançará saltando-a.

### Verificação do progresso

1. Para um **Scanner** cuja entrada é "1234", **next()** retorna "1". Verdadeiro ou falso?
2. Para verificar se a fonte da entrada tem mais algum token, chame o método \_\_\_\_\_ de **Scanner**.
3. Para obter o próximo inteiro, chame o método \_\_\_\_\_ de **Scanner**.

Respostas:

1. Falso. **next()** retorna "1234".
2. **hasNext()**
3. **nextInt()**

## A CLASSE Random

A classe **Random** é um gerador de números pseudoaleatórios. Eles são chamados de *pseudoaleatórios* porque são sequências distribuídas uniformemente e não valores realmente aleatórios. **Random** define os construtores a seguir:

```
Random()
Random(long valorpartida)
```

A primeira versão cria um gerador de números que usa um valor arbitrário, mas geralmente exclusivo, chamado *valor de partida*, para definir seu estado inicial. A segunda forma permite a especificação manual de um valor de partida. A inicialização de um objeto **Random** com um valor de partida define com mais eficácia a sequência que ele produzirá. Se você usar o mesmo valor de partida para inicializar outro objeto **Random**, extraírá a mesma sequência aleatória. Se quiser gerar sequências diferentes, especifique valores de partida diferentes. Os métodos públicos definidos por **Random** estão em um resumo na Tabela 24-6.

Como você pode ver, há sete tipos de números pseudoaleatórios que podem ser extraídos de um objeto **Random**. Os valores verdadeiro/falso estão disponíveis em **nextBoolean()**. Bytes podem ser obtidos com uma chamada a **nextBytes()**. Inteiros podem ser extraídos via método **nextInt()**. Inteiros longos podem ser obtidos com **nextLong()**. Para tipos inteiros, o intervalo de valores é o intervalo do tipo especificado. Os métodos **nextFloat()** e **nextDouble()** retornam valores de tipos **float** e **double**, respectivamente, entre 0,0 e menores que 1,0. Para concluir, **nextGaussian()** retorna um valor **double** centralizado em 0,0 com desvio padrão de 1,0. Essa é a distribuição que produz o que normalmente conhecemos como *curva de sino*.

Vejamos um exemplo que demonstra a sequência produzida por **nextGaussian()**. Ele obtém 100 valores gaussianos aleatórios e calcula a média desses valores. O programa também conta quantos valores se encontram dentro de dois desvios padrão, maiores ou menores, usando incrementos de 0,5 para cada categoria. O resultado é exibido graficamente de forma lateral na tela.

**Tabela 24-6** Métodos públicos definidos por Random

| Método                              | Descrição                                                                                     |
|-------------------------------------|-----------------------------------------------------------------------------------------------|
| boolean nextBoolean()               | Retorna <b>true</b> ou <b>false</b> .                                                         |
| void nextBytes(byte[ ] vals)        | Preenche <i>vals</i> com os valores <b>byte</b> gerados.                                      |
| double nextDouble()                 | Retorna o próximo valor como um <b>double</b> .                                               |
| float nextFloat()                   | Retorna o próximo valor como um <b>float</b> .                                                |
| double nextGaussian()               | Retorna o próximo valor como um <b>double</b> , usando a distribuição gaussiana.              |
| int nextInt()                       | Retorna o próximo valor como um <b>int</b> .                                                  |
| int nextInt(int n)                  | Retorna o próximo valor como um <b>int</b> dentro do intervalo que vai de zero a <i>n</i> -1. |
| long nextLong()                     | Retorna o próximo valor como um <b>long</b> .                                                 |
| void setSeed(long novoValorPartida) | Define o valor de partida com o especificado por <i>novoValorPartida</i> .                    |

```

// Demonstra valores gaussianos aleatórios.
import java.util.Random;

class RandomDemo {
 public static void main(String[] args) {
 Random r = new Random();
 double val;
 double sum = 0;
 int[] bell = new int[9];

 for(int i=0; i<100; i++) {
 val = r.nextGaussian(); ← Gera valores pseudoaleatórios.
 sum += val;
 double t = -2;

 for(int x=0; x < bell.length; x++, t += 0.5)
 if(val < t) {
 bell[x]++;
 break;
 }
 }
 System.out.println("Average of values: " + (sum/100));

 // exibe a curva de sino, lateralmente
 for(int i=0; i < bell.length; i++) {
 for(int x=bell[i]; x>0; x--)
 System.out.print("*");
 System.out.println();
 }
 }
}

```

## USE Observable E Observer

A classe **Observable** é usada para criar subclasses que outras partes do programa possam observar. Logo, ela fornece uma estrutura que dá suporte ao padrão Observer descrito no Capítulo 16. Quando um objeto de uma dessas subclasses passa por uma mudança, as classes que o observam são notificadas. As classes observadoras devem implementar

a interface **Observer**, que define o método **update( )**. Esse método é chamado quando um observador é notificado de uma alteração em um objeto observado.

**Observable** define os métodos mostrados na Tabela 24-7. Um objeto que está sendo observado deve seguir três regras simples. Em primeiro lugar, deve adicionar os observadores à sua lista de observadores para que eles possam ser notificados de uma alteração. Isso é feito com uma chamada a  **addObserver( )**. Em segundo lugar, se o objeto tiver mudado, deve chamar **setChanged( )**. Em terceiro lugar, quando ele estiver pronto para notificar os observadores dessa alteração, deve chamar **notifyObservers( )**. Isso faz o método **update( )** do(s) objeto(s) observador(es) ser(em) chamado(s). Tome cuidado – se o objeto chamar **notifyObservers( )** sem ter chamado anteriormente **setChanged( )**, nenhuma ação ocorrerá. O objeto observado deve chamar tanto **setChanged( )** quanto **notifyObservers( )** antes de **update( )** ser chamado.

Observe que **notifyObservers( )** tem duas formas: uma que recebe um argumento e uma que não recebe. Se você chamar **notifyObservers( )** sem um argumento, esse objeto será passado para o método **update( )** do observador como seu segundo parâmetro. Caso contrário, **null** será passado para **update( )**. Você pode usar o segundo parâmetro para passar qualquer tipo de objeto que seja apropriado para seu aplicativo.

**Tabela 24-7** Métodos definidos por Observable

| Método                                    | Descrição                                                                                                                                                                                                                                                                                                                                                      |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void addObserver(Observer <i>obj</i> )    | Adiciona <i>obj</i> à lista de objetos que estão observando o objeto chamador.                                                                                                                                                                                                                                                                                 |
| protected void clearChanged( )            | Uma chamada a esse método retorna o status do objeto chamador como “inalterado”.                                                                                                                                                                                                                                                                               |
| int countObservers( )                     | Retorna o número de objetos que estão observando o objeto chamador.                                                                                                                                                                                                                                                                                            |
| void deleteObserver(Observer <i>obj</i> ) | Remove <i>obj</i> da lista de objetos que estão observando o objeto chamador.                                                                                                                                                                                                                                                                                  |
| void deleteObservers( )                   | Remove todos os observadores do objeto chamador.                                                                                                                                                                                                                                                                                                               |
| boolean hasChanged( )                     | Retorna <b>true</b> se o estado do objeto chamador tiver sido configurado como “alterado”. Isso só ocorre quando <b>setChanged( )</b> é chamado no objeto chamador e não ocorre uma chamada subsequente a <b>notifyObservers( )</b> ou <b>clearChanged( )</b> . Caso contrário, ele retorna <b>false</b> .                                                     |
| void notifyObservers( )                   | Se o estado do objeto chamador indicar que ele mudou, esse método notificará todos os observadores do objeto chamando seu método <b>update( )</b> (definido por <b>Observer</b> ). Em seguida, ele chamará <b>clearChanged( )</b> para voltar o objeto a um estado “inalterado”. Um valor <b>null</b> é passado como segundo argumento para <b>update( )</b> . |
| void notifyObservers(Object <i>obj</i> )  | Se o estado do objeto chamador indicar que ele mudou, esse método notificará todos os observadores do objeto chamando seu método <b>update( )</b> (definido por <b>Observer</b> ). Em seguida, ele chamará <b>clearChanged( )</b> para voltar o objeto a um estado “inalterado”. O argumento passado para <b>update( )</b> é especificado por <i>obj</i> .     |
| protected void setChanged( )              | Chamado se o objeto chamador tiver mudado.                                                                                                                                                                                                                                                                                                                     |

Para observar um objeto observável, você deve implementar a interface **Observer**. Essa interface só define o método mostrado abaixo:

```
void update(Observable obObservado, Object arg)
```

Aqui, *obObservado* é o objeto que está sendo observado e *arg* é o valor passado por **notifyObservers()**. O método **update()** é chamado automaticamente por **notifyObservers()** quando ocorre uma mudança no objeto observado.

Vejamos um exemplo que demonstra **Observable** e **Observer**. Ele cria duas classes observadoras, chamadas **Watcher1** e **Watcher2**. Ambas implementam a interface **Observer**. Elas exibem o valor do argumento que é passado via **update()**. **Watcher2** também emite um aviso sonoro quando o contador chega a zero. (Isso é feito com o uso do valor 7, que é o valor ASCII/Unicode da “campainha” do computador.) A classe que está sendo monitorada se chama **BeingWatched**. Ela estende **Observable**.

Dentro de **BeingWatched** temos o método **counter()**, que faz uma contagem regressiva a partir de um valor especificado. Ele usa **sleep()** para esperar um quinto de segundo entre as contagens. Sempre que o contador muda, **notifyObservers()** é chamado com a contagem atual sendo passada como argumento. Isso faz o método **update()** do observador ser chamado.

Dentro de **main()**, instâncias de **Watcher1** e **Watcher2**, chamadas **observing1** e **observing2**, são criadas. Em seguida, um objeto **BeingWatched** é construído. Ele é chamado de **observed**. Agora, **observing1** e **observing2** são adicionadas à lista de observadores de **observed**. Ou seja, seus métodos **update()** serão chamados sempre que **counter()** chamar **notifyObservers()**.

```
// Demonstra Observable e Observer.

import java.util.*;

// Esta é a primeira classe observadora.
class Watcher1 implements Observer { ← Implema Observable.
 public void update(Observable obj, Object arg) {
 System.out.println("update() in Watcher1 called, count is " +
 ((Integer)arg).intValue());
 }
}

// Esta é a segunda classe observadora.
class Watcher2 implements Observer { ←
 public void update(Observable obj, Object arg) {

 System.out.println("update() in Watcher2 called, count is " +
 ((Integer)arg).intValue());

 // Soa o alarme quando termina
 if(((Integer)arg).intValue() == 0)
 System.out.print('\7');
 }
}
```

```
// Esta é a classe que está sendo observada.
class BeingWatched extends Observable { ← Estende Observable.

 // Faz apenas a contagem regressiva até zero.
 // Cada contagem representa uma mudança sobre
 // a qual os observadores são notificados.
 void counter(int count) {
 for(; count >= 0; count--) {

 setChanged(); // define o estado como alterado. ← Indica que o
 // estado mudou.

 notifyObservers(count); // notifica observadores ← Notifica todos
 // os observadores.

 try {
 Thread.sleep(200);
 } catch(InterruptedException e) {
 System.out.println("Sleep interrupted");
 }
 }
 }

 class TwoObservers {
 public static void main(String[] args) {

 BeingWatched observed = new BeingWatched(); ← Cria um objeto observado.

 Watcher1 observing1 = new Watcher1(); ← Cria dois observadores.
 Watcher2 observing2 = new Watcher2();

 // adiciona os dois observadores ao objeto observado
 observed.addObserver(observing1); ← Adiciona os objetos observadores para que eles
 observed.addObserver(observing2); ← possam receber notificações de mudança.

 observed.counter(5);
 }
 }
}
```

A saída desse programa é mostrada aqui:

```
update() in Watcher2 called, count is 5
update() in Watcher1 called, count is 5
update() in Watcher2 called, count is 4
update() in Watcher1 called, count is 4
update() in Watcher2 called, count is 3
update() in Watcher1 called, count is 3
update() in Watcher2 called, count is 2
update() in Watcher1 called, count is 2
update() in Watcher2 called, count is 1
update() in Watcher1 called, count is 1
```

```
| update() in Watcher2 called, count is 0
| update() in Watcher1 called, count is 0
```

## AS CLASSES Timer E TimerTask

Um recurso interessante e útil oferecido por `java.util` é a possibilidade de agendarmos uma tarefa para execução em algum momento futuro. As classes que dão suporte a isso são **Timer** e **TimerTask**. Usando essas classes, podemos criar uma thread para ser executada em segundo plano enquanto espera durante um período específico. Quando a hora chega, a tarefa vinculada à thread é executada. Várias opções permitem o agendamento de uma tarefa para execução repetida e em uma data específica. Embora seja possível criar uma tarefa manualmente para ser executada em um momento específico com o uso da classe **Thread**, **Timer** e **TimerTask** simplificam muito esse processo.

**Timer** e **TimerTask** operam juntas. **Timer** é a classe que você usará para agendar uma tarefa para execução. A tarefa agendada deve ser instância de **TimerTask**. Logo, para agendar uma tarefa, primeiro você criará um objeto **TimerTask**, e então o agendará para execução usando uma instância de **Timer**.

**TimerTask** implementa a interface **Runnable**; portanto, pode ser usada na criação de uma thread. Seu construtor é mostrado aqui:

```
protected TimerTask()
```

Observe que o construtor é protegido (**protected**).

A classe **TimerTask** define os três métodos abaixo:

```
abstract void run()
boolean cancel()
long scheduledExecutionTime()
```

O método **run( )**, definido pela interface **Runnable**, contém o código que será executado. Observe que **run( )** é abstrato, ou seja, deve ser sobreposto. Logo, a maneira mais fácil de criar uma tarefa controlada pelo temporizador é estendendo **TimerTask** e sobrepondo **run( )**. O método **cancel( )** encerra a tarefa. Ele só retorna **true** quando não ocorre uma execução da tarefa. A hora agendada para a última execução da tarefa é retornada por **scheduledExecutionTime( )**. Esse valor é fornecido em milissegundos a partir de primeiro de janeiro de 1970.

Uma vez que uma tarefa tiver sido criada, ela será agendada para execução por um objeto de tipo **Timer**. Os construtores de **Timer** são:

```
Timer()
Timer(boolean threadD)
Timer(String nomeT)
Timer(String nomeT, boolean threadD)
```

A primeira versão cria um objeto **Timer** que é executado como uma thread normal. A segunda usa uma thread daemon se *ThreadD* for **true**. Uma thread daemon só será executada enquanto o resto do programa estiver em execução. O terceiro e o quarto construtores permitem a especificação de um nome para a thread de **Timer**.

Para agendar uma tarefa para execução, você pode usar um dos métodos **schedule( )** da **Timer**. Há várias formas. Duas delas são mostradas abaixo:

```
void schedule(TimerTask tarefaT, long espera)
void schedule(TimerTask tarefaT, long espera, long repetir)
```

Aqui, *tarefaT* é agendada para execução após o período passado em *espera* ter decorrido. Quando o segundo construtor é usado, a tarefa é executada repetidamente segundo o intervalo especificado por *repetir*. Tanto *espera* quanto *repetir* são especificados em milissegundos. Outras formas de **schedule( )** permitem a especificação de uma data na qual a tarefa será executada.

Você pode interromper o timer chamando **cancel( )**:

```
void cancel()
```

Todas as tarefas serão encerradas.

Se você criar uma tarefa que não for um daemon, deve chamar **cancel( )** para encerrá-la quando seu programa terminar. Se não o fizer, a tarefa controlada pelo temporizador continuará a ser executada mesmo com o método **main( )** tendo terminado.

O programa a seguir demonstra **Timer** e **TimerTask**. Ele define uma tarefa controlada por um temporizador cujo método **run( )** exibe a mensagem "Timer task executed". Essa tarefa é agendada para execução uma vez a cada meio segundo após uma espera inicial de um segundo.

```
// Demonstra Timer e TimerTask.

import java.util.*;

class MyTimerTask extends TimerTask {
 public void run() {
 System.out.println("Timer task executed.");
 }
}

class TimerTest {
 public static void main(String[] args) {
 MyTimerTask myTask = new MyTimerTask();
 Timer myTimer = new Timer();

 // Define uma espera inicial de 1 segundo
 // e então repete a cada meio segundo.
 myTimer.schedule(myTask, 1000, 500);

 try {
 Thread.sleep(5000);
 } catch (InterruptedException exc) {}

 myTimer.cancel();
 }
}
```

Outro método fornecido por **Timer** que você pode achar interessante é **scheduleAtFixedRate( )**. Ele permite o agendamento de uma tarefa para execuções repetidas que se aproxima muito do uso de uma taxa fixa de execução. O tempo de cada repetição se baseia na primeira execução e não na execução anterior. Logo, as “flutuações” no período são limitadas.

### Verificação do progresso

1. A interface **Observer** e a classe **Observable** fornecem uma estrutura para uso do padrão \_\_\_\_\_.
2. Se você tiver um objeto **Random** e quiser um valor **double** gerado por uma distribuição uniforme no intervalo de 0 a 1, use o método \_\_\_\_\_; e se quiser um valor **double** gerado por uma distribuição gaussiana, deve usar o método \_\_\_\_\_.
3. Se quiser executar uma tarefa em intervalos regulares, crie um objeto da classe \_\_\_\_\_ para executá-la e, então, agende-a para execução em intervalos regulares usando um objeto da classe \_\_\_\_\_.

## CLASSES E INTERFACES UTILITÁRIAS VARIADAS

Abaixo temos uma breve descrição de outras classes de nível superior de **java.util** que não são discutidas em nenhum outro local do livro.

|                        |                                                                                                        |
|------------------------|--------------------------------------------------------------------------------------------------------|
| BitSet                 | Define um conjunto de bits.                                                                            |
| EventListenerProxy     | Estende a classe <b>EventListener</b> para permitir parâmetros adicionais.                             |
| Currency               | Encapsula informações sobre moeda.                                                                     |
| EventObject            | A superclasse de todas as classes de eventos.                                                          |
| FormattableFlags       | Define flags de formatação que são usados com a interface <b>Formattable</b> .                         |
| ListResourceBundle     | Gerencia os recursos como um array.                                                                    |
| Objects                | Vários métodos que operam com objetos.                                                                 |
| PropertyPermission     | Gerencia permissões de propriedades.                                                                   |
| PropertyResourceBundle | Gerencia recursos com o uso de arquivos de propriedades.                                               |
| ResourceBundle         | Encapsula um conjunto de recursos específicos de localidades como pares chave/valor.                   |
| ServiceLoader          | Fornece um meio de encontrar provedores de serviços.                                                   |
| StringTokenizer        | Divide um string em partes individuais chamadas tokens.                                                |
| UUID                   | Encapsula e gerencia identificadores universalmente exclusivos (UUIDs, Universally Unique Identifiers) |

Respostas:

1. Observer
2. **nextDouble()**, **nextGaussian()**
3. **TimerTask**, **Timer**

Com exceção de **Formattable** e **EventListener**, todas as interfaces de **java.util** serão discutidas aqui ou no próximo capítulo. A interface **Formattable** é implementada por classes que fornecem formatação personalizada. **EventListener** é herdada por todos os ouvintes de eventos.

### Pergunte ao especialista

**P** O que os métodos da classe **Objects** fazem?

**R** Todos os métodos definidos nessa classe são **static**. Alguns deles executam ações semelhantes às dos métodos não estáticos da classe **Object**. Por exemplo, **Objects** contém um método **toString()** como o mostrado abaixo:

```
static String toString(Object obj)
```

Esse método chama **obj.toString()** e retorna o resultado se **obj** não for **null**; se **obj** for **null**, ele retorna "null". Portanto, **obj.toString()** e **Objects.toString(obj)** são quase idênticos. A única diferença ocorre quando **obj** é **null**. Nesse caso, **obj.toString()** lança uma **NullPointerException**, enquanto **Objects.toString(obj)** retorna "null".

A classe **Objects** também tem métodos convenientes para a comparação de dois objetos ou para verificarmos se um objeto é **null**. Por exemplo, há um método **requireNonNull()** que tem esta forma geral:

```
public static <T> T requireNonNull(T obj)
```

Esse método retorna **obj** quando **obj** não é **null**. Caso contrário, ele lança uma **NullPointerException**. Portanto, o segmento de código

```
| Object obj = Objects.requireNonNull(x);
executa basicamente a mesma função que este segmento:
```

```
| Object obj;
| if(x != null) obj = x;
| else throw new NullPointerException();
```

## EXERCÍCIOS

- Quais são as principais diferenças entre a classe **javax.swing.Timer** discutida no Capítulo 12 e a classe **java.util.Timer**?
- Qual é a diferença entre o valor retornado por uma chamada ao método **System.currentTimeMillis()** mencionado no último capítulo e o retornado por uma chamada ao método **Calendar.getInstance().getTimeInMillis()** usado em capítulos anteriores?
- Quando um **Formatter** vê um caractere **%** no string de formato, como ele sabe se esse caractere faz parte de um especificador de formato ou se é apenas um caractere comum que deve ser inserido no string de saída?
- Suponhamos que você tivesse um objeto **cal** do tipo **GregorianCalendar** quisesse exibir, com **System.out.printf()**, a data que ele representa na forma

*ano:mês:dia:hora:minuto:segundo* usando a forma longa de ano, mês e dia e o formato de 24 horas. Por exemplo, se `cal` representasse o meio-dia de 11 de março de 1984, a saída seria “1984:março:11:12:00:00”. Quais seriam os argumentos da chamada a `printf()`? Use índices relativos onde for possível.

5. Suponhamos que `x` fosse uma variável `long` a ser exibida com o uso de `System.out.printf()` com largura de campo mínima igual a 20, sinal de adição (+) inicial se for positiva, vírgulas após cada três dígitos a partir da direita e alinhamento à esquerda. Quais seriam os argumentos da chamada a `printf()`?
6. Suponhamos que você tivesse uma variável `double x`, usasse um `Formatter` para criar um string contendo o valor de `x` e depois usasse um `Scanner` para recuperar esse valor a partir do string. É garantida a obtenção do mesmo valor com o qual começou? Por quê?
7. Qual é a saída da instrução a seguir?

```
| System.out.printf("%%%(-11.2E%%", -1.23456789);
```

8. O que aconteceria se você criasse um objeto `GregorianCalendar` que representasse uma data inválida como 32 de janeiro de 1970? Mais precisamente, o que aconteceria se executasse este segmento de código:

```
| GregorianCalendar cal = new GregorianCalendar(1970, 0, 32);
e exibisse mês, dia e ano armazenados em cal usando chamadas a getDisplayName()?
```

9. Crie um programa que exiba os nomes de todos os meses em alemão.
10. Crie um programa que verifique se a data correspondente a 70 dias antes de primeiro de abril de 2000 é anterior ou posterior à data correspondente a 500 dias após primeiro de setembro de 1998. O programa deve exibir as duas datas e dizer qual vem primeiro.
11. Crie um método `static numDaysLeftInYear()` que use uma variável `cal` de tipo `GregorianCalendar` como parâmetro e retorne o número de dias restantes no ano representado por `cal`. Por exemplo, se `cal` representar 28 de dezembro de 2000, `numDaysLeftInYear()` retornará 3 (sem incluir 28 de dezembro como um dos dias restantes no ano). Escreva também um programa para testar seu método.
12. Suponhamos que `x` fosse uma variável inteira já inicializada. `System.out.printf("%d", x)` exibiria o mesmo string que `System.out.print(new java.util.Formatter().format("%d", x))`?
13. Melhore o método `formatCities()` da seção Tente isto 24-1 para que exiba os números com vírgulas separando cada três dígitos a partir da direita.
14. Melhore o método `formatCities()` da seção Tente isto 24-1 para que exiba uma linha de hifens ao longo do topo e da base da tabela. A linha começa na barra vertical esquerda e termina na barra vertical direita.
15. Crie um programa semelhante à classe `TableFormatDemo` discutida no capítulo que, em vez de listar os quadrados e cubos dos números de 1 a 10, liste os números de 1 a 20 na notação hexadecimal, octal e científica. Use indexação relativa onde for possível.

16. Crie um método **weekDay( )** que use um **Calendar** como parâmetro. Ele retorna o nome completo do dia da semana representado pelo **Calendar**.
17. Crie um método **convertToRows( )** que use dois parâmetros: o nome de um arquivo de texto para a entrada e o nome de um arquivo de texto para a saída. O arquivo de entrada deve conter uma lista de inteiros separados por espaços em branco. O método **convertToRows( )** lê os inteiros no arquivo de entrada e os grava no arquivo de saída, quatro inteiros por linha. No arquivo de saída, os inteiros devem ser alinhados à direita em quatro colunas com largura de 15 caracteres. A última linha pode ter menos de quatro inteiros. Crie um programa para testar seu novo método. Use um **Scanner** para ler a entrada e um **Formatter** para gravar a saída.
18. Na amostragem de uma distribuição normal padrão (gaussiana), aproximadamente 68% das amostras ficam entre  $-1$  e  $+1$ , 95% entre  $-2$  e  $+2$  e 99,7% entre  $-3$  e  $+3$ . Crie um programa que gere 1.000 valores aleatórios a partir de uma distribuição gaussiana e registre quantos deles pertencem a cada um desses intervalos. Em seguida, exiba os resultados.
19. Refaça o programa **SavingsAccountDemo** do Capítulo 16 para que use **Observer/Observable** em vez de criar suas próprias classes e interfaces a partir do zero. A classe **SavingsAccount** deve herdar de **Observable** e as classes **FeeHandler** e **FraudHandler** devem implementar **Observer** e não **BalanceChangeHandler**.
20. Crie um programa contendo um **Timer** que comece a contar imediatamente de 1 a 10 em intervalos de meio segundo e, após mais meio segundo, soe um alarme.

---

# Usando as estruturas de dados do Collections Framework

## PRINCIPAIS HABILIDADES E CONCEITOS

- Entender as estruturas de dados básicas
- Conhecer as interfaces de coleções
- Conhecer as classes de coleções
- Usar um iterador
- Conhecer as interfaces de mapas
- Conhecer as classes de mapas
- Usar um comparador
- Ter uma visão geral dos algoritmos
- Conhecer as classes de coleções legadas

Este capítulo continua nossa investigação do pacote **java.util** descrevendo um dos subsistemas mais poderosos de Java: o *Collections Framework*. O Collections Framework é uma hierarquia sofisticada de interfaces e classes que fornece tecnologia de ponta para o gerenciamento de grupos de objetos (coleções). Um conhecimento básico do Collection Frameworks é importante para todos os programadores de Java porque ele é muito usado em aplicativos Java e dentro de outras partes da biblioteca de APIs.

O Collections Framework fornece vários meios para o armazenamento de um conjunto de objetos. Ele faz isso fornecendo implementações genéricas de várias estruturas de dados comuns. Embora você vá estudar estruturas de dados em detalhes em um curso posterior, um conhecimento geral é necessário para o Collections Framework ser usado de maneira eficaz. Logo, começaremos com uma visão geral das estruturas de dados.

***Nota:** Este capítulo só discutirá as partes do Collections Framework definidas em **java.util**. O Java também define coleções de concorrência, projetadas para uso em ambientes de execução altamente paralela. Elas ficam no pacote **java.util.concurrent**.*

## VISÃO GERAL DAS ESTRUTURAS DE DADOS

Antes de discutirmos uma estrutura de dados específica, será útil definir o termo *estrutura de dados*. Aqui está uma definição: uma estrutura de dados determina um meio de organizar dados e especifica as regras pelas quais esses dados podem ser acessados. Logo, uma estrutura de dados determina o modelo, a forma e as características básicas de uma estrutura conceitual usada para gerenciar grupos de dados.

Há muitos tipos de estruturas de dados. O mais simples é o array, que você conheceu na Parte I. Os arrays da linguagem Java são implementados como listas lineares de tamanho fixo. Os elementos individuais são acessados via um índice, com o uso da notação de subscritos de array. Portanto, os arrays dão suporte a um acesso aleatório rápido aos elementos. Como você verá, o Collections Framework também dá suporte a arrays dinâmicos, que podem crescer quando necessário.

Veja mais algumas estruturas de dados comuns:

- pilha
- fila
- lista encadeada
- árvore
- tabela hash

Todas elas têm suporte no Collections Framework. Cada uma tem propriedades, recursos e características de desempenho diferentes. Logo, nem todas as estruturas de dados são apropriadas para qualquer tarefa. A escolha da estrutura de dados depende da aplicação.

### Pilhas e filas

Você já viu casos de uma pilha e de uma fila porque as duas já foram usadas como exemplos neste livro. A título de recapitação, tanto a pilha quanto a fila definem estritamente a ordem em que elementos podem ser adicionados e removidos. No caso de uma pilha, a ordem é: último a entrar, primeiro a sair (LIFO, last-in, first-out). Ou seja, o primeiro elemento inserido na pilha é o último elemento a ser usado. Para visualizar uma pilha, imagine uma pilha de pratos. O primeiro prato na mesa é o último a ser usado e o último prato é o primeiro a ser usado. As pilhas não dão suporte ao acesso aleatório a seus elementos. Em vez disso, dão suporte a duas operações básicas: *inserção* e *extração*. Um item é colocado na pilha com uma operação de inserção. O item de cima é retirado da pilha com uma operação de extração.

Em sua forma padrão, uma fila é, essencialmente, o oposto de uma pilha. Sua ordem é: primeiro a entrar, primeiro a sair (FIFO, first-in, first-out). Ou seja, o primeiro item inserido na fila é o primeiro item recuperado, o segundo item inserido é o segundo recuperado, e assim por diante. A fila do caixa de uma loja é um exemplo de fila do mundo real. A ordem FIFO da fila é imposta pelo fato de itens só serem adicionados ao fim da fila e só serem removidos da frente. Logo, as filas não dão suporte ao acesso aleatório a seus elementos. Com frequência, a frente da fila é chamada de *cabeça* e o fim de *cauda*. Devemos destacar, no entanto, que existem outros tipos de filas em que a ordem dos elementos é determinada por um fator diferente da ordem de

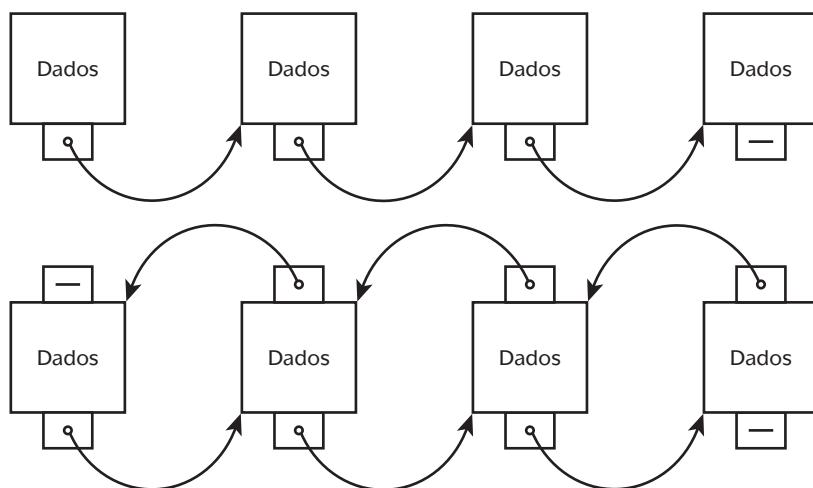
inserção, como a prioridade. Mesmo assim, em todos os casos, o mecanismo geral de inserção e remoção de um elemento em uma fila permanece o mesmo.

## Listas encadeadas

As pilhas e filas compartilham uma característica: as duas têm regras rigorosas que definem a ordem em que os dados podem ser acessados. Uma lista encadeada é diferente porque seus elementos podem ser acessados de maneira mais flexível. Em uma lista encadeada, cada informação carrega com ela um link que conduz ao próximo elemento da lista. Novos elementos podem ser inseridos na lista a qualquer momento com uma simples reorganização dos links. Da mesma forma, um elemento pode ser removido da lista com a reorganização dos links. Como resultado, as listas encadeadas são estruturas de dados dinâmicas que crescem ou diminuem automaticamente à medida que novos elementos são adicionados ou removidos.

As listas encadeadas podem ser simplesmente ou duplamente encadeadas. Cada elemento de uma lista simplesmente encadeada contém um link que conduz ao próximo elemento. Em uma lista duplamente encadeada, cada elemento contém links que conduzem tanto para o elemento seguinte quanto para o anterior. (Consulte a Figura 25-1.) A principal diferença entre as duas é que uma lista simplesmente encadeada só pode ser seguida em uma direção, enquanto uma lista duplamente encadeada pode ser seguida nas duas direções. A implementação de lista encadeada do Collections Framework usa uma lista duplamente encadeada.

Embora as listas encadeadas ofereçam um meio conveniente de armazenamento de listas de tamanho variável, encontrar um elemento pode ser demorado porque uma busca sequencial deve ser usada. Em outras palavras, para procurar um item em uma lista encadeada, você deve começar pelo primeiro item. Se ele não for o que você está procurando, será preciso seguir o link que conduz ao próximo item e verificá-lo. Esse processo se repete até o item ser encontrado. Isso torna as listas encadeadas muito boas para situações em que uma operação será aplicada a uma lista inteira, mas é inefficiente quando o acesso aleatório a um elemento específico é necessário.



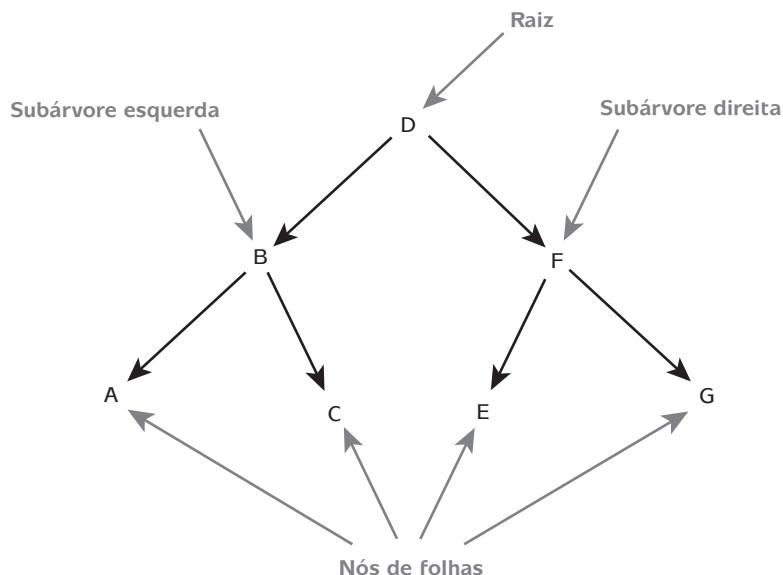
**Figura 25-1** Listas simplesmente e duplamente encadeadas.

## Árvores

Uma das estruturas de dados mais poderosas é a árvore. Como o nome sugere, uma árvore organiza os dados como uma hierarquia. Quando representada visualmente, essa organização tem uma estrutura de árvore. No entanto, na ciência da computação, com frequência as árvores são desenhadas de cabeça para baixo. Isso gera uma terminologia interessante. A *raiz* é o primeiro item da árvore. Cada item é chamado de *nó* da árvore. Qualquer parte da árvore que descenda de um nó é chamada de *subárvore*, com esse nó formando a raiz da subárvore. Um nó que não tenha subárvores é chamado de *nó terminal* ou *folha*. A *altura* da árvore é igual ao número de camadas *descendentes* que seus nós apresentam a partir da raiz. Ao trabalhar com árvores, você pode pensar nelas existindo na memória com a aparência que têm no papel. Mas lembre-se: uma árvore é apenas uma maneira de organizar dados logicamente na memória, e a memória é linear.

Um tipo comum de árvore é a *árvore binária classificada*, também chamada de *árvore binária de busca*. Neste tipo de árvore, cada nó contém um link que conduz a um membro esquerdo e a um membro direito. Para cada nó da árvore, itens com chaves menores são armazenados em um lado e itens com chaves maiores são armazenados no outro lado. A título de simplificação, suponhamos que as chaves menores estivessem à esquerda da raiz e as maiores estivessem à direita. Dada essa organização, para encontrar um nó em uma árvore binária classificada, só teríamos que seguir os links, indo para a esquerda ou para a direita verificando se a chave do nó atual é maior ou menor do que a chave da busca. Essa abordagem gera buscas rápidas porque o número de nós que têm de ser verificados não excede a altura da árvore. A Figura 25-2 mostra uma pequena árvore binária classificada.

É importante destacar que os tempos rápidos de busca de uma árvore binária classificada só ocorrem quando a árvore tem um balanceamento aceitável. Em outras



**Figura 25-2** Árvore binária classificada.

palavras, nenhuma subárvore pode estar em um nível de profundidade maior do que a outra. Na pior das hipóteses, uma árvore desbalanceada pode decair para uma lista linear, o que resultaria em um desempenho reduzido.

## Tabelas hash

Uma tabela hash é uma estrutura de dados que armazena informações em algum tipo de tabela (como um array) em que o local onde os dados estão armazenados é determinado pelo *código hash* associado a esses dados. Em geral, o código hash é calculado com base em algum aspecto que identifique os dados, o qual age como a *chave*. Nas tabelas hash Java, o código hash é calculado a partir de uma chave explicitamente especificada. O cálculo do código hash deve funcionar de tal modo que cada chave produza um valor (razoavelmente) exclusivo. A principal vantagem de uma tabela hash é que tempos de acesso muito velozes podem ser obtidos porque o código hash age como índice dos dados. Portanto, inserções, exclusões e buscas em uma tabela hash podem ser executadas muito rapidamente. Contudo, como regra geral, as tabelas hash não armazenam dados em ordem classificada.

## Selecionando uma estrutura de dados

Você pode usar as características das estruturas de dados básicas para ajudá-lo a selecionar qual é a melhor para a tarefa. Vejamos algumas dicas. Quando precisar de uma lista “último a entrar, primeiro a sair”, use uma pilha. Se precisar de uma lista “primeiro a entrar, primeiro a sair”, use uma fila. Para obtenção de tempo de acesso constante, a tabela hash é uma boa opção se uma ordem classificada não for necessária. A árvore binária classificada é uma excelente escolha quando uma lista classificada é necessária e tempos de busca velozes são prioridade. Uma lista encadeada é apropriada quando a lista for acessada principalmente em ordem sequencial e a classificação não for necessária (ou necessária apenas ocasionalmente). Um array dinâmico oferece acesso aleatório muito rápido a seus elementos. No entanto, inserções e exclusões podem ser caras em termos de tempo.

Agora que você tem um conhecimento básico das principais estruturas de dados, é hora de examinar o Collections Framework.

### Verificação do progresso

1. Uma estrutura de dados é \_\_\_\_\_.
2. Em que ordem uma fila armazena seus elementos?
3. Cite dois tipos de listas encadeadas.
4. Uma árvore binária classificada é uma boa opção quando o acesso rápido aos elementos é necessário?

Respostas:

1. uma maneira de organizar dados e uma especificação das regras pelas quais eles poderão ser acessados
2. Primeiro a entrar, primeiro a sair.
3. Simplesmente encadeada e duplamente encadeada.
4. Sim.

## Pergunte ao especialista

**P** Você mencionou que as listas encadeadas são menos eficientes do que os arrays quando o acesso a um elemento específico é necessário. Isso ocorre porque você pode acessar o elemento diretamente em um array, mas na lista encadeada é preciso percorrê-la começando em uma das extremidades. Dada a velocidade dos computadores modernos, isso importa?

**R** Sim, principalmente para programas que manipulam grandes quantidades de dados. Considere este exemplo: Suponhamos que um programa tivesse que armazenar 100 milhões de itens em uma lista e que o acesso aleatório fosse necessário 4 milhões de vezes ao dia. Suponhamos também que o computador fosse tão rápido a ponto de executar 1 bilhão de acessos ao array ou percorrer 1 bilhão de links da lista por segundo. Em outras palavras, suponhamos que cada uma dessas operações levasse apenas um nanosegundo (um bilionésimo de segundo). Se você armazenar os 100 milhões de itens em um array, o acesso a 4 milhões de itens todo dia levará 4 milhões de nanosegundos, ou apenas cerca de quatro milésimos de segundo. Ou seja, o acesso a 4 milhões de itens consumiria um tempo insignificante do computador a cada dia. Mas digamos que você armazenasse os 100 milhões de itens em uma lista encadeada. Nesse caso, para chegar a qualquer elemento específico a partir da extremidade mais próxima, teria que percorrer em média um quarto da lista. Portanto, aproximadamente 25 milhões de links teriam que ser percorridos para o elemento ser acessado. Já que 4 milhões de itens têm que ser acessados todo dia, a lista será percorrida 4 milhões de vezes, cada uma delas percorrendo uma média de 25 milhões de links. Isso nos dá 100 trilhões de passagens pela lista, o que leva 100 trilhões de nanosegundos, ou mais de 27 horas. Logo, se não há horas suficientes no dia para o computador executar todos os 4 milhões de acessos, imagine para fazer algum outro trabalho!

Como a discussão anterior mostra, a questão da eficiência dos algoritmos e estruturas de dados é um assunto importante que você estudará em detalhes em seu curso de estruturas de dados.

## VISÃO GERAL DAS COLEÇÕES

O Collections Framework padroniza o modo como um programa Java trata grupos de objetos. Ele faz isso fornecendo implementações de várias das estruturas de dados mais usadas, como o array dinâmico, a pilha, a fila, a lista duplamente encadeada, a árvore e a tabela de hash que acabamos de descrever. Raramente você terá que codificar uma dessas estruturas de dados. Resumindo, o Collections Framework oferece soluções “acabadas” para muitas tarefas comuns de programação.

O Collections Framework tem como base um conjunto de interfaces bem definidas. Várias implementações das interfaces são fornecidas. A vantagem dessa abordagem é que, uma vez que você entender a funcionalidade descrita por uma interface, poderá usar qualquer implementação específica. Também poderá usar facilmente uma implementação diferente se os requisitos de seu programa mudarem.

Além das coleções, a estrutura também define várias interfaces e classes de mapas. Os *mapas* armazenam pares chave/valor. Embora façam parte do Collections Framework, não são “coleções” no uso estrito do termo. No entanto, você pode obter um mapa em *view de coleção*. Essa visão contém os elementos do mapa armazenados em uma coleção. Logo, você pode processar o conteúdo de um mapa como uma coleção, se quiser.

Os *algoritmos* são outra parte importante do Collections Framework. Eles operam com coleções e são definidos como métodos **static** dentro da classe **Collections**. Os algoritmos fornecem um meio padronizado de tratar coleções.

Outro item intimamente associado ao Collections Framework é o iterador. Um *iterador* oferece uma maneira padronizada e geral de acessarmos os elementos de uma coleção, um após o outro. Logo, ele fornece um meio de *enumerar o conteúdo de uma coleção*. Como cada coleção fornece um iterador, os elementos de qualquer classe de coleção podem ser acessados com um.

## AS INTERFACES DE COLEÇÕES

O Collections Framework define várias interfaces. Essas interfaces determinam a natureza básica das classes de coleções porque especificam grande parte da funcionalidade fornecida por essas classes. As classes de coleções apenas fornecem implementações diferentes das interfaces padrão. Portanto, para conhecer o Collections Framework, você deve começar conhecendo suas interfaces.

As interfaces de coleções empacotadas em **java.util** estão resumidas na tabela a seguir:

| Interface    | Descrição                                                                                                                   |
|--------------|-----------------------------------------------------------------------------------------------------------------------------|
| Collection   | Permite o trabalho com grupos de objetos; fica no topo da hierarquia de coleções.                                           |
| Deque        | Estende <b>Queue</b> para tratar uma fila de extremidade dupla. Pode ser usada para implementar uma pilha ou uma fila.      |
| List         | Estende <b>Collection</b> para tratar listas de objetos.                                                                    |
| NavigableSet | Estende <b>SortedSet</b> para tratar a recuperação de um elemento com base em quanto seu valor está próximo de outro valor. |
| Queue        | Estende <b>Collection</b> para tratar filas.                                                                                |
| Set          | Estende <b>Collection</b> para tratar conjuntos, que devem conter elementos exclusivos.                                     |
| SortedSet    | Estende <b>Set</b> para tratar conjuntos classificados.                                                                     |

Além dessas interfaces, as coleções também usam as interfaces **Comparator**, **Iterator** e **ListIterator**, que serão descritas com mais detalhes posteriormente neste capítulo. Resumidamente, **Comparator** determina como dois objetos são comparados; **Iterator** e **ListIterator** dão suporte a iteradores. Outra interface usada por algumas coleções é **RandomAccess**. Ao implementar **RandomAccess**, uma coleção indica que dá suporte ao acesso eficiente e aleatório a seus elementos.

Para fornecer maior flexibilidade em seu uso, as interfaces de coleções permitem que alguns métodos sejam opcionais. Os métodos opcionais permitem a modificação do conteúdo de uma coleção. As coleções que dão suporte a esses métodos são chamadas de *modificáveis*. As coleções que não permitem que seu conteúdo seja alterado são chamadas de *não modificáveis*. Todas as coleções internas são modificáveis.

As seções a seguir examinarão as interfaces de coleções.

## A interface Collection

A interface **Collection** é a base na qual o Collections Framework foi construído porque deve ser implementada por qualquer classe que definir uma coleção. **Collection** é uma interface genérica que tem esta declaração:

```
interface Collection<E>
```

E especifica o tipo de objetos que a coleção conterá. **Collection** estende a interface **Iterable**. Isso é importante porque só classes que implementam **Iterable** podem ser usadas com um laço **for** de estilo for-each. **Iterable** também permite que uma coleção seja percorrida com o uso de um iterador.

**Collection** declara os métodos básicos que todas as coleções terão. Esses métodos estão resumidos na Tabela 25-1. Já que todas as coleções implementam **Collection**, é necessário conhecer seus métodos para termos um entendimento claro da estrutura. Examinemos cada um.

Em sua base, uma estrutura de dados dá suporte a duas operações essenciais: a inserção e a retirada de um item em uma coleção. Na interface **Collection**, essas operações são suportadas pelos métodos **add()** e **remove()**. Observe que **add()** recebe um argumento de tipo **E**, que é o parâmetro de tipo de **Collection**. Ou seja, os objetos adicionados a uma coleção devem ser compatíveis com o tipo de dado esperado pela coleção.

Agora, observe os métodos **size()** e **isEmpty()**. São duas operações mais básicas comumente implementadas por uma estrutura de dados. O número de elementos armazenado atualmente na coleção pode ser obtido com uma chamada a **size()**. Você pode usar esse valor, por exemplo, como ponto de partida ao percorrer uma coleção usando um laço. Cuidado, pois o valor retornado por **size()** muda quando o número de elementos armazenado na coleção muda. Quando uma coleção não contém elementos, **isEmpty()** retorna **true**. Você pode usar esse método para evitar acessar uma coleção vazia, por exemplo. Uma coleção pode ser esvaziada com uma chamada a **clear()**.

Embora as operações que acabamos de descrever sejam suficientes para a manipulação de muitos tipos de operações com coleções, **Collection** especifica três métodos que são convenientes no trabalho com grupos de elementos. Eles são **addAll()**, **removeAll()** e **retainAll()**. Você pode adicionar o conteúdo inteiro de uma coleção a outra coleção chamando **addAll()**. Para remover um grupo de objetos, chame **removeAll()**. E para remover todos os elementos exceto os de um grupo especificado, chame **retainAll()**.

Duas coleções podem ser comparadas em relação à igualdade com uma chamada a **equals()**. O significado preciso de “igualdade” pode diferir de uma coleção para outra. Em coleções que implementam as interfaces **List** ou **Set**, seus conteúdos são comparados. Você pode determinar se uma coleção contém um elemento chamando **contains()**. Para determinar se todos os elementos de uma coleção fazem parte de outra coleção, chame **containsAll()**.

O método **iterator()** retorna um iterador para uma coleção. Os iteradores são muito usados no trabalho com coleções porque, como explicado anteriormente, eles fornecem um meio pelo qual os elementos da coleção podem ser enumerados. O laço **for** de estilo for-each, que também percorre uma coleção, oferece uma alternativa conveniente em alguns casos.

Para concluir, os métodos **toArray()** retornam um array contendo os elementos armazenados na coleção. Uma versão retorna um array de tipos **Object**. A

**Tabela 25-1** Métodos declarados por Collection

| Método                                              | Descrição                                                                                                                                                                                                                                                                                                                                                                   |
|-----------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean add(E <i>obj</i> )                          | Adiciona <i>obj</i> à coleção chamadora. Retorna <b>true</b> se <i>obj</i> tiver sido adicionado à coleção. Caso contrário, retorna <b>false</b> .                                                                                                                                                                                                                          |
| boolean addAll(Collection<? extends E> <i>col</i> ) | Adiciona os elementos de <i>col</i> à coleção chamadora. Retorna <b>true</b> se pelo menos um elemento tiver sido adicionado. Caso contrário, retorna <b>false</b> .                                                                                                                                                                                                        |
| void clear()                                        | Esvazia a coleção chamadora.                                                                                                                                                                                                                                                                                                                                                |
| boolean contains(Object <i>obj</i> )                | Retorna <b>true</b> se <i>obj</i> estiver na coleção chamadora. Caso contrário, retorna <b>false</b> .                                                                                                                                                                                                                                                                      |
| boolean containsAll(Collection<?> <i>col</i> )      | Retorna <b>true</b> se todos os elementos de <i>col</i> também estiverem na coleção chamadora. Caso contrário, retorna <b>false</b> .                                                                                                                                                                                                                                       |
| boolean equals(Object <i>obj</i> )                  | Retorna <b>true</b> se a coleção chamadora e <i>obj</i> forem iguais. Caso contrário, retorna <b>false</b> .                                                                                                                                                                                                                                                                |
| int hashCode()                                      | Retorna o código hash da coleção chamadora.                                                                                                                                                                                                                                                                                                                                 |
| boolean isEmpty()                                   | Retorna <b>true</b> se a coleção chamadora estiver vazia. Caso contrário, retorna <b>false</b> .                                                                                                                                                                                                                                                                            |
| Iterator<E> iterator()                              | Retorna um iterador para a coleção chamadora.                                                                                                                                                                                                                                                                                                                               |
| boolean remove(Object <i>obj</i> )                  | Remove um item igual a <i>obj</i> da coleção chamadora. Retorna <b>true</b> se for bem-sucedido. Retorna <b>false</b> se <i>obj</i> não estiver na coleção.                                                                                                                                                                                                                 |
| boolean removeAll(Collection<?> <i>col</i> )        | Remove todos os elementos de <i>col</i> da coleção chamadora. Retorna <b>true</b> se pelo menos um item for removido. Caso contrário, retorna <b>false</b> .                                                                                                                                                                                                                |
| boolean retainAll(Collection<?> <i>col</i> )        | Remove todos os elementos da coleção chamadora, exceto os de <i>col</i> . Retorna <b>true</b> se pelo menos um elemento for removido. Caso contrário, retorna <b>false</b> .                                                                                                                                                                                                |
| int size()                                          | Retorna o número de elementos armazenados atualmente na coleção chamadora.                                                                                                                                                                                                                                                                                                  |
| Object[ ] toArray()                                 | Retorna os elementos armazenados na coleção chamadora na forma de um array. O array é independente da coleção chamadora.                                                                                                                                                                                                                                                    |
| <T> T[ ] toArray(T[ ] <i>array</i> )                | Retorna os elementos armazenados na coleção chamadora na forma de um array. O array é independente da coleção chamadora. Se o tamanho de <i>array</i> for igual ao número de elementos ou maior, eles serão retornados em <i>array</i> . Se o tamanho de <i>array</i> for menor que o número de elementos, um novo array com o tamanho necessário será alocado e retornado. |

outra retorna um array de elementos que têm o mesmo tipo do array especificado como parâmetro. A segunda forma é conveniente porque retorna o tipo de array desejado. Esses métodos são mais importantes do que parecem porque você pode ter que processar o conteúdo de uma coleção como um array. Por exemplo, pode querer passar uma coleção para um método que espera um array. O array obtido em **toArray( )** é independente da coleção, exceto, claro, pelo fato de os dois referenciarem os mesmos elementos.

## A interface List

A interface **List** estende **Collection** e declara o comportamento de uma coleção que armazena uma sequência de elementos. Os elementos podem ser inseridos ou acessados por sua posição na lista, com o uso de um índice de base zero. É válido uma implementação de **List** conter duplicatas. **List** é uma interface genérica que tem esta declaração:

```
interface List<E>
```

Aqui, **E** especifica o tipo de objetos que a lista conterá.

Além dos métodos definidos por **Collection**, **List** adiciona alguns métodos próprios, que estão resumidos na Tabela 25-2.

Observe que a maioria dos métodos adicionados por **List** recebe um índice como argumento ou retorna um índice. Esses métodos dão suporte ao acesso aleatório a **List** baseado em índice. Por exemplo, para obter o objeto armazenado em um local específico, chame **get()**, especificando o índice do objeto. Para definir o valor de um local específico, chame **set()**, passando o índice e o valor. Para encontrar o índice de um objeto, use **indexOf()** e **lastIndexOf()**.

Outro método digno de nota é **subList()**. Ele permite a obtenção da sublista de uma lista com a especificação de seus índices inicial e final. Como era de se esperar, **subList()** torna o processamento de listas mais conveniente. Mas tome cuidado, porque a lista retornada usa a lista original. Logo, alterar uma muda a outra. Essa organização é chamada de visão da coleção. A documentação da API Java diz que a visão é “amparada” pela coleção em que é chamada. Outra coisa: se você fizer uma alteração na estrutura da coleção subjacente (exceto se for por intermédio da visão), por exemplo, alterando seu tamanho, a sublista perderá a validade. O importante é usar as sublistas com cuidado.

Outro fato interessante é que **List** altera a semântica dos métodos **add(E)** e **addAll(Collection)** definidos por **Collection** para que adicionem elementos ao fim da lista. Para adicionar um elemento em um local específico, use as formas de índice **add()** ou **addAll()** especificadas por **List**.

## A interface Set

A interface **Set** define um conjunto. Ela estende **Collection** e declara o comportamento de uma coleção que não permite elementos duplicados. Portanto, o método **add()** retorna **false** quando é feita uma tentativa de adicionar elementos duplicados a um conjunto. Ela não declara nenhum método adicional. **Set** é uma interface genérica que tem esta declaração:

```
interface Set<E>
```

Aqui, **E** especifica o tipo de objetos que o conjunto conterá.

## A interface SortedSet

A interface **SortedSet** estende **Set** e declara o comportamento de um conjunto classificado em ordem crescente. **SortedSet** é uma interface genérica que tem esta declaração:

```
interface SortedSet<E>
```

E especifica o tipo de objetos que o conjunto conterá. Além dos métodos fornecidos por **Set**, a interface **SortedSet** adiciona os métodos resumidos na Tabela 25-3.

**Tabela 25-2** Métodos declarados por List

| Método                                                                  | Descrição                                                                                                                                                                                       |
|-------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void add(int <i>índice</i> , E <i>obj</i> )                             | Adiciona <i>obj</i> à lista chamadora no índice passado em <i>índice</i> .                                                                                                                      |
| boolean addAll(int <i>índice</i> , Collection<? extends E> <i>col</i> ) | Adiciona os elementos de <i>col</i> à lista chamadora no índice passado em <i>índice</i> . Retorna <b>true</b> se pelo menos um elemento for adicionado. Caso contrário, retorna <b>false</b> . |
| E get(int <i>índice</i> )                                               | Retorna o objeto armazenado no índice passado em <i>índice</i> na coleção chamadora.                                                                                                            |
| int indexOf(Object <i>obj</i> )                                         | Procura a primeira ocorrência de <i>obj</i> na lista chamadora. Se encontrada, seu índice será retornado. Caso contrário, $-1$ será retornado.                                                  |
| int lastIndexOf(Object <i>obj</i> )                                     | Procura a última ocorrência de <i>obj</i> na lista chamadora. Se encontrada, seu índice será retornado. Caso contrário, $-1$ será retornado.                                                    |
| ListIterator<E> listIterator()                                          | Retorna um iterador para o início da lista chamadora.                                                                                                                                           |
| ListIterator<E> listIterator(int <i>índice</i> )                        | Retorna um iterador para a lista chamadora que começa no índice passado em <i>índice</i> .                                                                                                      |
| E remove(int <i>índice</i> )                                            | Remove da lista chamadora o elemento do índice passado em <i>índice</i> e retorna o elemento excluído.                                                                                          |
| E set(int <i>índice</i> , E <i>obj</i> )                                | Atribui <i>obj</i> ao local especificado por <i>índice</i> na lista chamadora. Retorna o valor anterior.                                                                                        |
| List<E> subList(int <i>início</i> , int <i>fim</i> )                    | Retorna uma lista que inclui elementos de <i>início</i> a <i>fim</i> – 1 da lista chamadora. A lista resultante é uma visão da lista chamadora.                                                 |

**Tabela 25-3** Métodos declarados por SortedSet

| Método                                               | Descrição                                                                                                                                                                       |
|------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Comparator<? super E> comparator()                   | Retorna o comparador do conjunto chamador. Se nenhum comparador for usado, <b>null</b> será retornado.                                                                          |
| E first()                                            | Retorna o elemento de menor valor. Já que o conjunto é classificado, ele será o primeiro elemento do conjunto chamador.                                                         |
| SortedSet<E> headSet(E <i>fim</i> )                  | Retorna um <b>SortedSet</b> que inclui os elementos do conjunto chamador menores do que <i>fim</i> . O conjunto resultante é uma visão do conjunto chamador.                    |
| E last()                                             | Retorna o elemento de maior valor. Já que o conjunto é classificado, ele será o último elemento do conjunto chamador.                                                           |
| SortedSet<E> subSet(E <i>início</i> , E <i>fim</i> ) | Retorna um <b>SortedSet</b> incluindo os elementos do conjunto chamador que estão entre <i>início</i> e <i>fim</i> – 1. O conjunto resultante é uma visão do conjunto chamador. |
| SortedSet<E> tailSet(E <i>início</i> )               | Retorna um <b>SortedSet</b> incluindo os elementos do conjunto chamador maiores ou iguais a <i>início</i> . O conjunto resultante é uma visão do conjunto chamador.             |

Observe que **SortedSet** define três métodos que obtêm subconjuntos. Eles podem tornar o processamento de conjuntos mais conveniente. Você pode obter um subconjunto de um conjunto classificado chamando **subSet( )** e especificando o primeiro e o último objeto do conjunto desejado. Se precisar que o subconjunto comece com o primeiro elemento do conjunto, use **headSet( )**. Se quiser o subconjunto que termina o conjunto, use **tailSet( )**. Esses métodos facilitam a obtenção de um intervalo dos elementos de um conjunto. Em todos os casos, o conjunto retornado é uma “visão”, ou seja, referencia o conjunto original. A alteração de um afeta o outro. Diferentemente das visões obtidas de um **List**, essas visões permitem alterações na estrutura da coleção subjacente sem que sejam invalidadas. Um último ponto: você não pode adicionar às visões um elemento que esteja fora de seu intervalo. Tentar fazê-lo causará uma exceção.

## A interface NavigableSet

A interface **NavigableSet** estende **SortedSet** e declara o comportamento de uma coleção que dá suporte à recuperação de um elemento com base em quanto seu valor está próximo de outro. **NavigableSet** é uma interface genérica que tem esta declaração:

```
interface NavigableSet<E>
```

Aqui, **E** especifica o tipo de objetos que o conjunto conterá. Além dos métodos herdados de **SortedSet**, **NavigableSet** adiciona os métodos resumidos na Tabela 25-4.

Preste atenção nos métodos **ceiling( )** e **floor( )**. Eles encontram o próximo elemento do conjunto mais próximo ou igual a outro elemento. Para encontrar um elemento que esteja mais próximo mas não seja igual a outro, use **higher( )** e **lower( )**. Observe também que foram adicionadas formas adicionais de **subSet( )**, **tailSet( )** e **headSet( )** que trabalham com ocorrências próximas.

## A interface Queue

A interface **Queue** estende **Collection** e declara o comportamento de uma fila. Normalmente, trata-se de uma lista “primeiro a entrar, primeiro a sair”. No entanto, há tipos de filas em que a ordem é baseada em outros critérios. **Queue** é uma interface genérica que tem esta declaração:

```
interface Queue<E>
```

Aqui, **E** especifica o tipo de objetos que a fila conterá. Os métodos adicionados por **Queue** são mostrados na Tabela 25-5.

Apesar de sua simplicidade, **Queue** oferece vários pontos de interesse. Em primeiro lugar, elementos só podem ser removidos da cabeça (início) da fila. Em segundo lugar, o ponto de inserção do elemento não é especificado. Para uma fila FIFO, os elementos seriam adicionados ao fim da fila, mas outras implementações são permitidas. Em terceiro lugar, há dois métodos que obtêm e removem elementos: **remove( )** e **poll( )**. A diferença entre eles é o que ocorre quando chamados em uma fila vazia. Nesse caso, o método **remove( )** lança uma exceção **NoSuchElementException**. O método **poll( )** retorna **null**. Em quarto lugar, há dois métodos, **element( )** e **peek( )**, que obtêm, mas não removem, o elemento do início da fila. Eles também diferem em seu comportamento quando chamados em uma fila vazia, o primeiro lançando uma exceção **NoSuchElementException** e o outro retornando **null**.

**Tabela 25-4** Métodos declarados por NavigableSet

| Método                                                                                                                                   | Descrição                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| E ceiling(E <i>obj</i> )                                                                                                                 | Procura no conjunto o menor elemento <i>e</i> de modo que <i>e</i> $\geq$ <i>obj</i> . Se esse elemento for encontrado, ele será retornado. Caso contrário, <b>null</b> será retornado.                                                                                                                                                                                                                                    |
| Iterator<E> descendingIterator( )                                                                                                        | Retorna um iterador que se move do maior para o menor. Em outras palavras, retorna um iterador inverso.                                                                                                                                                                                                                                                                                                                    |
| NavigableSet<E> descendingSet( )                                                                                                         | Retorna um <b>NavigableSet</b> contendo as entradas do conjunto chamador em ordem inversa. O conjunto resultante é uma visão do conjunto chamador.                                                                                                                                                                                                                                                                         |
| E floor(E <i>obj</i> )                                                                                                                   | Procura no conjunto o maior elemento <i>e</i> de modo que <i>e</i> $\leq$ <i>obj</i> . Se esse elemento for encontrado, ele será retornado. Caso contrário, <b>null</b> será retornado.                                                                                                                                                                                                                                    |
| NavigableSet<E><br>headSet(E <i>limiteSuperior</i> , boolean <i>incl</i> )                                                               | Retorna um <b>NavigableSet</b> incluindo todos os elementos do conjunto chamador que forem menores do que <i>limiteSuperior</i> . Se <i>incl</i> for <b>true</b> , um elemento igual a <i>limiteSuperior</i> será incluído. O conjunto resultante é uma visão do conjunto chamador.                                                                                                                                        |
| E higher(E <i>obj</i> )                                                                                                                  | Procura no conjunto o menor elemento <i>e</i> de modo que <i>e</i> $>$ <i>obj</i> . Se esse elemento for encontrado, ele será retornado. Caso contrário, <b>null</b> será retornado.                                                                                                                                                                                                                                       |
| E lower(E <i>obj</i> )                                                                                                                   | Procura no conjunto o maior elemento <i>e</i> de modo que <i>e</i> $<$ <i>obj</i> . Se esse elemento for encontrado, ele será retornado. Caso contrário, <b>null</b> será retornado.                                                                                                                                                                                                                                       |
| E pollFirst( )                                                                                                                           | Retorna o primeiro elemento do conjunto chamador, removendo-o no processo. Já que o conjunto é classificado, trata-se do elemento de menor valor. Retorna <b>null</b> se chamado em um conjunto vazio.                                                                                                                                                                                                                     |
| E pollLast( )                                                                                                                            | Retorna o último elemento do conjunto chamador, removendo-o no processo. Já que o conjunto é classificado, trata-se do elemento de maior valor. Retorna <b>null</b> se chamado em um conjunto vazio.                                                                                                                                                                                                                       |
| NavigableSet<E><br>subSet(E <i>limiteInferior</i> ,<br>boolean <i>inclInf</i> ,<br>E <i>limiteSuperior</i> ,<br>boolean <i>inclSup</i> ) | Retorna um <b>NavigableSet</b> incluindo todos os elementos do conjunto chamador que forem maiores do que <i>limiteInferior</i> e menores do que <i>limiteSuperior</i> . Se <i>inclInf</i> for <b>true</b> , um elemento igual a <i>limiteInferior</i> será incluído. Se <i>inclSup</i> for <b>true</b> , um elemento igual a <i>limiteSuperior</i> será incluído. O conjunto resultante é uma visão do conjunto chamador. |
| NavigableSet<E><br>tailSet(E <i>limiteInferior</i> , boolean <i>incl</i> )                                                               | Retorna um <b>NavigableSet</b> incluindo todos os elementos do conjunto chamador que forem maiores do que <i>limiteInferior</i> . Se <i>incl</i> for <b>true</b> , um elemento igual a <i>limiteInferior</i> será incluído. O conjunto resultante é uma visão do conjunto chamador.                                                                                                                                        |

**Tabela 25-5** Métodos declarados por Queue

| Método               | Descrição                                                                                                                                                                                                                                    |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| E element( )         | Retorna o elemento do início da fila. O elemento não é removido. Se chamado em uma fila vazia, uma <b>NoSuchElementException</b> será lançada.                                                                                               |
| boolean offer(E obj) | Tenta adicionar <i>obj</i> à fila. Retorna <b>true</b> se <i>obj</i> for adicionado; caso contrário, retorna <b>false</b> . Esse método falhará se for feita uma tentativa de adicionar um elemento a uma fila de capacidade restrita cheia. |
| E peek( )            | Retorna o elemento do início da fila. O elemento não é removido. Retorna <b>null</b> se chamado em uma fila vazia.                                                                                                                           |
| E poll( )            | Retorna o elemento do início da fila, removendo-o no processo. Retorna <b>null</b> se chamado em uma fila vazia.                                                                                                                             |
| E remove( )          | Remove o elemento do início da fila, retornando-o no processo. Se chamado em uma fila vazia, uma <b>NoSuchElementException</b> será lançada.                                                                                                 |

Para concluir, observe que **offer()** apenas tenta adicionar um elemento a uma fila. Embora as implementações de filas de **java.util** sejam dinâmicas, é permitida a implementação de uma fila de capacidade restrita. Por exemplo, uma fila pode ter tamanho fixo. Em uma fila de capacidade restrita, **offer()** pode falhar quando a fila estiver cheia. Nesse caso, ele retorna **false**. O método **add()**, que é herdado de **Collection**, também pode ser usado para adicionar um elemento a uma fila. Se a fila tiver tamanho fixo, ele lançará uma **IllegalStateException** quando chamado em uma fila cheia.

## A interface Deque

A interface **Deque** estende **Queue** e declara o comportamento de uma fila de extremidade dupla. As filas de extremidade dupla funcionam como filas FIFO padrão ou como pilhas LIFO. **Deque** é uma interface genérica que tem esta declaração:

```
interface Deque<E>
```

Aqui, **E** especifica o tipo de objetos que a deque conterá. Além dos métodos que herda de **Queue**, **Deque** adiciona os mostrados na Tabela 25-6.

Observe que **Deque** inclui os métodos **push()** e **pop()**. Esses métodos permitem que um **Deque** funcione como uma pilha. É claro que você também pode usar um **Deque** como uma fila. Há várias maneiras de fazer isso. Esta é uma delas: use **offerLast()** para inserir um item no fim da fila e **pollFirst()** para remover um item do início dela.

Embora as implementações de **Deque** em **java.util** sejam dinâmicas, também podemos ter implementações de capacidade restrita. Quando for esse o caso, uma tentativa de adicionar um elemento à deque pode falhar. **Deque** permite lidar com essa falha de duas maneiras. Em primeiro lugar, métodos como **addFirst()** e **addLast()** lançam uma **IllegalStateException** se uma deque de capacidade restrita estiver cheia. Em segundo lugar, métodos como **offerFirst()** e **offerLast()** retornam **false** se o ele-

**Tabela 25-6** Métodos declarados por Deque

| Método                            | Descrição                                                                                                                                                                                                                                                                         |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void addFirst(E <i>obj</i> )      | Tenta adicionar <i>obj</i> ao início da deque. Lança uma <b>IllegalStateException</b> se uma deque de capacidade restrita estiver cheia.                                                                                                                                          |
| void addLast(E <i>obj</i> )       | Tenta adicionar <i>obj</i> ao fim da deque. Lança uma <b>IllegalStateException</b> se uma deque de capacidade restrita estiver cheia.                                                                                                                                             |
| Iterator<E> descendingIterator()  | Retorna um iterador que se move do fim ao início da deque. Em outras palavras, retorna um iterador inverso.                                                                                                                                                                       |
| E getFirst()                      | Retorna o elemento do início da deque. O objeto não é removido. Se chamado em uma deque vazia, uma <b>NoSuchElementException</b> é lançada.                                                                                                                                       |
| E getLast()                       | Retorna o elemento do fim da deque. O objeto não é removido. Se chamado em uma deque vazia, uma <b>NoSuchElementException</b> é lançada.                                                                                                                                          |
| boolean offerFirst(E <i>obj</i> ) | Tenta adicionar <i>obj</i> ao início da deque. Retorna <b>true</b> se <i>obj</i> for adicionado; caso contrário, retorna <b>false</b> . Portanto, esse método retorna <b>false</b> quando é feita uma tentativa de adicionar <i>obj</i> a uma deque de capacidade restrita cheia. |
| boolean offerLast(E <i>obj</i> )  | Tenta adicionar <i>obj</i> ao fim da deque. Retorna <b>true</b> se <i>obj</i> for adicionado; caso contrário, retorna <b>false</b> . Portanto, esse método retorna <b>false</b> quando é feita uma tentativa de adicionar <i>obj</i> a uma deque de capacidade restrita cheia.    |
| E peekFirst()                     | Retorna o elemento do início da deque. O objeto não é removido. Retorna <b>null</b> se chamado em uma deque vazia.                                                                                                                                                                |
| E peekLast()                      | Retorna o elemento do fim da deque. O objeto não é removido. Retorna <b>null</b> se chamado em uma deque vazia.                                                                                                                                                                   |
| E pollFirst()                     | Retorna o elemento do início da deque, removendo-o no processo. Retorna <b>null</b> se chamado em uma deque vazia.                                                                                                                                                                |
| E pollLast()                      | Retorna o elemento do fim da deque, removendo-o no processo. Retorna <b>null</b> se chamado em uma deque vazia.                                                                                                                                                                   |
| E pop()                           | Retorna o elemento do início da deque, removendo-o no processo. Se chamado em uma deque vazia, uma <b>NoSuchElementException</b> é lançada.                                                                                                                                       |
| void push(E <i>obj</i> )          | Adiciona <i>obj</i> ao início da deque. Lança uma <b>IllegalStateException</b> se uma deque de capacidade restrita estiver cheia.                                                                                                                                                 |
| E removeFirst()                   | Retorna o elemento do início da deque, removendo-o no processo. Se chamado em uma deque vazia, uma <b>NoSuchElementException</b> é lançada.                                                                                                                                       |

**Tabela 25-6** Métodos declarados por Deque (continuação)

| Método                                               | Descrição                                                                                                                                                                       |
|------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean<br>removeFirstOccurrence(Object <i>obj</i> ) | Procura a primeira ocorrência de <i>obj</i> . Se encontrada, o elemento é removido. Retorna <b>true</b> se for bem-sucedido e <b>false</b> se a deque não contiver <i>obj</i> . |
| E removeLast( )                                      | Retorna o elemento do fim da deque, removendo-o no processo. Se chamado em uma deque vazia, uma <b>NoSuchElementException</b> é lançada                                         |
| boolean<br>removeLastOccurrence(Object <i>obj</i> )  | Procura a última ocorrência de <i>obj</i> . Se encontrada, o elemento é removido. Retorna <b>true</b> se for bem-sucedido e <b>false</b> se a deque não contiver <i>obj</i> .   |

mento não puder ser adicionado. Você também tem duas maneiras de tratar a tentativa de remoção de um elemento de uma deque vazia. Métodos como **removeFirst()** e **removeLast()** lançam uma **NoSuchElementException**. Métodos como **pollFirst()** e **pollLast()** retornam **null**.

Um último ponto: observe o método **descendingIterator()**. Ele fornece um iterador que retorna elementos em ordem inversa. Em outras palavras, fornece um iterador que se move do fim ao início da deque.

### Verificação do progresso

1. O Collections Framework fornece implementações das estruturas de dados padrão. Verdadeiro ou falso?
2. A interface **Collections** define todos os métodos a seguir: **add()**, **remove()**, **clear()**, **contains()**, **isEmpty()**, **iterator()**, **size()**, **toArray()**. Verdadeiro ou falso?
3. Um **List** pode armazenar duplicatas. Verdadeiro ou falso?
4. Um **Set** pode armazenar duplicatas. Verdadeiro ou falso?

## AS CLASSE DE COLEÇÕES

As interfaces de coleções são implementadas por várias classes. Algumas das classes fornecem implementações que podem ser usadas como se encontram. Outras são abstratas, fornecendo implementações parciais que são usadas como pontos de partida para a criação de coleções concretas. Como regra geral, as classes de coleções não são sincronizadas.

As classes de coleções padrão empacotadas em **java.util** estão resumidas na tabela a seguir:

### Respostas:

1. Verdadeiro.
2. Verdadeiro. Também define vários outros métodos.
3. Verdadeiro.
4. Falso.

| Classe                 | Descrição                                                                                                                         |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| AbstractCollection     | Implementa partes da interface <b>Collection</b> .                                                                                |
| AbstractList           | Estende <b>AbstractCollection</b> e implementa partes da interface <b>List</b> .                                                  |
| AbstractQueue          | Estende <b>AbstractCollection</b> e implementa partes da interface <b>Queue</b> .                                                 |
| AbstractSequentialList | Estende <b>AbstractList</b> para ser usada por uma coleção projetada para acesso sequencial em vez de aleatório a seus elementos. |
| AbstractSet            | Estende <b>AbstractCollection</b> e implementa partes da interface <b>Set</b> .                                                   |
| ArrayList              | Implementa um array dinâmico estendendo <b>AbstractList</b> .                                                                     |
| ArrayDeque             | Implementa uma fila de extremidade dupla estendendo <b>AbstractCollection</b> . Também implementa a interface <b>Deque</b> .      |
| EnumSet                | Estende <b>AbstractSet</b> para ser usada com elementos <b>enum</b> .                                                             |
| HashSet                | Implementa um conjunto armazenado em uma tabela hash estendendo <b>AbstractSet</b> .                                              |
| LinkedHashSet          | Estende <b>HashSet</b> para permitir iterações por ordem de inserção.                                                             |
| LinkedList             | Implementa uma lista encadeada estendendo <b>AbstractSequentialList</b> . Também implementa <b>Deque</b> .                        |
| PriorityQueue          | Implementa uma fila baseada em prioridades estendendo <b>AbstractQueue</b> .                                                      |
| TreeSet                | Implementa um conjunto armazenado em uma árvore estendendo <b>AbstractSet</b> . Também implementa <b>SortedSet</b> .              |

As classes abstratas, como **AbstractCollection**, fornecem implementações parciais que são complementadas pelas classes concretas. **EnumSet** é uma coleção especializada usada para armazenar elementos **enum** e não será examinada aqui. As outras classes de coleções concretas serão descritas nas próximas seções.

**Nota:** Além das classes de coleções que acabamos de mostrar, **java.util** inclui várias classes legadas que também trabalham com grupos de dados. Elas são **Vector**, **Stack** e **Hashtable**. As classes legadas fazem parte de Java desde o começo. Elas serão descritas resumidamente no fim deste capítulo.

## A classe ArrayList

A classe **ArrayList** estende **AbstractList** e implementa a interface **List**. **ArrayList** é uma classe genérica que tem esta declaração:

```
class ArrayList<E>
```

Aqui, **E** especifica o tipo de objetos que a lista conterá.

**ArrayList** dá suporte a arrays dinâmicos que podem crescer quando necessário. Em Java, os arrays padrão têm tamanho fixo. Após um array ser criado, ele não pode crescer ou diminuir, ou seja, temos que saber antecipadamente quantos elementos um array conterá. Mas às vezes só conseguimos saber o tamanho exato do array no tempo de execução. Para lidar com essa situação, o Collections Framework define **ArrayList**. Basicamente, um **ArrayList** é um array de tamanho variável com referências de objeto. Isto é, um **ArrayList** pode aumentar ou diminuir de tamanho dinamicamente. As listas de array são criadas com um tamanho inicial. Quando esse tamanho é excedido, a coleção é aumentada automaticamente. Quando objetos são re-

movidos, o array pode ser diminuído. Os elementos de um **ArrayList** são acessados por intermédio de um índice.

**ArrayList** tem os construtores mostrados aqui:

```
ArrayList()
ArrayList(Collection<? extends E> col)
ArrayList(int capacidadeInicial)
```

O primeiro construtor cria uma lista de array vazia. O segundo constrói uma lista de array inicializada com os elementos da coleção *col*. O terceiro cria uma lista de array com a capacidade inicial passada em *capacidadeInicial*. A capacidade é o tamanho do array subjacente usado para armazenar os elementos. Em todos os casos, a capacidade aumenta automaticamente à medida que elementos são adicionados a uma lista de array. A capacidade inicial padrão é igual a 10.

O programa a seguir demonstra **ArrayList**. Uma lista de array é criada para objetos de tipo **Character** e vários caracteres são adicionados a ela. Em seguida, algumas operações são executadas na lista. Embora uma coleção só possa armazenar referências, podemos adicionar literais **char** porque o *autoboxing* os inserirá automaticamente como objetos **Character**.

```
// Demonstra ArrayList.

import java.util.*;

class ArrayListDemo {
 public static void main(String[] args) {
 // Cria uma lista de array.
 ArrayList<Character> al = new ArrayList<Character>(); ← Constrói um
 System.out.println("Initial size: " + al.size()); ArrayList para
 caracteres.

 // Adiciona elementos ao fim da lista, um de cada vez.
 al.add('A'); ←
 al.add('B'); ←
 al.add('C'); ← Adiciona elementos à lista.
 al.add('D'); ←
 al.add('E'); ←

 System.out.println("\nSize after additions: " + al.size());

 // Exibe a lista de array usando a representação de toString().
 System.out.println("Contents: " + al);

 // Agora, adiciona elementos ao meio da lista.
 // Isso fará o array se expandir.
 for(int i = 0; i < 3; i++)
 al.add(2, (char) ('x' + i));

 System.out.println("\nSize after additions: " + al.size());
 System.out.println("Contents: " + al);
```

```

// Exclui os elementos recém-adicionados.
// Isso fará o array se contrair.
for(int i = 0; i < 3; i++)
 al.remove(2); ← Remove um elemento.

System.out.println("\nSize after deletions: " + al.size());
System.out.println("Contents: " + al);

// Usa set() para definir o valor de um índice.
for(int i=0; i < al.size(); i++)
 al.set(i, Character.toLowerCase(al.get(i))); ← Altera um elemento.

System.out.println("\nAfter changing to lowercase.");
System.out.println("Contents: " + al);

// Encontra e remove um valor
int idx = al.indexOf('d'); ← Encontra o índice de um valor.
if(idx >= 0) al.remove(idx);

System.out.println("\nAfter finding and removing d.");
System.out.println("Contents: " + al);

// Esvazia a lista.
al.clear(); ← Remove todos os elementos da lista.
System.out.println("\nAfter clearing the list.");
System.out.println("Contents: " + al);

// Adiciona os dígitos de 0 a 9
for(int i=0; i < 10; i++)
 al.add((char) ('0' + i));

// Exibe elementos alternados.
System.out.print("\nHere is every other digit: ");
for(int i=0; i < al.size(); i+=2)
 System.out.print(al.get(i) + " ");
}
}

```

A saída desse programa é:

```

Initial size: 0

Size after additions: 5
Contents: [A, B, C, D, E]

Size after additions: 8
Contents: [A, B, z, y, x, C, D, E]

Size after deletions: 5
Contents: [A, B, C, D, E]

After changing to lowercase.

```

```
Contents: [a, b, c, d, e]

After finding and removing d.
Contents: [a, b, c, e]

After clearing the list.
Contents: []

Here is every other digit: 0 2 4 6 8
```

Observe que **al** começa vazia e seu tamanho cresce à medida que elementos são adicionados. Quando elementos são removidos, o tamanho diminui. Como **ArrayList** implementa um array dinâmico, seus elementos podem ser acessados por intermédio de um índice. Isso é demonstrado com o uso das formas de índice **add()**, **remove()**, **get()** e **set()**. Por exemplo, considere esta linha do programa:

```
| al.set(i, Character.toLowerCase(al.get(i)));
```

Ela usa **get()** para obter o caractere do índice especificado e, então, usa **set()** para armazenar a versão minúscula desse caractere no mesmo local. Observe também que a forma de **add()** que não usa índice apenas insere novos elementos no fim do array.

No exemplo, repare que o conteúdo de uma coleção é exibido com o uso da conversão padrão fornecida por **toString()**, que foi herdado de **AbstractCollection**. A conversão de **toString()** é muito conveniente nos testes com coleções porque fornece uma maneira fácil de vermos seus conteúdos. É claro que também podemos exibir os elementos de uma coleção individualmente, usando um laço, como a última linha da saída faz.

Embora a capacidade de um objeto **ArrayList** aumente automaticamente à medida que objetos são armazenados nele, você pode aumentar sua capacidade manualmente chamando **ensureCapacity()**. Talvez queira fazê-lo se souber de antemão que armazenará na coleção muito mais itens do que ela pode conter. Aumentando sua capacidade uma vez, no início, poderá evitar várias realocações posteriores. Já que as realocações são dispendiosas em termos de tempo, evitar realocações desnecessárias melhora o desempenho. O método **ensureCapacity()** é mostrado abaixo:

```
void ensureCapacity(int cap)
```

Aqui, *cap* especifica a nova capacidade mínima da coleção.

Inversamente, se quiser reduzir a capacidade de um objeto **ArrayList** para que se ajuste ao número de itens que contém atualmente, chame o método **trimToSize()**, mostrado a seguir:

```
void trimToSize()
```

## A classe **LinkedList**

A classe **LinkedList** estende **AbstractSequentialList** e implementa as interfaces **List** e **Deque**. Ela fornece uma estrutura de dados de lista duplamente encadeada. **LinkedList** é uma classe genérica que tem esta declaração:

```
class LinkedList<E>
```

E especifica o tipo de objetos que a lista conterá. **LinkedList** tem os dois construtores abaixo:

```
LinkedList()
LinkedList(Collection<? extends E> col)
```

O primeiro construtor cria uma lista encadeada vazia. O segundo constrói uma lista encadeada inicializada com os elementos da coleção *col*.

Já que **LinkedList** implementa a interface **Deque**, temos acesso aos métodos definidos por ela. Por exemplo, para adicionar elementos ao início de uma lista, podemos usar **addFirst()** ou **offerFirst()**. Para adicionar elementos ao fim da lista, usaremos **addLast()** ou **offerLast()**. Para obter o primeiro elemento, podemos usar **getFirst()** ou **peekFirst()**. Para obter o último elemento, teríamos **getLast()** ou **peekLast()**. Para remover o primeiro elemento, temos **removeFirst()** ou **pollFirst()**. Para remover o último, usaremos **removeLast()** ou **pollLast()**. É claro que também temos acesso a toda a funcionalidade fornecida por **List**, como acessar a lista via um índice ou obter uma sublista.

O programa a seguir demonstra vários recursos de **LinkedList**:

```
// Demonstra LinkedList.

import java.util.*;

class LinkedListDemo {
 public static void main(String[] args) {
 // Cria uma lista encadeada.
 LinkedList<Character> ll = new LinkedList<Character>();

 // Adiciona elementos à lista encadeada.
 ll.add('B');
 ll.add('E');
 ll.add('F');
 System.out.println("Original contents: " + ll);

 // Demonstra addLast() e addFirst().
 ll.addLast('G'); ← Adiciona elementos ao fim e ao início da lista.
 ll.addFirst('A');
 System.out.println("\nAfter calls to addFirst() and addLast().");
 System.out.println("Contents: " + ll);

 // Adiciona elementos em um índice.
 ll.add(2, 'D'); ← Adiciona elementos em um índice específico.
 ll.add(2, 'C');
 System.out.println("\nAfter insertions.");
 System.out.println("Contents: " + ll);

 // Exibe o primeiro e o último elementos.
 System.out.println("\nHere are the first and last elements: " +
 ll.getFirst() + " " + ll.getLast()); ←
 }
}
```

Obtém os elementos do  
início e do fim da lista.

```

// Cria uma visão de sublistas.
List<Character> sub = ll.subList(2, 5); ← Obtém a visão de uma sublistas.
System.out.println("\nContents of sublist visão: " + sub);

// Cria uma nova lista contendo a sublistas
LinkedList<Character> ll2 = new LinkedList<Character>(sub);

// Remove de ll os elementos de ll2.
ll.removeAll(ll2); ← Remove de ll todos os elementos de ll2.

System.out.println("\nAfter removing ll2 from ll.");
System.out.println("Contents: " + ll);

// Remove o primeiro e o último elementos.
ll.removeFirst(); ← Remove o primeiro e o último elementos.
ll.removeLast();

System.out.println("\nAfter deleting first and last element: ");
System.out.println("Contents: " + ll);

// Obtém e define um valor usando um índice.
ll.set(0, Character.toLowerCase(ll.get(0))); ← Altera um valor.

System.out.println("\nAfter change: " + ll);
}
}

```

A saída do programa é mostrada abaixo:

```

Original contents: [B, E, F]

After calls to addFirst() and addLast().
Contents: [A, B, E, F, G]

After insertions.
Contents: [A, B, C, D, E, F, G]

Here are the first and last elements: A G

Contents of sublist view: [C, D, E]

After removing ll2 from ll.
Contents: [A, B, F, G]

After deleting first and last element:
Contents: [B, F]

After change: [b, F]

```

Como o programa mostra, já que **LinkedList** implementa a interface **List**, chamadas a **add()** acrescentam itens ao fim da lista, como ocorre em chamadas a **addLast()**. Para inserir itens em um local específico, use a forma **add(int, E)** de **add()**, como ilustrado pela chamada a **add(2, 'D')** no exemplo. Os métodos **removeFirst()** e **removeLast()** fornecem uma maneira fácil de remoção do primeiro e do último elementos sem ser preciso especificar um índice.

Observe como uma parte de **l1** é obtida com a chamada a **sublist()**. Lembre-se, **sublist()** retorna uma “visão” que usa a lista em que é chamado. No entanto, em um **List**, essa visão é invalidada se ocorrer alguma alteração no tamanho da coleção subjacente. É por isso que um novo **LinkedList** deve ser criado contendo os elementos de **sub** antes de **removeAll()** poder ser usado para excluí-los de **l1**. Se você tentasse chamar **removeAll()** usando **sub** diretamente, como mostrado aqui,

```
| l1.removeAll(sub); // Errado!
```

ele lançaria uma exceção porque o tamanho de **l1** mudaria durante o processo de remoção, invalidando a visão.

### TENTE ISTO 25-1 Visões de coleção versus arrays obtidos a partir de uma coleção

`ViewAndArrayDemo.java`

Vimos que várias das interfaces do Collections Framework têm métodos que retornam uma coleção que usa a mesma estrutura de dados subjacente da coleção original. Como explicado, isso se chama *visão* da coleção. O método **subList()** da interface **List** é um exemplo desse tipo de método. Já que uma visão não é independente da coleção subjacente, se alterarmos uma, alteraremos a outra. Também foi explicado que o array retornado pelo método **toArray()** da interface **Collection** é independente da coleção, ou seja, fornece seu próprio espaço de armazenamento para os elementos da coleção, em vez de usar uma visão. Logo, alterações nele *não* afetam a coleção. Neste projeto, criaremos um programa pequeno que verifica essa diferença-chave entre a visão de uma coleção e um array gerado a partir de uma coleção.

#### PASSO A PASSO

1. Crie um novo arquivo chamado **ViewAndArrayDemo.java** e insira o código a seguir:

```
import java.util.*;

public class ViewAndArrayDemo {
 public static void main(String[] args) {
 LinkedList<String> list = new LinkedList<String>();
 list.add("A");
 list.add("B");
 }
}
```

```

 list.add("C");
 list.add("D");

 String[] array = list.toArray(new String[4]);
 List<String> sublist = list.subList(0,4);

```

Inicialmente construímos um **LinkedList** simples de strings contendo "A", "B", "C" e "D". Em seguida, chamamos **toArray()**, que retorna um array contendo os mesmos strings da lista. Então, chamamos **subList()**, que retorna uma lista também contendo os mesmos strings da lista original. Agora estamos prontos para verificar a diferença entre **array** e **sublist** em seu relacionamento com a lista original.

2. Adicione o código a seguir para completar o arquivo **ViewAndArrayDemo.java**:

```

array[1] = "F";
sublist.set(2,"O");
System.out.println(list);
}
}

```

A primeira linha altera o segundo string de **array** para "F" e a segunda linha altera o terceiro string de **sublist** para "O". A última linha exibe a lista original para vermos se as alterações em **array** ou **sublist** afetaram.

3. Compile e execute o programa **ViewAndArrayDemo**. Você verá a saída abaixo:

```
| [A, B, O, D]
```

Essa saída mostra que a alteração no array não alterou a lista original, confirmado o fato de o array ser independente da lista. No entanto, a alteração em **sublist** (a visão) causou uma alteração correspondente na lista original. Isso confirma o fato de que a visão usa a mesma estrutura de dados subjacente da lista original. Em outras palavras, **sublist** é apenas uma “visão” diferente da lista. Resumindo, é importante entender a diferença entre a visão de uma coleção e um array obtido a partir de uma coleção. Efeitos diferentes ocorrem quando um deles é alterado.

## A classe HashSet

**HashSet** estende **AbstractSet** e implementa a interface **Set**. Ela cria uma coleção que usa uma tabela hash para armazenamento. **HashSet** é uma classe genérica que tem esta declaração:

```
class HashSet<E>
```

Aqui, **E** especifica o tipo de objetos que o conjunto conterá.

Como explicado anteriormente neste capítulo, uma tabela hash armazena informações usando um mecanismo chamado hashing. No hashing, o conteúdo de uma chave é usado na determinação de um valor exclusivo, chamado seu *código hash*. O código hash é então usado como um tipo de índice empregado na localização dos dados associados à chave. No uso de um **HashSet**, a transformação da chave em seu código hash

é executada automaticamente – não precisamos calcular o código hash. A vantagem do hashing é que ele permite que o tempo de execução de **add()**, **contains()**, **remove()** e **size()** permaneçam constantes até mesmo para conjuntos grandes.

**HashSet** tem os construtores a seguir:

```
HashSet()
HashSet(Collection<? extends E> col)
HashSet(int capacidadeInicial)
HashSet(int capacidadeInicial, float taxaPreenchimento)
```

O primeiro construtor cria um conjunto hash vazio. A segunda forma inicializa o conjunto hash usando os elementos de *col*. A terceira inicializa a capacidade inicial do conjunto hash com *capacidadeInicial*. A capacidade inicial padrão é 16. A quarta forma inicializa tanto a capacidade inicial quanto a taxa de preenchimento (também chamada de *fator de carga*) do conjunto hash a partir de seus argumentos. A taxa de preenchimento deve ficar entre 0,0 e 1,0 e determina até onde o conjunto hash pode ser preenchido antes de ter seu tamanho aumentado. Especificamente, quando o número de elementos é maior do que a capacidade do conjunto hash multiplicada por sua taxa de preenchimento, o conjunto é expandido. Para construtores que não usam uma taxa de preenchimento, 0,75 é usado.

É importante observar que **HashSet** não armazena seus elementos em uma ordem específica, porque o processo de hashing não serve para a criação de conjuntos classificados. Se você precisar de armazenamento classificado, então outra coleção, como **TreeSet**, seria uma melhor opção.

Veja um exemplo que demonstra **HashSet**:

```
// Demonstra HashSet.

import java.util.*;

class HashSetDemo {
 public static void main(String[] args) {
 // Cria um conjunto hash.
 HashSet<Character> hs = new HashSet<Character>();

 // Adiciona elementos ao conjunto hash.
 hs.add('A');
 hs.add('B');
 hs.add('C');
 hs.add('D');
 System.out.println("Original contents: " + hs);

 // Adiciona mais elementos.
 hs.add('E');
 hs.add('F');
 hs.add('G');
 hs.add('H');
 System.out.println("\nContents after additions: " + hs);

 // Exclui E e H.
 hs.remove('E');
```

```

 hs.remove('H');
 System.out.println("\nContents after deleting E and H: " + hs);

 // Adiciona E novamente.
 hs.add('E');
 System.out.println("\nContents after adding E: " + hs);

 // Adiciona uma coleção de elementos ao conjunto hash.
 ArrayList<Character> al = new ArrayList<Character>();
 al.add('X');
 al.add('Y');
 al.add('Z');
 hs.addAll(al);

 System.out.println("\nContents after adding collection: " + hs);
}
}

```

A saída a seguir é gerada pelo programa:

```

Original contents: [D, A, B, C]

Contents after additions: [D, E, F, G, A, B, C, H]

Contents after deleting E and H: [D, F, G, A, B, C]

Contents after adding E: [D, E, F, G, A, B, C]

Contents after adding collection: [D, E, F, G, A, B, C, Y, X, Z]

```

Preste atenção na ordem dos elementos. Como explicado, **HashSet** não armazena seus elementos em uma ordem específica. Em vez disso, a ordem está associada ao código hash de cada elemento. Até mesmo quando o conteúdo de **al** (um **ArrayList**) é adicionado a **hs**, os elementos não são ordenados na mesma sequência.

Antes de sairmos desse exemplo, um ponto importante deve ser reforçado. **HashSet** (e qualquer coleção que implemente a interface **Set**) não pode conter elementos duplicados. Portanto, se você adicionasse estas linhas ao fim do programa:

```

hs.add('Q');
hs.add('Q');

```

só um Q seria adicionado ao conjunto. A segunda chamada falharia e retornaria **false**.

## A classe TreeSet

**TreeSet** estende **AbstractSet** e implementa a interface **NavigableSet**. Ela cria um conjunto que usa como armazenamento um tipo de árvore binária classificada e balanceada. Os objetos são classificados em ordem crescente. Os tempos de acesso e recuperação são bem rápidos, o que torna **TreeSet** uma ótima opção para o armazenamento de grandes quantidades de informações classificadas que tenham de ser encontradas rapidamente. **TreeSet** é uma classe genérica que tem esta declaração:

```
class TreeSet<E>
```

E especifica o tipo de objetos que o conjunto conterá.

**TreeSet** tem os construtores a seguir:

```
TreeSet()
TreeSet(Collection<? extends E> col)
TreeSet(Comparator<? super E> comp)
TreeSet(SortedSet<E> ss)
```

A primeira forma cria um conjunto de árvore vazio que será classificado em ordem crescente. A segunda constrói um conjunto de árvore contendo os elementos de *col*. A terceira cria um conjunto de árvore vazio que será classificado de acordo com o comparador especificado por *comp*. (Os comparadores serão descritos posteriormente neste capítulo.) A quarta forma constrói um conjunto de árvore contendo os elementos de *ss*.

Abaixo temos um exemplo que demonstra um **TreeSet**, comparando-o ao **HashSet**. Ele cria um array contendo caracteres em ordem não classificada. Esses elementos são então adicionados a um **HashSet** e depois a um **TreeSet**. No conjunto hash, a ordem dos elementos será indeterminada. No entanto, o **TreeSet** os classifica-rá à medida que constrói a árvore.

```
// Um TreeSet cria uma árvore classificada.

import java.util.*;

class TreeSetDemo {
 public static void main(String[] args) {
 char[] chrs = { 'V', 'J', 'L', 'E', 'T', 'Q', 'C', 'P' };

 // Cria um conjunto de árvore e um conjunto hash.
 TreeSet<Character> ts = new TreeSet<Character>();
 HashSet<Character> hs = new HashSet<Character>();

 System.out.print("Contents of chrs: ");
 for(char ch : chrs)
 System.out.print(ch + " ");

 System.out.println();

 // Primeiro, adiciona os caracteres ao conjunto hash.
 for(char ch : chrs)
 hs.add(ch);

 System.out.println("Contents of hash set: " + hs);

 // Em seguida, adiciona os caracteres ao conjunto de árvore.
 for(char ch : chrs)
 ts.add(ch);

 System.out.println("Contents of tree set: " + ts);
 }
}
```

A saída desse programa é:

```
Contents of chrs: V J L E T Q C P
Contents of hash set: [T, E, V, Q, P, C, L, J]
Contents of tree set: [C, E, J, L, P, Q, T, V]
```

Como você pode ver, os elementos do conjunto de árvore são armazenados em ordem classificada.

Já que **TreeSet** implementa a interface **NavigableSet**, você pode usar os métodos definidos por **NavigableSet** para recuperar elementos de um **TreeSet**. Por exemplo, dado o programa anterior, a instrução a seguir usa **subSet()** para obter um subconjunto de **ts** contendo os elementos entre **C** (inclusive) e **P** (exclusive). Em seguida, ela exibe o conjunto resultante:

```
| System.out.println(ts.subSet('C', 'P'));
```

A saída gerada por essa instrução é:

```
| [C, E, J, L]
```

Se quiser, faça testes com os outros métodos definidos por **NavigableSet**.

## A classe **LinkedHashSet**

A classe **LinkedHashSet** estende **HashSet** e não adiciona membros próprios. Ela é uma classe genérica que tem esta declaração:

```
class LinkedHashSet<E>
```

Aqui, **E** especifica o tipo de objetos que o conjunto conterá. Seus construtores são equivalentes aos de **HashSet**.

Como **HashSet**, **LinkedHashSet** usa uma tabela hash para armazenar elementos. No entanto, também gera uma sobreposição usando uma lista duplamente encadeada com as entradas do conjunto, encadeando-as na ordem em que foram inseridas. Isso permite a iteração pelo conjunto por ordem de inserção. Ou seja, quando percorremos um **LinkedHashSet** usando um iterador ou um laço **for** de estilo for-each, os elementos são retornados na ordem em que foram inseridos. Essa também é a ordem em que aparecem no string retornado pelo método **toString()** quando chamado em um objeto **LinkedHashSet**.

O código a seguir demonstra as diferenças entre implementações de **Set** adicionando **LinkedHashSet** ao exemplo anterior:

```
// LinkedHashSet comparada com HashSet e TreeSet.

import java.util.*;

class SetsDemo {
 public static void main(String[] args) {
 char[] chrs = { 'V', 'J', 'L', 'E', 'T', 'Q', 'C', 'P' };

 // Cria todos os três conjuntos.
 TreeSet<Character> ts = new TreeSet<Character>();
 HashSet<Character> hs = new HashSet<Character>();
```

```

LinkedHashSet<Character> lhs = new LinkedHashSet<Character>();

System.out.print("Contents of chrs: ");
for(char ch : chrs)
 System.out.print(ch + " ");

System.out.println();

// Primeiro, adiciona os caracteres ao conjunto hash.
for(char ch : chrs)
 hs.add(ch);

System.out.println("Contents of hash set: " + hs);

// Em seguida, adiciona os caracteres ao conjunto de árvore.
for(char ch : chrs)
 ts.add(ch);

System.out.println("Contents of tree set: " + ts);

// Para concluir, adiciona os caracteres ao conjunto hash encadeado.
for(char ch : chrs)
 lhs.add(ch);

System.out.println("Contents of linked hash set: " + lhs);
}
}

```

A saída é mostrada aqui:

|                              |                          |
|------------------------------|--------------------------|
| Contents of chrs:            | V J L E T Q C P          |
| Contents of hash set:        | [T, E, V, Q, P, C, L, J] |
| Contents of tree set:        | [C, E, J, L, P, Q, T, V] |
| Contents of linked hash set: | [V, J, L, E, T, Q, C, P] |

Observe que, no conjunto hash encadeado, a ordem reflete aquela em que os elementos foram adicionados.

## A classe ArrayDeque

A classe **ArrayDeque** estende **AbstractCollection** e implementa a interface **Deque**. Ela cria um array dinâmico que cresce quando necessário. Já que **ArrayDeque** implementa **Deque**, pode ser usada para dar suporte tanto a filas quanto a pilhas. **ArrayDeque** é uma classe genérica que tem esta declaração:

```
class ArrayDeque<E>
```

Aqui, **E** especifica o tipo de objetos armazenados na coleção.

**ArrayDeque** define os seguintes construtores:

```

ArrayDeque()
ArrayDeque(int capacidadeInicial)
ArrayDeque(Collection<? extends E> col)
```

A primeira forma cria uma deque vazia. Sua capacidade inicial é 16. O segundo construtor cria uma deque que tem a capacidade inicial passada em *capacidadeInicial*. O terceiro cria uma deque que é inicializada com os elementos da coleção passada em *col*. Em todos os casos, a capacidade cresce quando necessário para tratar os elementos adicionados à deque.

Como **ArrayDeque** implementa **Deque**, pode ser usada como uma fila ou uma pilha. Lembre-se, **Deque** especifica os métodos de pilha tradicionais, chamados **push()** e **pop()**. Usando esses métodos, você pode fazer um **ArrayDeque** funcionar como uma pilha. Com o uso de métodos como **offerLast()** e **pollFirst()**, um **ArrayDeque** funcionará como uma fila FIFO. O programa a seguir demonstra o uso de um **ArrayDeque** tanto como uma pilha quanto como uma fila:

```
// Demonstra ArrayDeque.
// Primeiro usa a deque como uma pilha.
// Em seguida, usa-a como uma fila FIFO.

import java.util.*;

class ArrayDequeDemo {
 public static void main(String[] args) {
 // Cria uma deque de array.
 ArrayDeque<Character> adq = new ArrayDeque<Character>();

 System.out.println("Using adq as a stack.");
 // Usa adq como uma pilha.
 System.out.print("Pushing: ");

 // insere itens na pilha
 for(char ch = 'A'; ch <= 'Z'; ch++) {
 adq.push(ch); ←
 System.out.print(ch);
 }

 System.out.println(); ← Usa adq como uma pilha.

 // agora, os extrai
 System.out.print("Popping: ");
 while(adq.peek() != null)
 System.out.print(adq.pop()); ←

 System.out.println("\n");

 System.out.println("Using adq as a FIFO queue.");
 // Usa adq como uma fila FIFO.
 System.out.print("Queueing: ");
 for(char ch = 'A'; ch <= 'Z'; ch++) {
 adq.offerLast(ch); ← Agora, usa adq como uma fila.
 System.out.print(ch);
 }

 System.out.println();
 }
}
```

```
// agora, remove-os
System.out.print("Removing: ");
while(adq.peek() != null)
 System.out.print(adq.pollFirst()); ←
}
}
```

A saída é esta:

```
Using adq as a stack.
Pushing: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Popping: ZYXWVUTSRQPONMLKJIHGFEDCBA

Using adq as a FIFO queue.
Queueing: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Removing: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Como a saída mostra, com o uso de **push()** e **pop()** obtemos o comportamento de pilha. Quando **offerLast()** e **pollFirst()** são usados, uma fila FIFO (primeiro a entrar, primeiro a sair) é implementada.

## A classe PriorityQueue

**PriorityQueue** estende **AbstractQueue** e implementa a interface **Queue**. Basicamente, ela cria o que seria uma fila classificada, com a ordem de classificação indicando a prioridade. O início da fila contém o elemento de prioridade mais alta, e o fim, o de prioridade mais baixa. **PriorityQueue** é uma classe genérica que tem esta declaração:

```
class PriorityQueue<E>
```

Aqui, E especifica o tipo de objetos armazenados na fila. Os **PriorityQueues** são dinâmicos e crescem quando necessário.

**PriorityQueue** define vários construtores. Vejamos três:

```
PriorityQueue()
PriorityQueue(int capacidadeInicial)
PriorityQueue(int capacidadeInicial, Comparator<? super E> comp)
```

A primeira forma cria uma fila vazia. Sua capacidade inicial é 11. O segundo construtor cria uma fila com a capacidade inicial passada em *capacidadeInicial*. O terceiro cria uma fila com a capacidade inicial e o comparador especificados. Outros construtores criam filas inicializadas com os elementos de uma coleção diferente.

Se nenhum comparador for especificado na construção de um **PriorityQueue**, a fila será classificada de acordo com sua ordem natural. Logo, por padrão, o início da fila terá o menor valor. No entanto, fornecendo um comparador, você pode especificar um esquema de ordenação diferente. Por exemplo, ao armazenar itens que incluem um *timestamp*, você pode priorizar a fila de tal forma que os itens mais antigos apareçam primeiro.

Uma advertência: embora você possa iterar por um **PriorityQueue** usando um iterador ou um laço **for** de estilo for-each, a ordem da iteração é desconhecida. Para usar um **PriorityQueue** apropriadamente, você deve chamar métodos como **offer()** e **poll()**, que são definidos pela interface **Queue**.

## Verificação do progresso

1. Um **ArrayList** difere de um array porque \_\_\_\_\_.
2. Um **PriorityQueue** é uma fila em que \_\_\_\_\_.
3. A classe **LinkedList** pode ser usada como uma pilha. Verdadeiro ou falso?
4. **HashSet** usa uma \_\_\_\_\_ para armazenar seus elementos.

## ACESSANDO UMA COLEÇÃO COM UM ITERADOR

Não é raro querermos percorrer os elementos de uma coleção. Por exemplo, podemos querer exibir cada elemento dela. Uma maneira de fazê-lo é empregando um **iterador**, que é um objeto que implementa a interface **Iterator** ou **ListIterator**. **Iterator** nos permite percorrer uma coleção, obtendo ou removendo elementos. **ListIterator** estende **Iterator** para permitir a passagem bidirecional por uma lista e a modificação de elementos. **Iterator** e **ListIterator** são interfaces genéricas assim declaradas:

```
interface Iterator<E>
interface ListIterator<E>
```

Aqui, **E** especifica o tipo de objetos que estão sendo percorridos.

A interface **Iterator** especifica os métodos mostrados na Tabela 25-7. Os métodos especificados por **ListIterator** são mostrados na Tabela 25-8. (Os herdados de **Iterator** também são mostrados para ficar claro seu comportamento em relação a um **ListIterator**.) Observe que tanto **Iterator** quanto **ListIterator** permitem a remoção de um elemento da coleção que está sendo percorrida com uma chamada a **remove()**. Em **ListIterator**, os métodos **add()** e **set()** permitem a alteração da coleção que está sendo percorrida. No entanto, é preciso entender que as operações que modificam a coleção subjacente são opcionais. Por exemplo, **remove()** lançará uma **UnsupportedOperationException** quando usado com uma coleção de somente leitura.

Ao usar um **Iterator**, temos que tomar cuidado com a ordem na qual chamamos **remove()**. Uma chamada a **remove()** que não for precedida por uma chamada a **next()** resultará no lançamento de uma **IllegalStateException**. O mesmo se aplica a **ListIterator**. Se **remove()** não for precedido por uma chamada a **next()** ou **previous()**, ou se a coleção que estiver sendo percorrida for alterada por uma chamada intermediária a **add()** ou **remove()**, uma **IllegalStateException** será lançada.

## Usando um iterador

Antes de poder acessar uma coleção com um iterador, você deve obter um. Todas as classes de coleções fornecem um método **iterator()** que retorna um iterador para o início da coleção. Usando esse iterador, você pode acessar cada elemento da coleção,

### Respostas:

1. usa os métodos **get()** e **set()**, entre outros, no acesso ao array em vez de usar notação de colchetes, e pode ser redimensionado dinamicamente.
2. os elementos têm prioridades e o elemento de prioridade mais alta fica no início da fila. Logo, é o próximo a ser removido.
3. Verdadeiro. Ela tem métodos **push()** e **pop()** que tornam fácil usá-la como uma pilha.
4. tabela hash

**Tabela 25-7** Métodos declarados por Iterator

| Método             | Descrição                                                                                                 |
|--------------------|-----------------------------------------------------------------------------------------------------------|
| boolean hasNext( ) | Retorna <b>true</b> se houver um próximo elemento. Caso contrário, retorna <b>false</b> .                 |
| E next( )          | Retorna o próximo elemento. Se não houver mais elementos, uma <b>NoSuchElementException</b> será lançada. |
| void remove( )     | Remove o elemento retornado por <b>next( )</b> da coleção que está sendo percorrida.                      |

**Tabela 25-8** Métodos declarados por ListIterator

| Método                 | Descrição                                                                                                                                                                                                                 |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void add(E obj)        | Adiciona <i>obj</i> à coleção em frente ao elemento que será retornado pela próxima chamada a <b>next( )</b> .                                                                                                            |
| boolean hasNext( )     | Retorna <b>true</b> se houver um próximo elemento. Caso contrário, retorna <b>false</b> . Esse método é usado quando nos movemos para a frente.                                                                           |
| boolean hasPrevious( ) | Retorna <b>true</b> se houver um elemento anterior. Caso contrário, retorna <b>false</b> . Esse método é usado quando nos movemos para trás.                                                                              |
| E next( )              | Retorna o próximo elemento na direção progressiva. Se não houver mais elementos nessa direção, uma <b>NoSuchElementException</b> será lançada.                                                                            |
| int nextIndex( )       | Retorna o índice do próximo elemento na direção progressiva. Se não houver um próximo elemento, o tamanho da coleção que está sendo percorrida será retornado.                                                            |
| E previous( )          | Retorna o elemento anterior, que é o próximo elemento na direção regressiva. Se não houver mais elementos nessa direção, uma <b>NoSuchElementException</b> será lançada.                                                  |
| int previousIndex( )   | Retorna o índice do próximo elemento na direção regressiva. Se não houver um elemento anterior, -1 será retornado.                                                                                                        |
| void remove( )         | Remove o elemento retornado por <b>next( )</b> ou <b>previous( )</b> da coleção que está sendo percorrida.                                                                                                                |
| void set(E obj)        | Atribui <i>obj</i> ao elemento que está sendo examinado atualmente, ou seja, ao último elemento retornado por uma chamada a <b>next( )</b> ou <b>previous( )</b> . A alteração afeta a coleção que está sendo percorrida. |

um de cada vez. Para usar um iterador e percorrer o conteúdo de uma coleção, siga estas etapas:

1. Obtenha um iterador para o início da coleção chamando seu método **iterator( )**.
2. Defina um laço que faça uma chamada a **hasNext( )**. Faça o laço iterar até **hasNext( )** retornar **true**.
3. Dentro do laço, obtenha cada elemento chamando **next( )**.

Para coleções que implementam **List**, também podemos obter um iterador chamando **listIterator( )**. Como explicado, um iterador de lista nos permite acessar a coleção na direção progressiva ou regressiva e alterar ou incluir um elemento na coleção que está sendo percorrida. Caso contrário, **ListIterator** é usada como **Iterator**.

O exemplo a seguir implementa essas etapas, demonstrando as interfaces **Iterator** e **ListIterator**. Ele usa um **ArrayList** de strings, mas os princípios gerais podem

ser aplicados a qualquer tipo de coleção que dê suporte a iteradores. Observe como a coleção subjacente é modificada pelo uso de **remove()**, **add()** e **set()**.

```
// Demonstra tanto Iterator quanto ListIterator.

import java.util.*;

class IteratorDemo {
 public static void main(String[] args) {
 // Cria uma lista de strings.
 ArrayList<String> al = new ArrayList<String>();

 // Adiciona entradas à lista de array.
 al.add("Alpha");
 al.add("Beta");
 al.add("Gamma");
 al.add("Delta");
 al.add("Epsilon");
 al.add("Zeta");
 al.add("Eta");

 // Primeiro, usa Iterator.

 // Usa Iterator para exibir o conteúdo da lista.
 System.out.print("Original contents: ");
 Iterator<String> itr = al.iterator(); ← Obtém um iterador.
 while(itr.hasNext()) ← Procura um próximo elemento.
 System.out.print(itr.next() + " "); ← Obtém o próximo elemento.
 System.out.println();

 // Usa Iterator para remover Gamma da lista.
 itr = al.iterator();
 while(itr.hasNext()) {
 if(itr.next().equals("Gamma"))
 itr.remove(); ← Usa um iterador para remover um elemento.
 }

 System.out.print("Contents after deletion: ");
 itr = al.iterator();
 while(itr.hasNext())
 System.out.print(itr.next() + " ");
 System.out.println();

 // Agora, usa ListIterator.

 // Usa ListIterator para adicionar Gamma novamente à lista.
 ListIterator<String> litr = al.listIterator(); ← Obtém um iterador de lista.
 while(litr.hasNext()) {
 if(litr.next().equals("Beta"))
 litr.add("Gamma"); ← Adiciona um elemento usando o iterador de lista.
 }
 }
}
```

```

}

System.out.print("Contents after addition: ");
litr = al.listIterator();
while(litr.hasNext())
 System.out.print(litr.next() + " ");
System.out.println();

// Usa ListIterator para modificar os objetos que estão sendo percorridos.
String str;
litr = al.listIterator();
while(litr.hasNext()) {
 str = litr.next();

 if(str.equals("Eta"))
 litr.set("Omega"); <———— Usa um iterador de lista para alterar um elemento.
 else if(str.equals("Zeta"))
 litr.set("Psi");
 else if(str.equals("Epsilon"))
 litr.set("Chi");
 else if(str.equals("Delta"))
 litr.set("...");
}

System.out.print("Contents after changes: ");
litr = al.listIterator();
while(litr.hasNext())
 System.out.print(litr.next() + " ");
System.out.println();

// Usa ListIterator para exibir a lista de trás para frente.
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
 System.out.print(litr.previous() + " "); <———— Exibe a lista na ordem inversa.
}
System.out.println();
}
}

```

A saída é mostrada abaixo:

|                          |                                         |
|--------------------------|-----------------------------------------|
| Original contents:       | Alpha Beta Gamma Delta Epsilon Zeta Eta |
| Contents after deletion: | Alpha Beta Delta Epsilon Zeta Eta       |
| Contents after addition: | Alpha Beta Gamma Delta Epsilon Zeta Eta |
| Contents after changes:  | Alpha Beta Gamma ... Chi Psi Omega      |
| Modified list backwards: | Omega Psi Chi ... Gamma Beta Alpha      |

No fim do programa, preste atenção em como a lista é exibida na ordem inversa. Após ela ser exibida na direção progressiva, **litr** aponta para o fim da lista. (Lembre-se, **litr.hasNext()** retorna **false** quando o fim da lista é alcançado.) Para percorrer a lista ao contrário, o programa continua a usar **litr**, mas dessa vez verifica se ela tem um próximo elemento. Se tiver, esse elemento será obtido e exibido.

## A alternativa aos iteradores com o uso de for-each

Quando não há a intenção de modificar o conteúdo de uma coleção ou de obter elementos na ordem inversa, a versão for-each do laço **for** costuma ser uma alternativa mais conveniente para se percorrer uma coleção do que o uso de um iterador. Lembre-se de que o laço **for** de estilo for-each pode percorrer qualquer coleção de objetos que implemente a interface **Iterable**. Já que todas as classes de coleções implementam essa interface, elas podem ser tratadas por for-each.

Por exemplo, dado o **ArrayList** chamado de **al** do exemplo anterior, o laço **for** a seguir exibe seu conteúdo:

```
for(String s : al)
 System.out.println(s);
```

Como você pode ver, o laço **for** é significativamente mais curto e fácil de usar do que a abordagem baseada em iterador. No entanto, ele só pode ser usado na passagem por uma coleção na direção progressiva, e o conteúdo da coleção não pode ser modificado.

### Verificação do progresso

1. Os dois métodos da interface **Iterator** que são usados na passagem pelos elementos de um objeto **Collection** são \_\_\_\_\_ e \_\_\_\_\_.
2. Para coleções que implementam **List**, um **ListIterator** é um **Iterator** que nos permite percorrer progressiva e regressivamente a coleção e alterar seus valores. Verdadeiro ou falso?

## TRABALHANDO COM MAPAS

No que diz respeito ao Collections Framework, um *mapa* armazena associações entre chaves e valores, ou *pares chave/valor*. Dada uma chave, você pode encontrar seu valor. Há um aspecto dos mapas que é importante mencionar desde o início: eles não implementam a interface **Iterable**. Ou seja, você *não pode* percorrer um mapa usando um laço **for** de estilo for-each. Além disso, não pode obter um iterador para um mapa. No entanto, como veremos em breve, pode obter a visão de coleção do mapa, e essa sim permite o uso do laço **for** ou de um iterador.

### As interfaces de mapas

Já que as interfaces de mapas definem o caráter e a natureza do mapa, esta discussão sobre mapas começará com elas. As interfaces de **java.util** mostradas a seguir dão suporte a mapas:

Respostas:

1. **hasNext()** e **next()**
2. Verdadeiro.

| Interface    | Descrição                                                                                                          |
|--------------|--------------------------------------------------------------------------------------------------------------------|
| Map          | Mapeia chaves exclusivas para valores.                                                                             |
| Map.Entry    | Encapsula um elemento (um par chave/valor) em um mapa.                                                             |
| NavigableMap | Estende <b>SortedMap</b> para tratar a recuperação de entradas com base em quanto uma chave está próxima de outra. |
| SortedMap    | Estende <b>Map</b> de tal modo que as chaves são mantidas em ordem crescente.                                      |

Examinaremos cada uma.

### A interface Map

A interface **Map** mapeia chaves exclusivas para valores. Uma chave é um objeto que usamos para recuperar um valor. Dada uma chave e um valor, podemos armazenar o valor em um **Map**. Após o valor ser armazenado, podemos recuperá-lo usando sua chave. **Map** é genérica e declarada assim:

```
interface Map<K, V>
```

Aqui, **K** especifica o tipo das chaves, e **V**, o tipo dos valores. Os métodos declarados por **Map** estão resumidos na Tabela 25-9.

Duas operações básicas são os principais elementos de **Map**: **get( )** e **put( )**. Para inserir um valor em um mapa, use **put( )**, especificando a chave e o valor. Para obter um valor, chame **get( )**, passando a chave como argumento. O valor será retornado. Esses dois métodos atendem aos aspectos básicos da natureza dos mapas: armazenar um par chave/valor e retornar o valor quando dada a chave.

Como mencionado anteriormente, embora os mapas façam parte do Collections Framework, eles não implementam a interface **Collection**. No entanto, você pode obter a visão de coleção do mapa. Para fazê-lo, pode usar o método **entrySet( )**. Ele retorna um objeto **Set** contendo os elementos do mapa. Para obter a visão de coleção das chaves, use **keySet( )**, e para obter a visão de coleção dos valores, use **values( )**. Todas as três visões de coleção referenciam os elementos do mapa original. Se um for alterado, o outro é afetado. Contudo, você não pode adicionar uma entrada ao mapa por intermédio da visão. As visões de coleção são o meio pelo qual os mapas são integrados ao corpo maior do Collections Framework.

Para concluir, observe os métodos **containsKey( )** e **containsValue( )**. Eles permitem a procura de uma chave ou de um valor específico, respectivamente, em um mapa.

### A interface SortedMap

A interface **SortedMap** estende **Map**. Ela armazena entradas em ordem crescente com base nas chaves. **SortedMap** é genérica e declarada assim:

```
interface SortedMap<K, V>
```

Aqui, **K** especifica o tipo das chaves, e **V**, o tipo dos valores. Os métodos adicionados por **SortedMap** estão resumidos na Tabela 25-10.

Observe que **SortedMap** fornece métodos que permitem o trabalho com *submapas* (em outras palavras, subconjuntos de um mapa). Para obter um submapa, use **headMap( )**, **tailMap( )** ou **subMap( )**. Já que o mapa é classificado, esses métodos

**Tabela 25-9** Métodos declarados por Map

| Método                                                     | Descrição                                                                                                                                                                                                                                         |
|------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void clear()                                               | Esvazia o mapa chamador excluindo todos os pares chave/valor.                                                                                                                                                                                     |
| boolean containsKey(Object <i>k</i> )                      | Retorna <b>true</b> se o mapa chamador tiver <i>k</i> como chave. Caso contrário, retorna <b>false</b> .                                                                                                                                          |
| boolean containsValue(Object <i>v</i> )                    | Retorna <b>true</b> se o mapa chamador tiver <i>v</i> como valor. Caso contrário, retorna <b>false</b> .                                                                                                                                          |
| Set<Map.Entry<K, V>> entrySet()                            | Retorna um <b>Set</b> contendo todas as entradas do mapa chamador como objetos de tipo <b>Map.Entry</b> . A coleção resultante é uma visão do mapa chamador.                                                                                      |
| boolean equals(Object <i>obj</i> )                         | Retorna <b>true</b> se <i>obj</i> for um <b>Map</b> e tiver as mesmas entradas do <b>Map</b> chamador. Caso contrário, retorna <b>false</b> .                                                                                                     |
| V get( Object <i>k</i> )                                   | Retorna o valor associado à chave <i>k</i> no mapa chamador. Retorna <b>null</b> se a chave não for encontrada.                                                                                                                                   |
| int hashCode()                                             | Retorna o código hash do mapa chamador.                                                                                                                                                                                                           |
| boolean isEmpty()                                          | Retorna <b>true</b> se o mapa chamador estiver vazio. Caso contrário, retorna <b>false</b> .                                                                                                                                                      |
| Set<K> keySet()                                            | Retorna um <b>Set</b> contendo todas as chaves do mapa chamador. A coleção resultante é uma visão do mapa chamador.                                                                                                                               |
| V put(K <i>k</i> , V <i>v</i> )                            | Insere uma entrada no mapa chamador, sobrepondo qualquer valor anterior associado à chave. A chave e o valor são <i>k</i> e <i>v</i> , respectivamente. Retorna <b>null</b> se a chave não existir. Caso contrário, o valor anterior é retornado. |
| void putAll(Map<? extends K,<br>? extends V> <i>mapa</i> ) | Insere todas as entradas de <i>mapa</i> no mapa chamador.                                                                                                                                                                                         |
| V remove(Object <i>k</i> )                                 | Remove a entrada cuja chave é <i>k</i> no mapa chamador. O valor do elemento removido é retornado.                                                                                                                                                |
| int size()                                                 | Retorna o número de entradas do mapa chamador.                                                                                                                                                                                                    |
| Collection<V> values()                                     | Retorna uma coleção contendo todos os valores do mapa. A coleção resultante é uma visão do mapa chamador.                                                                                                                                         |

permitem a especificação do intervalo de entradas a ser obtido com base nos valores das chaves passadas. Cuidado, porque os submapas retornados são “visões” e estão sujeitos a várias restrições.

Observe também que você pode obter a primeira chave do mapa chamando **firstKey()**. Para obter a última chave, chame **lastKey()**. Já que o mapa é classificado, estamos nos referindo, respectivamente, à menor e à maior chave do mapa.

A interface **NavigableMap**

A interface **NavigableMap** estende **SortedMap**. Ela declara o comportamento de um mapa que dá suporte à recuperação de uma entrada com base em quanto uma chave está próxima de outra. Em outras palavras, um **NavigableMap** nos permite encontrar

**Tabela 25-10** Métodos declarados por SortedMap

| Método                                                  | Descrição                                                                                                                                                                              |
|---------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Comparator<? super K> comparator( )                     | Retorna o comparador do mapa classificado chamador. Se nenhum comparador estiver sendo usado, <b>null</b> é retornado.                                                                 |
| K firstKey( )                                           | Retorna a chave do menor valor. Já que o mapa é classificado, ela será a primeira chave do mapa chamador. Se chamado em um mapa vazio, uma <b>NoSuchElementException</b> será lançada. |
| SortedMap<K, V> headMap(K <i>fim</i> )                  | Retorna um <b>SortedMap</b> contendo os elementos do mapa chamador com chaves menores do que <i>fim</i> . O mapa resultante é uma visão do mapa chamador.                              |
| K lastKey( )                                            | Retorna a chave do maior valor. Já que o mapa é classificado, ela será a última chave do mapa chamador. Se chamado em um mapa vazio, uma <b>NoSuchElementException</b> será lançada.   |
| SortedMap<K, V> subMap(K <i>início</i> , K <i>fim</i> ) | Retorna um <b>SortedMap</b> contendo os elementos do mapa chamador que estejam entre <i>início</i> e <i>fim</i> – 1. O mapa resultante é uma visão do mapa chamador.                   |
| SortedMap<K, V> tailMap(K <i>início</i> )               | Retorna um <b>SortedMap</b> contendo os elementos do mapa chamador maiores ou iguais a <i>início</i> . O mapa resultante é uma visão do mapa chamador.                                 |

uma entrada com base em ocorrências próximas. **NavigableMap** é uma interface genérica que tem esta declaração:

```
interface NavigableMap<K,V>
```

Aqui, **K** especifica o tipo das chaves, e **V**, o tipo dos valores associados a elas. Além dos métodos herdados de **SortedMap**, **NavigableMap** adiciona os resumidos na Tabela 25-11.

Preste atenção nos métodos **ceilingX** e **floorX**. Eles procuram no mapa uma chave que esteja próxima de outra. Por exemplo, os métodos **ceilingEntry( )** e **floorEntry( )** encontram uma entrada cuja chave esteja próxima ou seja igual a outra chave. Observe também os métodos **higherX** e **lowerX**. Eles procuram no mapa uma chave que esteja próxima de outra, mas que não seja igual a ela. Você também pode obter vários submapas baseados em ocorrências próximas.

### A interface Map.Entry

A interface **Map.Entry** nos permite trabalhar com uma entrada do mapa. Lembre-se, o método **entrySet( )** declarado pela interface **Map** retorna um **Set** contendo as entradas do mapa. Cada um desses elementos do conjunto é um objeto **Map.Entry**. **Map.Entry** é genérica e declarada desta forma:

```
interface Map.Entry<K, V>
```

Aqui, **K** especifica o tipo das chaves, e **V**, o tipo dos valores.

**Tabela 25-11** Métodos declarados por NavigableMap

| Método                                                                       | Descrição                                                                                                                                                                                                                                                      |
|------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Map.Entry<K,V> ceilingEntry(K <i>obj</i> )                                   | Procura no mapa a menor chave <i>k</i> de modo que $k \geq obj$ . Se a chave for encontrada, sua entrada será retornada. Caso contrário, <b>null</b> será retornado.                                                                                           |
| K ceilingKey(K <i>obj</i> )                                                  | Procura no mapa a menor chave <i>k</i> de modo que $k \geq obj$ . Se a chave for encontrada, ela será retornada. Caso contrário, <b>null</b> será retornado.                                                                                                   |
| NavigableSet<K> descendingKeySet( )                                          | Retorna um <b>NavigableSet</b> contendo todas as chaves do mapa chamador em ordem inversa. A coleção resultante é uma visão do mapa chamador.                                                                                                                  |
| NavigableMap<K,V> descendingMap( )                                           | Retorna um <b>NavigableMap</b> contendo as entradas do mapa chamador em ordem inversa. A coleção resultante é uma visão do mapa chamador.                                                                                                                      |
| Map.Entry<K,V> firstEntry( )                                                 | Retorna a primeira entrada. Já que o mapa é classificado, ela será a entrada do menor valor.                                                                                                                                                                   |
| Map.Entry<K,V> floorEntry(K <i>obj</i> )                                     | Procura no mapa a maior chave <i>k</i> de modo que $k \leq obj$ . Se a chave for encontrada, sua entrada será retornada. Caso contrário, <b>null</b> será retornado.                                                                                           |
| K floorKey(K <i>obj</i> )                                                    | Procura no mapa a maior chave <i>k</i> de modo que $k \leq obj$ . Se a chave for encontrada, ela será retornada. Caso contrário, <b>null</b> será retornado.                                                                                                   |
| NavigableMap<K,V><br>headMap(K <i>limiteSuperior</i> , boolean <i>incl</i> ) | Retorna um <b>NavigableMap</b> contendo os elementos do mapa chamador com chaves menores que <i>limiteSuperior</i> . Se <i>incl</i> for <b>true</b> , um elemento igual a <i>limiteSuperior</i> será incluído. O mapa resultante é uma visão do mapa chamador. |
| Map.Entry<K,V> higherEntry(K <i>obj</i> )                                    | Procura no mapa a menor chave <i>k</i> de modo que $k > obj$ . Se a chave for encontrada, sua entrada será retornada. Caso contrário, <b>null</b> será retornado.                                                                                              |
| K higherKey(K <i>obj</i> )                                                   | Procura no conjunto a menor chave <i>k</i> de modo que $k > obj$ . Se a chave for encontrada, ela será retornada. Caso contrário, <b>null</b> será retornado.                                                                                                  |
| Map.Entry<K,V> lastEntry( )                                                  | Retorna a última entrada. Já que o mapa é classificado, ela será a entrada do maior valor.                                                                                                                                                                     |
| Map.Entry<K,V> lowerEntry(K <i>obj</i> )                                     | Procura no conjunto a maior chave <i>k</i> de modo que $k < obj$ . Se a chave for encontrada, sua entrada será retornada. Caso contrário, <b>null</b> será retornado.                                                                                          |
| K lowerKey(K <i>obj</i> )                                                    | Procura no conjunto a maior chave <i>k</i> de modo que $k < obj$ . Se a chave for encontrada, ela será retornada. Caso contrário, <b>null</b> será retornado.                                                                                                  |
| NavigableSet<K> navigableKeySet( )                                           | Retorna um <b>NavigableSet</b> contendo todas as chaves do mapa chamador. O conjunto resultante é uma visão do mapa chamador.                                                                                                                                  |

**Tabela 25-11** Métodos declarados por NavigableMap (continuação)

| Método                                                                                                                                               | Descrição                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Map.Entry<K,V> pollFirstEntry()                                                                                                                      | Retorna a primeira entrada, removendo-a no processo. Já que o mapa é classificado, ela será a entrada de chave com menor valor. Se o mapa estiver vazio, <b>null</b> será retornado.                                                                                                                                                                                                                            |
| Map.Entry<K,V> pollLastEntry()                                                                                                                       | Retorna a última entrada, removendo-a no processo. Já que o mapa é classificado, ela será a entrada de chave com maior valor. Se o mapa estiver vazio, <b>null</b> será retornado.                                                                                                                                                                                                                              |
| NavigableMap<K,V><br>subMap(K <i>limiteInferior</i> ,<br>boolean <i>inclInferior</i> ,<br>K <i>limiteSuperior</i> ,<br>boolean <i>inclSuperior</i> ) | Retorna um <b>NavigableMap</b> contendo todas as entradas do mapa chamador com chaves maiores que <i>limiteInferior</i> e menores que <i>limiteSuperior</i> . Se <i>inclInferior</i> for <b>true</b> , uma chave igual a <i>limiteInferior</i> será incluída. Se <i>inclSuperior</i> for <b>true</b> , um elemento igual a <i>limiteSuperior</i> será incluído. O mapa resultante é uma visão do mapa chamador. |
| NavigableMap<K,V><br>tailMap(K <i>limiteInferior</i> , boolean <i>incl</i> )                                                                         | Retorna um <b>NavigableMap</b> contendo todas as entradas do mapa chamador com chaves maiores que <i>limiteInferior</i> . Se <i>incl</i> for <b>true</b> , uma chave igual à <i>limiteInferior</i> será incluída. O mapa resultante é uma visão do mapa chamador.                                                                                                                                               |

Além das sobreposições de **equals( )** e **hashCode( )**, **Map.Entry** especifica três métodos que dão acesso a uma entrada do mapa. Eles são:

```
K getKey()
V getValue()
V setValue(V v)
```

Como o nome sugere, para qualquer instância de **Map.Entry**, **getKey( )** retorna sua chave e **getValue( )** retorna seu valor. Você pode definir o valor chamando **setValue( )**, passando o novo valor.

## As classes de mapas

Várias classes fornecem implementações das interfaces de mapas. A classe abstrata **AbstractMap** implementa grande parte da interface **Map**. Ela é estendida por várias classes concretas de mapas. Examinaremos as três a seguir:

| Classe        | Descrição                                                             |
|---------------|-----------------------------------------------------------------------|
| HashMap       | Estende <b>AbstractMap</b> para usar uma tabela hash.                 |
| TreeMap       | Estende <b>AbstractMap</b> para usar uma árvore.                      |
| LinkedHashMap | Estende <b>HashMap</b> para permitir iterações por ordem de inserção. |

**Nota:** Outras classes de mapas são **WeakHashMap**, **IdentityHashMap** e **EnumMap**. Seu uso é especializado e elas não serão examinadas aqui.

A classe **HashMap**

A classe **HashMap** estende **AbstractMap** e implementa a interface **Map**. Ela usa uma tabela hash para armazenar o mapa. Isso permite que o tempo de execução de **get( )** e **put( )** permaneça constante até mesmo para grandes conjuntos de dados. **HashMap** é uma classe genérica que tem esta declaração:

```
class HashMap<K, V>
```

Aqui, **K** especifica o tipo das chaves, e **V**, o tipo dos valores.

Os construtores a seguir são definidos:

```
HashMap()
HashMap(Map<? extends K, ? extends V> mapa)
HashMap(int capacidadeInicial)
HashMap(int capacidadeInicial, float taxaPreenchimento)
```

O primeiro construtor cria um mapa hash vazio. O segundo inicializa o mapa hash usando os elementos de *mapa*. O terceiro inicializa a capacidade inicial do mapa hash com *capacidadeInicial*. O quarto inicializa tanto a capacidade inicial quanto a taxa de preenchimento (fator de carga) do mapa hash usando seus argumentos. O significado de capacidade e taxa de preenchimento é o mesmo usado na classe **HashSet**, descrita anteriormente. A capacidade inicial padrão é 16. A taxa de preenchimento padrão é 0,75.

É bom ressaltar que um mapa hash *não* armazena elementos em uma ordem específica. Além disso, a ordem pode mudar durante o tempo de vida do mapa. Portanto, a ordem em que os elementos são adicionados a um mapa hash não é necessariamente aquela em que serão obtidos se percorrermos uma visão de coleção do mapa.

O programa a seguir ilustra **HashMap**. Ele mapeia nomes para saldos de conta. Observe como uma visão de coleção é obtida e usada:

```
import java.util.*;

class HashMapDemo {
 public static void main(String[] args) {

 // Cria um mapa hash.
 HashMap<String, Double> hm = new HashMap<String, Double>();

 // Insere elementos no mapa.
 hm.put("John Doe", 3434.34);
 hm.put("Tom Smith", 123.22);
 hm.put("Jane Baker", 1378.00);
 hm.put("Todd Hall", 99.22);
 hm.put("Ralph Smith", -19.08); }

 // Obtém um conjunto das entradas.
 Set<Map.Entry<String, Double>> set = hm.entrySet(); ←————— Obtém uma visão
 // Exibe o conjunto. de conjunto das
 for(Map.Entry<String, Double> me : set) { entradas.
```

```

 System.out.print(me.getKey() + " : ");
 System.out.println(me.getValue()); ← Exibe as chaves
 }
 System.out.println();

 // Deposita 1000 na conta de John Doe.
 double balance = hm.get("John Doe");
 hm.put("John Doe", balance + 1000);

 System.out.println("John Doe's new balance: " +
 hm.get("John Doe"));
}
}

```

A saída desse programa é mostrada aqui (a ordem exata pode variar):

```

Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Todd Hall: 99.22
Jane Baker: 1378.0

John Doe's new balance: 4434.34

```

O programa começa criando um mapa hash e então adiciona o mapeamento entre nomes e saldos chamando **put()**. Em seguida, o conteúdo do mapa é exibido com o uso de uma visão de coleção, obtida com a chamada a **entrySet()**. As chaves e valores são exibidos com uma chamada aos métodos **getKey()** e **getValue()** definidos por **Map.Entry**. Preste atenção em como o depósito é feito na conta de John Doe. Primeiro, o saldo atual é obtido com uma chamada a **get()**. Depois, **put()** é chamado com o novo saldo. Isso substitui automaticamente qualquer valor pré-existente associado à chave específica pelo novo valor. Logo, após a conta de John Doe ser atualizada, o mapa hash continuará contendo apenas uma conta “John Doe”.

### A classe TreeMap

A classe **TreeMap** estende **AbstractMap** e implementa a interface **NavigableMap**. Ela cria mapas armazenados em uma estrutura de árvore. Um **TreeMap** fornece um meio eficaz de armazenar pares chave/valor em ordem classificada e permite a recuperação rápida. É bom ressaltar que, diferentemente de um mapa hash, um mapa de árvore garante que seus elementos serão classificados por ordem crescente das chaves. **TreeMap** é uma classe genérica que tem esta declaração:

```
class TreeMap<K, V>
```

Aqui, **K** especifica o tipo das chaves, e **V**, o tipo dos valores.

**TreeMap** define os construtores a seguir:

```

TreeMap()
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> mapa)
TreeMap(SortedMap<K, ? extends V> mapa)

```

O primeiro construtor cria um mapa de árvore vazio. O segundo constrói um mapa de árvore que será classificado com o uso do **Comparator** especificado por *comp*. O terceiro construtor inicializa um mapa de árvore com as entradas de *mapa*, em ordem classificada. O quarto construtor inicializa um mapa de árvore com as entradas de *mapa*, que serão classificadas na mesma ordem de *mapa*.

O programa abaixo recria o exemplo anterior com o uso de **TreeMap**:

```
import java.util.*;

class TreeMapDemo {
 public static void main(String[] args) {

 // Cria um mapa de árvore.
 TreeMap<String, Double> tm = new TreeMap<String, Double>();

 // Insere elementos no mapa.
 tm.put("John Doe", 3434.34);
 tm.put("Tom Smith", 123.22);
 tm.put("Jane Baker", 1378.00);
 tm.put("Todd Hall", 99.22);
 tm.put("Ralph Smith", -19.08);

 // Obtém um conjunto das entradas.
 Set<Map.Entry<String, Double>> set = tm.entrySet();

 // Exibe os elementos.
 for(Map.Entry<String, Double> me : set) {
 System.out.print(me.getKey() + ": ");
 System.out.println(me.getValue());
 }
 System.out.println();

 // Deposita 1000 na conta de John Doe.
 double balance = tm.get("John Doe");
 tm.put("John Doe", balance + 1000);

 System.out.println("John Doe's new balance: " +
 tm.get("John Doe"));
 }
}
```

A saída é mostrada a seguir:

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22

John Doe's current balance: 4434.34
```

Observe que **TreeMap** classifica as entradas com base nas chaves. No entanto, nesse caso, elas são classificadas começando com o prenome e não com o sobrenome. Você pode alterar esse comportamento especificando um comparador quando o mapa for criado, como descrito resumidamente.

A classe **LinkedHashMap**

**LinkedHashMap** estende **HashMap**. Ela é semelhante a **HashMap** porque suas entradas são armazenadas em uma tabela hash. No entanto, também sobrepõe as entradas com uma lista duplamente encadeada. A menos que especificado diferentemente, a ordem das entradas da lista é a mesma daquela em que foram inseridas. Isso nos permite iterar pelo mapa por ordem de inserção. Ou seja, na iteração pela visão de coleção de um **LinkedHashMap**, os elementos serão retornados na ordem em que foram inseridos. Também podemos criar um **LinkedHashMap** que retorne seus elementos na ordem da última vez em que foram acessados.

**LinkedHashMap** é uma classe genérica que tem esta declaração:

```
class LinkedHashMap<K, V>
```

Aqui, **K** especifica o tipo das chaves, e **V**, o tipo dos valores.

**LinkedHashMap** define os construtores a seguir:

```
LinkedHashMap()
LinkedHashMap(Map<? extends K, ? extends V> mapa)
LinkedHashMap(int capacidadeInicial)
LinkedHashMap(int capacidadeInicial, float taxaPreenchimento)
LinkedHashMap(int capacidadeInicial, float taxaPreenchimento, boolean ordemPorAcesso)
```

O primeiro construtor cria um mapa vazio. O segundo constrói um mapa inicializado com os elementos de *mapa*. O terceiro construtor inicializa a capacidade inicial. O quarto inicializa tanto a capacidade inicial quanto a taxa de preenchimento (fator de carga). O significado de capacidade e taxa de preenchimento é o mesmo usado em **HashMap**. A capacidade inicial padrão é 16. A taxa de preenchimento padrão é 0,75. O último construtor nos permite especificar se os elementos serão armazenados na lista encadeada por ordem de inserção ou por ordem do último acesso. Se *ordemPorAcesso* for **true**, a ordem por acesso será usada. Se *ordemPorAcesso* for **false**, a ordem de inserção será usada. Quando a ordem por acesso é usada, o último elemento da lista é o usado mais recentemente.

**LinkedHashMap** só adiciona um método aos definidos por **HashMap**. Trata-se do método **removeEldestEntry()**, mostrado abaixo:

```
protected boolean removeEldestEntry(Map.Entry<K, V> entrada)
```

Esse método é chamado por **put()** e **putAll()**. A entrada mais antiga é passada em *entrada*. Por padrão, o método retorna **false** e não faz nada. No entanto, se você o sobrepor, poderá forçar o **LinkedHashMap** a remover a entrada mais antiga do mapa. Para que isso ocorra, a sobreposição deve retornar **true**. Para manter a entrada mais antiga, retorne **false**.

## Verificação do progresso

1. Um **Map** sempre mapeia chaves exclusivas para valores exclusivos. Verdadeiro ou falso?
2. Que método você usaria em um **Map** para recuperar o valor associado a uma chave específica? E que método usaria para adicionar um novo par chave/valor a um **Map**?
3. Para obter um objeto **Collection** com as chaves de um **Map**, use o método \_\_\_\_\_, e para obter um objeto **Collection** com os valores de um **Map**, use \_\_\_\_\_.
4. O que torna um **LinkedHashMap** único?

## COMPARADORES

Tanto **TreeSet** quanto **TreeMap** armazenam elementos em ordem classificada. No entanto, o modo como os elementos são comparados é que define o que significa exatamente “ordem classificada”. Por padrão, essas classes armazenam seus elementos usando o que em Java é conhecido como “ordem natural”. Trata-se da ordem definida pela maneira como uma classe implementa o método **compareTo()** especificado por **Comparable**. Com frequência, ela é a ordem que queremos e esperamos (A antes de B, 1 antes de 2 e assim por diante). Para ordenar os elementos de uma maneira diferente, temos que especificar um **Comparator** ao construir o conjunto ou mapa. Isso nos permite controlar com precisão como os elementos serão armazenados dentro de coleções e mapas classificados. Também nos permite armazenar um objeto em uma coleção que não implemente a interface **Comparable**.

**Comparator** é uma interface genérica que tem esta declaração:

```
interface Comparator<T>
```

Aqui, **T** especifica o tipo de objetos que estão sendo comparados.

A interface **Comparator** define dois métodos: **compare()** e **equals()**. O método **compare()**, mostrado abaixo, compara dois elementos em relação à ordem:

```
int compare(T obj1, T obj2)
```

Aqui, *obj1* e *obj2* são os objetos a serem comparados. Esse método retorna zero se os objetos forem iguais. Ele retorna um valor positivo se *obj1* for maior do que *obj2*. Caso contrário, um valor negativo é retornado. Implementando **compare()**, podemos alterar a maneira como os objetos são ordenados. Por exemplo, para classificar em ordem inversa, é só criar um comparador que inverta o resultado de uma comparação.

---

Respostas:

1. Falso. Todas as chaves devem ser exclusivas, mas não os valores. Na verdade, todas as chaves poderiam ser mapeadas para o mesmo valor.
2. **get()** e **put()**
3. **keySet()** e **values()**
4. **LinkedHashMap** sobrepõe as entradas com uma lista duplamente encadeada, permitindo que elas sejam acessadas em uma ordem conhecida.

O método **equals( )**, mostrado a seguir, verifica se um objeto é igual ao comparador que o chamou:

```
boolean equals(Object obj)
```

Aqui, *obj* é o objeto a ser testado quanto à igualdade. O método retorna **true** se tanto *obj* quanto o objeto chamador forem objetos **Comparator** e usarem a mesma ordem. Caso contrário, ele retorna **false**. Em muitos casos, podemos simplesmente usar a versão padrão de **equals( )** fornecida por **Object**. É isso que os exemplos a seguir farão.

Antes de examinarmos um exemplo que use um **Comparator**, é preciso esclarecer um ponto importante. Como regra geral, o resultado da comparação de dois objetos com **compare( )** deve ser compatível com o resultado da comparação desses dois objetos com **equals( )**. Portanto, se os dois objetos forem iguais de acordo com **equals( )**, **compareTo( )** deve retornar zero e vice-versa. Não obedecer a essa regra pode resultar em uma coleção adulterada, por exemplo.

*Nota: Em casos em que uma coleção será serializada, uma classe que implemente **Comparator** também deve implementar a interface **Serializable**. Como os exemplos a seguir não envolvem serialização, o comparador não implementa **Serializable**. Os detalhes referentes à implementação de **Serializable** não fazem parte do escopo deste livro.*

O programa abaixo demonstra o uso de um comparador. Ele é uma versão atualizada do programa **TreeMap** mostrado anteriormente que armazena saldos de conta. Na versão anterior, as contas eram classificadas por nome, mas a classificação começava com o prenome. O programa a seguir classifica as contas por sobrenome. Para fazer isso, ele usa um comparador que compara o sobrenome de cada conta. Isso resulta no mapa sendo classificado por sobrenome em vez de pelo prenome.

```
// Usa um comparador para classificar contas por sobrenome.
import java.util.*;

// Compara as últimas palavras inteiras de dois strings.
class NameComp implements Comparator<String> { ← Implementa um comparador.
 public int compare(String aStr, String bStr) {
 int i, j, k;

 // Encontra o índice do começo do sobrenome.
 i = aStr.lastIndexOf(' ');
 j = bStr.lastIndexOf(' ');

 k = aStr.substring(i).compareTo(bStr.substring(j));
 if(k==0) // os sobrenomes coincidem, verifica o nome inteiro
 return aStr.compareTo(bStr);
 else
 return k;
 }

 // Não é preciso sobrepor equals.
}
```

```

class TreeMapDemo2 {
 public static void main(String[] args) {
 // Cria um mapa de árvore que usa o comparador especificado.
 TreeMap<String, Double> tm = new TreeMap<String,
 Double>(new NameComp()); ←

 // Insere elementos no mapa.
 tm.put("John Doe", 3434.34);
 tm.put("Tom Smith", 123.22);
 tm.put("Jane Baker", 1378.00);
 tm.put("Todd Hall", 99.22);
 tm.put("Ralph Smith", -19.08);

 // Obtém um conjunto das entradas.
 Set<Map.Entry<String, Double>> set = tm.entrySet();

 // Exibe os elementos.
 for(Map.Entry<String, Double> me : set) {
 System.out.print(me.getKey() + ": ");
 System.out.println(me.getValue());
 }
 System.out.println();

 // Deposita 1000 na conta de John Doe.
 double balance = tm.get("John Doe");
 tm.put("John Doe", balance + 1000);

 System.out.println("John Doe's new balance: " +
 tm.get("John Doe"));
 }
}

```

Especifica um comparador ao construir um **TreeMap**.

Esta é a saída; observe que agora as contas estão classificadas por sobrenome:

```

Jane Baker: 1378.0
John Doe: 3434.34
Todd Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22

John Doe's new balance: 4434.34

```

A classe comparadora **TComp** compara dois strings que contêm prenomes e sobrenomes. Ela faz comparando os sobrenomes. Para tanto, encontra o índice do último espaço de cada string e então compara os substrings de cada elemento que começa nesse ponto. Em casos em que os sobrenomes são equivalentes, o string inteiro é comparado. É gerado um mapa de árvore que é classificado por sobrenome e dentro do sobrenome pelo prenome. Podemos ver isso porque Ralph Smith vem antes de Tom Smith na saída.

## OS ALGORITMOS DE COLEÇÕES

O Collections Framework define muitos algoritmos que podem ser aplicados a coleções e mapas. Esses algoritmos estão definidos como métodos **static** dentro da classe **Collections**. Embora não seja viável examinar todos, mostraremos cinco exemplos que darão uma ideia do que há disponível. Eles são **sort()**, **binarySearch()**, **replaceAll()**, **reverse()** e **rotate()**.

Talvez os dois algoritmos mais importantes de **Collection** sejam **sort()** e **binarySearch()**. O algoritmo **sort()** pode classificar qualquer coleção que implemente a interface **List**. Esta é uma de suas formas:

```
static <T extends Comparable<? super T>> void sort(List<T> lista)
```

Ela classifica os elementos de *lista* como determinado por sua ordem natural. Lembre-se, essa é a ordem determinada pelo método **compareTo()** de **Comparable**, que *T* deve implementar. Uma segunda forma permite a especificação de um comparador que possa ser usado para tipos que não implementem **Comparable**. Um benefício imediato de **sort()** é que ele nos permite usar uma coleção não classificada, como **LinkedList** e **ArrayList**, mas mesmo assim obtém uma lista classificada quando queremos uma.

Uma vez que você tiver um **List** classificado, poderá usar o método **binarySearch()** para executar buscas muito rápidas. Ele usa a busca binária para encontrar um valor especificado. (Para ver uma descrição de busca binária, consulte a próxima seção Pergunte ao especialista.) Esta é uma das formas de **binarySearch()**:

```
static <T> int binarySearch(List<? extends Comparable<? super T>> lista,
 T valor)
```

Ela procura *valor* em *lista* e retorna o índice de *valor* ou um resultado negativo se *valor* não for encontrado. Como em **sort()**, o tipo dos objetos armazenados na lista deve implementar **Comparable**. Uma segunda forma de **binarySearch()** permite a especificação de um comparador que possa ser usado para tipos que não implementem **Comparable**.

Outro algoritmo muito útil é **replaceAll()**. Ele substitui um item por outro em uma lista inteira. Podemos vê-lo abaixo:

```
static <T< boolean replaceAll(List<T> lista, T anterior, T novo)
```

Quando o método retornar, todas as instâncias de *anterior* terão sido substituídas por *novo*. Ele retorna **true** se pelo menos uma substituição ocorrer. Caso contrário, retorna **false**.

Para inverter uma lista, chame o método **reverse()**, mostrado a seguir:

```
static void reverse(List<T> lista)
```

Esse método pode ser útil quando você tiver uma lista classificada e quiser colocá-la na ordem inversa.

Uma lista pode ser girada com o uso do método **rotate()**. Ele é mostrado abaixo. O giro desloca o conteúdo de uma coleção para a esquerda ou para a direita. O elemento deslocado para fora de uma extremidade é inserido na outra extremidade.

```
static void rotate(List<T> lista, int n)
```

Os elementos de *lista* são girados *n* posições para a direita. Para girar para a esquerda, use um valor negativo para *n*.

O programa a seguir demonstra os algoritmos que acabamos de descrever:

```
// Demonstra vários algoritmos.

import java.util.*;

class AlgorithmsDemo {
 public static void main(String[] args) {

 // Cria uma lista encadeada.
 LinkedList<Character> ll = new LinkedList<Character>();

 // Insere itens na lista.
 for(int i = 0; i < 26; i+=2) {
 ll.add((char) ('A' + i));
 ll.add((char) ('Z' - i));
 }

 // Exibe a lista original.
 System.out.print("Original list: ");
 for(char ch : ll)
 System.out.print(ch);

 System.out.println();

 // Classifica a lista.
 Collections.sort(ll); ←————— Classifica a lista.
 System.out.print("List sorted: ");
 for(char ch : ll)
 System.out.print(ch);

 System.out.println("\n");

 // Pesquisa a lista.
 System.out.println("Using binarySearch() to find X.");
 int i = Collections.binarySearch(ll, 'X'); ←————— Pesquisa a lista.
 if(i >= 0)
 System.out.println("X found. Index is " + i);

 System.out.println();

 // Inverte a lista.
 Collections.reverse(ll); ←————— Inverte a lista.
 System.out.print("List reversed: ");
 for(char ch : ll)
 System.out.print(ch);

 System.out.println("\n");

 // Gira a lista.
```

```

Collections.rotate(ll, 5); ← Gira a lista.
System.out.print("List rotated: ");
for(char ch : ll)
 System.out.print(ch);

System.out.println("\n");

// Cria uma nova lista.
ll = new LinkedList<Character>();

// Adiciona um string a ela.
String str = "this is a test";
for(char ch : str.toCharArray())
 ll.add(ch);

System.out.print("Here is the new list: ");
for(char ch : ll)
 System.out.print(ch);

System.out.println();

// Substitui todos os t por *
Collections.replaceAll(ll, 't', '*'); ← Substitui elementos da lista.
System.out.print("After replacements: ");
for(char ch : ll)
 System.out.print(ch);
}
}

```

A saída do programa é:

```

Original list: AZCXEVGTIRKPMNOLQJSHUFWDYB
List sorted: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Using binarySearch() to find X.
X found. Index is 23

List reversed: ZYXWVUTSRQPONMLKJIHGXFEDCBA

List rotated: EDCBAZYXWVUTSRQPONMLKJIHGXF

Here is the new list: this is a test
After replacements: *his is a *es*

```

Além dos métodos descritos, **Collections** fornece muitos outros, inclusive alguns que geram uma camada de funcionalidade adicional. Por exemplo, se quiser uma versão sincronizada de uma coleção não sincronizada, você pode obter uma usando um dos métodos **synchronizedX**, como **synchronizedSet()** ou **synchronizedMap()**. Também pode obter versões somente leitura de uma coleção usando um dos métodos **unmodifiableX**, como **unmodifiableSet()** e **unmodifiableSortedMap()**. É recomendável que você examine os recursos da classe **Collections** por conta própria. Ela oferece várias funcionalidades prontas para uso que o ajudarão a gerenciar as coleções.

## Pergunte ao especialista

**P** Na discussão dos algoritmos de coleções, você descreveu o método **binarySearch()**. O que é uma busca binária e por que ela é útil?

**R** Em geral, há duas maneiras de se procurar um valor em uma coleção. A primeira é a *busca sequencial*, que começa no início de uma coleção, verificando cada elemento em sequência até o item ser encontrado ou o fim da coleção ser alcançado. Embora essa abordagem funcione, muitos elementos podem ter que ser verificados até encontrarmos o que procuramos. Isso pode resultar em longos tempos de busca. No entanto, para dados não classificados, é a única abordagem que funcionará.

Se os dados estiverem classificados, uma abordagem muito mais rápida, chamada *busca binária*, pode ser usada. A busca binária usa o que pode ser caracterizado como estratégia de “dividir e conquistar” para encontrar um elemento em uma coleção classificada. Funciona assim: o elemento do meio da coleção é verificado. Se for maior do que o item que está sendo procurado, então o elemento do meio da metade inferior da coleção é verificado. Caso contrário, o elemento do meio da metade superior é verificado. Esse processo de divisão da coleção continua até o elemento ser encontrado ou a busca falhar. Para grandes conjuntos de dados, o processo costuma tornar uma busca binária muito mais rápida do que uma busca sequencial. Isso ocorre porque, em média, menos elementos têm que ser verificados. (É claro que, se o elemento que está sendo procurado estiver próximo do início da lista, uma busca sequencial pode superar a busca binária!) Lembre-se também de que a busca binária só é aplicável a coleções classificadas.

## A CLASSE Arrays

Antes de passarmos para as classes legadas, há mais uma classe que faz parte do Collections Framework que merece uma breve menção. Essa classe é **Arrays**. Ela fornece vários métodos **static** que são úteis no trabalho com os arrays internos de tamanho fixo de Java. Por exemplo, fornece métodos para a cópia de um array, o preenchimento de um array com um valor, a classificação de um array e a busca em um array. Também fornece um método chamado **asList()** cujo valor de retorno é um objeto **List** que referencia o array especificado. Isso nos permite usar um array com se fosse uma coleção.

## Verificação do progresso

1. O método da interface **Comparator** é \_\_\_\_\_.
2. A classe **Collections** é uma classe que implementa a interface **Collection**. Verdadeiro ou falso?
3. Que algoritmo de **Collections** cria uma lista classificada?
4. Que algoritmo de **Collections** gira uma lista?

Respostas:

1. **compare()**
2. Falso. A classe **Collections** é uma classe com métodos **static** para o tratamento de objetos **Collection**.
3. **binarySearch()**
4. **rotate()**

## AS CLASSES E INTERFACES LEGADAS

As primeiras versões de **java.util** não incluíam o Collections Framework. Em vez disso, definiam várias classes e uma interface que forneciam um método *ad hoc* para o armazenamento de objetos. Quando as coleções foram adicionadas (pelo J2SE 1.2), duas das classes originais foram reformuladas para dar suporte às interfaces de coleções, e agora, tecnicamente, fazem parte do Collections Framework. No entanto, onde uma coleção moderna duplica a funcionalidade de uma classe legada, geralmente é melhor usar a classe de coleção mais recente.

As classes legadas definidas por **java.util** são:

|            |           |            |       |        |
|------------|-----------|------------|-------|--------|
| Dictionary | Hashtable | Properties | Stack | Vector |
|------------|-----------|------------|-------|--------|

Há uma interface legada chamada **Enumeration**. As seções a seguir examinarão resumidamente **Enumeration** e cada uma das classes legadas.

### A interface Enumeration

A interface **Enumeration** define os métodos com os quais podemos *enumerar* (obter um de cada vez) os elementos de uma coleção de objetos. Embora ainda em uso, essa interface legada foi substituída por **Iterator**.

### Vector

**Vector** implementa um array dinâmico. É semelhante a **ArrayList**, mas com duas diferenças importantes. Em primeiro lugar, **Vector** é sincronizada. Em segundo lugar, contém muitos métodos legados que duplicam a funcionalidade dos métodos definidos pelo Collections Framework. Com o surgimento das coleções, **Vector** foi reformulada para estender **AbstractList** e implementar a interface **List**. Ou seja, foi integrada ao Collections Framework. Já que **Vector** implementa **List**, você pode usar um **Vector** como usaria uma instância de **ArrayList**. Também pode tratá-lo usando seus métodos legados. No entanto, se não precisar de um array dinâmico sincronizado, **ArrayList** é uma melhor opção.

### Stack

**Stack** é uma subclasse de **Vector** que implementa a pilha LIFO padrão. Ela inclui todos os métodos definidos por **Vector** e adiciona métodos como **push()** e **pop()** que dão suporte a operações de pilha. Para códigos novos, recomenda-se **ArrayDeque** ou **LinkedList**. Ambas dão suporte à moderna interface **Deque**.

### Dictionary

**Dictionary** é uma classe abstrata que representa um depósito de armazenamento de chaves/valores e funciona de maneira semelhante a **Map**. Embora atualmente não esteja em desuso, é classificada como obsoleta porque foi totalmente substituída por **Map**.

### Hashtable

**Hashtable** é uma implementação concreta de um **Dictionary**. No entanto, com o surgimento das coleções, foi reformulada para também implementar a interface **Map**.

Logo, **Hashtable** foi integrada ao Collections Framework. Ela é semelhante a **HashMap**, mas é sincronizada. Mas se você não precisar de uma tabela hash sincronizada, **HashMap** é uma melhor opção.

## Properties

**Properties** é uma subclasse de **Hashtable**. É usada para armazenar listas de valores onde tanto a chave quanto o valor são **Strings**. A classe **Properties** é usada por métodos como **System.getProperties()** na obtenção de valores ambientais.

---

## EXERCÍCIOS

- Qual é a finalidade das seguintes classes de coleção: **AbstractCollection**, **AbstractList**, **AbstractQueue**, **AbstractSequentialList**, **AbstractSet**? Isto é, por que essas classes foram incluídas no Collections Framework?
- Tanto o método **poll()** quanto o método **remove()** da interface **Queue** removem e retornam o elemento do início da fila. Qual é a diferença entre eles?
- Um objeto **Collection** representa um grupo arbitrário de dados. Ao contrário, um **Set** representa um grupo de dados em que \_\_\_\_\_.
- O que aconteceria se você usasse o método **add()** para tentar adicionar a um **Set** um elemento que já fizesse parte dele?
- Se **m** for um **Map**, então **m.keySet().size() = m.values().size()**. Verdadeiro ou falso?
- Suponhamos que você tivesse três coleções de strings **c1**, **c2**, **c3** e quisesse descobrir quais strings aparecem em todas as três coleções, se isso ocorrer. Escreva um segmento de código que crie uma coleção **c4** contendo apenas os strings que aparecem nas três coleções. Ele não deve modificar nenhuma das três coleções originais.
- Suponhamos que você tivesse três coleções de strings **c1**, **c2**, **c3** e quisesse descobrir quais strings aparecem em **c1**, mas não aparecem em **c2** ou **c3**, se isso ocorrer. Crie um segmento de código que gere uma coleção **c4** contendo apenas os strings desejados. Ele não deve modificar nenhuma das três coleções originais.
- O que será exibido pelo segmento de código a seguir? Explique.

```
Collection<Object> c = new ArrayList<Object>();
c.add(new Object());
c.add(new Object());
System.out.println(c.contains(new Object()));
```

- Suponhamos que você tivesse um **ArrayList<String>** chamado **list** e um array de tipo **String[ ]** chamado **data**, ambos com o mesmo tamanho. Se quisesse copiar os dados de uma estrutura para a outra, poderia fazê-lo usando um laço. Mas há maneiras mais fáceis. Crie um segmento de código que atribua os strings de **list** a **data** e um que atribua os strings de **data** a **list** sem usar nenhum laço.

10. Crie um programa que converta um **ArrayList** em um **HashSet**, este em um **PriorityQueue**, este em um **ArrayDeque** e este em um **TreeSet** usando a versão de seus construtores que use um **Collection** como argumento. Comece criando um pequeno **ArrayList** de strings e exibindo a lista usando seu método **toString()**. Em seguida, faça o mesmo para construir e exibir os outros **Collections**, usando um como argumento do construtor do próximo. Ainda que todos os **Collections** contenham os mesmos strings, você deve vê-los sendo exibidos em ordens diferentes em alguns deles. Por quê?
11. Crie um programa que exiba os elementos de um **ArrayList** de string não vazio usando cada uma das abordagens a seguir. Por exemplo, se a lista contiver "A", "B", "C" e "D", então, em cada caso, a saída deve ser "A, B, C, D".
  - A. um laço **for** usando chamadas a **get()**
  - B. um laço usando o iterador retornado por **iterator()**
  - C. um laço **for** de estilo for-each
  - D. o método **toString()** da classe **LinkedList**
12. Escreva um programa que crie um **LinkedList** contendo os strings "A", "B", "C" e "D". Em seguida, use um **ListIterator** para percorrer a lista e exibir esta saída: "ABCBCD". O iterador não deve modificar a lista.
13. Escreva um programa que crie um **LinkedList** contendo quatro strings. Em seguida, use um **ListIterator** para trocar os dois valores do meio da lista, usando o método **set( )** do iterador (e não o método **set( )** da lista). O programa deve exibir a lista antes e depois da troca.
14. Implemente um método **sort( )** que use um array de inteiros como parâmetro. Primeiro o método deve criar um **PriorityQueue<Integer>**. Em seguida, deve adicionar todos os inteiros do array à fila. Para concluir, deve extraír um elemento de cada vez da fila usando **poll( )** e inseri-los novamente no array. Crie um programa para testar seu método.
15. Para adicionar um elemento **x** no fim de um **ArrayList** chamado **data**, você poderia chamar **data.add(x)**. Forneça outra chamada de método que faria o mesmo.
16. Para adicionar um elemento **x** no início de um objeto **LinkedList** chamado **data**, há quatro métodos da classe **LinkedList** que você poderia chamar no objeto. Forneça as quatro chamadas de método.
17. Para obter o elemento do início de um objeto **LinkedList** não vazio chamado **data**, há pelo menos seis métodos da classe **LinkedList** que você poderia chamar no objeto, além dos métodos **toArray( )**. Forneça as seis chamadas de método. Eles não devem remover o elemento de **data**.
18. Suponhamos que você tivesse um **LinkedList<String>** não vazio chamado **data**, e que quisesse atribuir a uma variável de string **s** o primeiro string de **data** usando um dos dois métodos **toArray( )**. Mostre como fazê-lo. Isto é, forneça instruções de atribuição que usem cada um dos métodos **toArray( )** para atribuir a **s** o primeiro string de **data**.

19. O segmento de código a seguir é compilado sem erros ou avisos, mas lança uma exceção quando executado. Explique por quê.
- ```
TreeSet<Object> t = new TreeSet<Object>();
t.add(new Object());
```
20. Se um **TreeSet** for criado com um **Comparator** e os elementos implementarem **Comparable**, como a ordem de classificação do conjunto será determinada? Os elementos serão comparados com o uso do **Comparator** ou de seu método **compareTo()**?
21. Escreva um programa que crie um **PriorityQueue** de strings que dê prioridade maior ao string mais longo. Isto é, o método **poll()** sempre retorna o string mais longo da coleção. O programa deve então testar o **PriorityQueue**.
22. Modifique o programa **HashMapDemo** para que exiba a mesma saída de antes, mas sem chamar **entrySet()**. Em vez disso, ele chama **keySet()**, percorre as chaves e então as exibe, assim como seus valores.
23. Suponhamos que você tivesse um **ArrayList<Integer>** chamado **list** e o quisesse classificado do maior valor para o menor. Infelizmente, o método **sort()** da classe **Collections** classificará **list** do menor para o maior. Qual é a maneira mais fácil de classificar **list** do maior para o menor?
24. Suponha que não houvesse um método **get()** na classe **LinkedList** e implemente você mesmo esse método. Mais precisamente, implemente um método **get()** genérico que use um inteiro **i** e um **LinkedList<E>** como seus dois parâmetros e retorne o **i**-ésimo valor da lista. Ele deve lançar uma **IndexOutOfBoundsException** se **i** for negativo ou maior ou igual ao tamanho da lista. Use o método **rotate()** da classe **Collections** para mover o elemento desejado para o início da lista e o método **getFirst()** da classe **LinkedList** para obter o elemento a ser retornado. Quando o método retornar, a lista deve estar novamente em sua forma original.
25. O método **get()** de **Map** recebe uma chave e encontra o valor associado. Crie um programa que faça o oposto, ou seja, dado um valor, que ele encontre uma chave associada. Mais precisamente, crie um método **getReverse()** genérico que receba um objeto de tipo **V** e um **Map<K, V>** como seus parâmetros e determine se o objeto aparece como um dos valores do mapa. Se aparecer, ele retornará a chave (ou qualquer uma das chaves, se houver mais de uma) associada ao valor. Se não aparecer, **null** será retornado. Use o método **equals()** do objeto para compará-lo com outros valores. Crie um programa para testar seu método.

Redes com java.net

PRINCIPAIS HABILIDADES E CONCEITOS

- Conhecer os aspectos básicos de rede
- A classe **InetAddress**
- A classe **Socket**
- A classe **URL**
- A classe **URLConnection**
- A classe **HttpURLConnection**
- A classe **DatagramSocket**
- A classe **DatagramPacket**

Como você aprendeu no Capítulo 1, Java foi projetada para atender às necessidades do ambiente de programação da Internet. Portanto, não deve surpreender o fato de a linguagem dar amplo suporte à rede em sua biblioteca de APIs. O principal pacote de rede de Java é o **java.net**. Ele fornece classes que permitem que um aplicativo acesse recursos de rede de maneira conveniente e eficaz.

Antes de começar, é necessário enfatizar que rede é um assunto avançado. Além disso, é um assunto muito extenso e, às vezes, complexo. Uma discussão sobre rede pode facilmente envolver vários outros tópicos, como detalhes relacionados a protocolos e à arquitetura geral da Internet, que não fazem parte do escopo deste livro. Nossa objetivo aqui é introduzir os elementos-chave do **java.net** e demonstrar seu uso. Isso lhe dará um conhecimento geral dos recursos de rede fornecidos por Java e permitirá que crie aplicativos de rede simples.

ASPECTOS BÁSICOS DE REDES

O conceito básico do suporte Java ao uso de rede é o *soquete*. Um soquete identifica uma extremidade em uma rede. Ele é a base das redes modernas porque permite que um único computador atenda muitos clientes diferentes ao mesmo tempo e forneça vários tipos de informações diferentes. Isso é feito com o uso de uma *porta*, que é um soquete numerado em uma máquina específica. Diz-se que um processo de servidor está “escutando” uma porta até um cliente se conectar a ela. Um servidor pode aceitar vários clientes conectados ao mesmo número de porta, mas cada sessão é exclusiva.

A comunicação por soquete ocorre via um protocolo. O *Internet Protocol (IP)* é um protocolo de roteamento de baixo nível que divide os dados em pequenos pacotes e os envia para um endereço por uma rede. No entanto, ele não garante a distribuição desses pacotes para o destino ou sua distribuição na ordem em que foram enviados. O *Transmission Control Protocol (TCP)* é um protocolo de nível superior que gerencia a transmissão de pacotes, classificando e retransmitindo-os quando necessário para transmitir os dados de maneira confiável. Para fazê-lo, estabelece uma *conexão*, que é um link entre dois soquetes. Um terceiro protocolo, o *User Datagram Protocol (UDP)*, pode ser usado diretamente para dar suporte à transmissão rápida e sem conexão de pacotes, porém não dá garantias.

Uma vez que uma conexão é estabelecida, um protocolo de nível superior assume de acordo com a porta que está sendo usada. O TCP/IP reserva as 1.024 portas inferiores para protocolos específicos. Veja alguns exemplos: o número de porta 21 é para o FTP, 25 é para email, 43 é para o whois e 80 é para o HTTP. É função de cada protocolo determinar como um cliente deve interagir com a porta. O protocolo que você deve conhecer melhor é o HTTP (Hypertext Transfer Protocol). Trata-se do protocolo que os navegadores Web e os servidores usam para transferir recursos, como páginas Web e imagens. Funciona assim: quando um cliente solicita um arquivo a um servidor HTTP, ele apenas envia o nome do arquivo para uma porta predefinida. O servidor retorna o conteúdo do arquivo. Ele também responde com um código de status para informar ao cliente se a solicitação pode ser atendida e, caso não possa, por quê.

Um componente-chave da Internet é o *endereço*. Todo computador que está na Internet tem um. Um endereço de Internet é um número que identifica de maneira exclusiva cada computador na rede. Originalmente, todos os endereços de Internet eram compostos por valores de 32 bits, organizados como quatro valores de 8 bits. Esse tipo de endereço era especificado pelo IPv4 (Internet Protocol, versão 4). No entanto, surgiu um novo esquema de endereçamento, chamado IPv6 (Internet Protocol, versão 6). O IPv6 usa um valor de 128 bits para representar um endereço, organizado em oito unidades de 16 bits. A principal vantagem do IPv6 é que ele dá suporte a um espaço de endereço muito maior que o do IPv4. Felizmente, ao usar Java, não precisamos saber se estão sendo usados endereços IPv4 ou IPv6, porque a linguagem é que manipula os detalhes.

Você não deve trabalhar diretamente com endereços numéricos. Em vez disso, usará um *nome de domínio* legível por humanos. Por exemplo, **www.mcgraw-hill.com** fica no domínio de nível superior *COM*. Seu nome é *mcgraw-hill*. Um nome de domínio da Internet é mapeado para um endereço IP pelo *Sistema de Nome de Domínio (DNS, Domain Name System)*. Isso permite que os usuários trabalhem com nomes de domínio, mas a Internet opera com endereços IP. Um endereço como **www.mcgraw-hill.com** também é chamado de *nome de host*.

AS CLASSES E INTERFACES DE REDES

Java dá amplo suporte às famílias de protocolos TCP e UDP. Logo, o pacote **java.net** fornece muitas classes e várias interfaces. Para dar uma ideia de seu extenso

conjunto de recursos, estas eram as classes definidos pelo **java.net** quando o livro foi escrito:

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress	SocketImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager	MulticastSocket	StandardSocketOptions
DatagramPacket	NetPermission	URI
DatagramSocket	NetworkInterface	URL
DatagramSocketImpl	PasswordAuthentication	URLClassLoader
HttpCookie	Proxy	URLConnection
HttpURLConnection	ProxySelector	URLDecoder
IDN	ResponseCache	URLEncoder
Inet4Address	SecureCacheResponse	URLStreamHandler

As interfaces do **java.net** são:

ContentHandlerFactory	CookiePolicy	CookieStore
DatagramSocketImplFactory	FileNameMap	ProtocolFamily
SocketImplFactory	SocketOption	SocketOptions
URLStreamHandlerFactory		

Como as tabelas sugerem, **java.net** fornece um rico conjunto de funcionalidades que proporcionam um controle rigoroso sobre as comunicações de rede.

Nas seções a seguir, examinaremos várias classes básicas fornecidas pelo **java.net** e mostraremos exemplos que ilustram seu uso. As classes descritas aqui são:

InetAddress	Socket	URL
URLConnection	HttpURLConnection	DatagramPacket
DatagramSocket		

Elas representam vários aspectos básicos da rede baseada em Java, inclusive como criar um endereço, como estabelecer uma conexão e como enviar e receber dados. Um conhecimento geral dessas operações essenciais fornece uma base em que outros aspectos da rede Java podem ser integrados.

A CLASSE InetAddress

A classe **InetAddress** é usada para encapsular tanto o endereço IP numérico quanto seu nome de domínio. Você interagirá com a classe usando o nome de um host IP, que é mais conveniente e inteligível que seu endereço IP. Ela pode tratar endereços IPv4 e IPv6.

InetAddress não tem construtores visíveis. Para criar um objeto **InetAddress**, você usará um de seus métodos **static**. Dois são mostrados abaixo:

```
static InetAddress getLocalHost()
    throws UnknownHostException

static InetAddress getByName(String nomeHost)
    throws UnknownHostException
```

O método **getLocalHost()** retorna o objeto **InetAddress** que representa o host local (que é o computador local). O método **getByName()** retorna o **InetAddress** de um nome de host passado para ele. Se esses métodos não puderem resolver o nome do host, lançarão uma **UnknownHostException**. Também há um método **getAllByName()** que retorna um array de **InetAddresses** representando todos os endereços para os quais um nome específico é mapeado. Esse método é útil em casos em que um único nome é usado para representar várias máquinas. Ele também lançará uma **UnknownHostException** se não puder resolver o nome para pelo menos um endereço.

Há vários métodos que podem ser chamados em uma instância de **InetAddress**. Usaremos dois deles:

```
String getHostAddress()
String getHostName()
```

O método **getHostAddress()** retorna um string que lista o endereço IP do host usando sua forma numérica. O método **getHostName()** retorna o nome que representa o endereço do host. Por conveniência, **InetAddress** também sobrepuja o método **toString()**. Ele retorna tanto o nome do host quanto seu endereço.

O exemplo a seguir demonstra **InetAddress** exibindo os endereços e nomes de vários sites:

```
// Demonstra InetAddress.

import java.net.*;

class InetAddressDemo {
    public static void main(String[] args) {

        try {
            InetAddress address = InetAddress.getByName("www.mcgraw-hill.com");
            System.out.println("Host name: " + address.getHostName());
            System.out.println("Address: " + address.getHostAddress());

            System.out.println();

            address = InetAddress.getByName("www.mhhe.com");
            System.out.println("Host name: " + address.getHostName());
            System.out.println("Address: " + address.getHostAddress());

            System.out.println();
        }
    }
}
```

```
        address = InetAddress.getByName("www.mheducation.com");
        System.out.println("Host name: " + address.getHostName());
        System.out.println("Address: " + address.getHostAddress());
    } catch (UnknownHostException exc) {
        System.out.println(exc);
    }
}
```

Esta é a saída produzida pelo programa. (É claro que a que você verá pode ser um pouco diferente.)

```
Host name: www.mcgraw-hill.com  
Address: 204.8.135.3  
  
Host name: www.mhhe.com  
Address: 12.26.55.139  
  
Host name: www.mheducation.com  
Address: 12.163.148.101
```

Pergunte ao especialista

P Como `InetAddress` lida com as diferenças entre o IPv4 e o IPv6?

R Para lidar com as diferenças entre o IPv4 e o IPv6, o JDK 1.4 adicionou duas subclasses de **InetAddress**: **Inet4Address** e **Inet6Address**. **Inet4Address** representa um endereço IPv4 de estilo tradicional. **Inet6Address** encapsula um endereço IPv6 de estilo novo. Como são subclasses de **InetAddress**, uma referência de tipo **InetAddress** pode apontar para ambas. Essa foi uma maneira de Java adicionar a funcionalidade do IPv6 sem danificar códigos existentes ou criar muitas outras classes. Na maioria dos casos, podemos simplesmente usar a classe **InetAddress** ao trabalhar com endereços IP porque ela consegue acomodar os dois estilos.

A CLASSE Socket

Soquetes TCP/IP são usados para implementar conexões ponto a ponto, baseadas em fluxo, confiáveis, bidirecionais e persistentes entre hosts na Internet. Por intermédio de um soquete, você pode usar o I/O (input/output, entrada/saída) Java baseado em fluxo para se comunicar com outros programas que residam na máquina local ou em qualquer outra máquina na Internet. É claro que essas conexões estão sujeitas a restrições impostas pelo gerenciador de segurança. Por exemplo, como regra geral, os applets só podem estabelecer conexões de soquete com o host do qual foram baixados. Essa restrição existe porque seria perigoso applets carregados de um firewall ter acesso a qualquer máquina.

Há dois tipos de soquetes TCP em Java. Um é para servidores e o outro para clientes. A classe **ServerSocket** foi projetada para ser um “ouvinte”, que espera

clientes se conectarem antes de fazer algo. Logo, **ServerSocket** é para servidores. A classe **Socket** é para clientes. Ela foi projetada para se conectar com soquetes de servidor e iniciar trocas de protocolo. Aqui, examinaremos **Socket**.

Socket define vários construtores. Este é o que usaremos:

```
Socket(String nomeHost, int porta)
       throws UnknownHostException, IOException
```

Ele cria um soquete conectado ao host e à porta nomeados, e estabelece implicitamente uma conexão entre o cliente e o servidor. Ele lançará uma **UnknownHostException** se o host não puder ser encontrado e uma **IOException** se ocorrer um erro de I/O.

Socket define vários métodos de instância. Por exemplo, um **Socket** pode ser examinado a qualquer momento quanto às informações de endereço e porta associadas a ele com o uso dos métodos a seguir:

```
InetAddress getInetAddress()
int getPort()
int getLocalPort()
```

Se o soquete estiver conectado a um servidor, o método **getInetAddress()** retornará seu endereço na forma de uma instância de **InetAddress**. Caso contrário, retornará **null**. Da mesma forma, se o soquete estiver conectado a um servidor, o método **getPort()** retornará o número da porta no servidor. Caso contrário, retornará 0. Você pode obter o número da porta local chamando o método **getLocalPort()**. Ele retorna -1 se o soquete não estiver vinculado a uma porta.

Você pode ganhar acesso aos fluxos de entrada e saída associados a um **Socket** usando os métodos **getInputStream()** e **getOutputStream()**, como mostrado aqui. Os dois podem lançar uma **IOException** se ocorrer um erro de I/O. Esses fluxos são usados exatamente da mesma forma que os fluxos de I/O descritos no Capítulo 11 para enviar e receber dados.

```
InputStream getInputStream() throws IOException
OutputStream getOutputStream() throws IOException
```

O método **getInputStream()** retorna o fluxo de entrada associado ao soquete chamar. O fluxo de saída do soquete é retornado por **getOutputStream()**.

Socket fornece vários outros métodos. Por exemplo, você pode determinar se um soquete está conectado a um servidor chamando **isConnected()**. Para verificar se um soquete está vinculado a um endereço, chame **isBound()**. Se um soquete for fechado, **isClosed()** retornará **true**. Para fechar um soquete, chame **close()**. O fechamento de um soquete também fecha os fluxos de I/O associados a ele. A partir do JDK 7, o soquete passou a implementar **AutoCloseable**, ou seja, você pode usar um bloco **try-with-resources** para gerenciá-lo.

O programa a seguir fornece um exemplo simples de **Socket**, usando “whois” para determinar um nome de domínio. Ele faz isso abrindo uma conexão com uma porta whois (porta 43) no servidor **Whois.InterNIC.net**. Em seguida, envia um nome de domínio para o servidor. Para concluir, exibe os dados que são retornados. O servidor whois procurará o argumento como um nome de domínio de Internet registrado. Se o nome for encontrado, retornará o endereço IP e as informações de contato desse site.

```
// Demonstra Sockets.

import java.net.*;
import java.io.*;

class SocketDemo {
    public static void main(String[] args) {
        int ch;
        Socket socket = null;

        try {
            // Cria um soquete conectado a whois.internic.net, porta 43.
            socket = new Socket("whois.internic.net", 43); ← Constrói um Socket.

            // Obtém os fluxos de entrada e saída.
            InputStream in = socket.getInputStream(); ← Obtém os fluxos
            OutputStream out = socket.getOutputStream(); ← de entrada e saída
                                                       do soquete.

            // Constrói um string de solicitação.
            String str = (args.length == 0 ? "mcgraw-hill.com" :
                           args[0]) + "\n";
            // Converte em bytes.
            byte[] buf = str.getBytes();

            // Envia a solicitação.
            out.write(buf); ← Envia uma solicitação.

            // Lê e exibe a resposta.
            while ((ch = in.read()) != -1) { ← Obtém a resposta.
                System.out.print((char) ch);
            }
        } catch(IOException exc) {
            System.out.println(exc);
        } finally {
            try {
                if(socket != null) socket.close();
            } catch(IOException exc) {
                System.out.println("Error closing socket: " + exc);
            }
        }
    }
}
```

Para usar o programa, especifique o nome do site em que está interessado na linha de comando. Se não for especificado um argumento de linha de comando, o site **mcgraw-hill.com** será usado por padrão. Se você usar o padrão, verá algo semelhante ao seguinte:

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.
```

```

Domain Name: MCGRAW-HILL.COM
Registrar: MELBOURNE IT, LTD. D/B/A INTERNET NAMES WORLDWIDE
Whois Server: whois.melbourneit.com
Referral URL: http://www.melbourneit.com
Name Server: CORP-55W-NS1.MCGRAW-HILL.COM
Name Server: CORP-HTS-NS1.MCGRAW-HILL.COM
Name Server: CORP-UKC-NS1.MCGRAW-HILL.COM
Status: clientTransferProhibited
Updated Date: 07-mar-2011
Creation Date: 07-may-1994
Expiration Date: 08-may-2012

```

É assim que o programa funciona. Primeiro, é construído um **Socket** que especifica o nome de host "whois.internic.net" e o número de porta 43. **Internic.net** é o site InterNIC que trata solicitações ao whois. A porta 43 é a porta do whois. Em seguida, os fluxos de entrada e saída são abertos no soquete e um string é construído contendo o nome do site sobre o qual informações são desejadas. Como mencionado, se nenhum site for especificado na linha de comando, "mcgraw-hill" será usado. O string é convertido no array **byte** com o uso do método **getBytes()** de **String** e enviada para fora do soquete. A resposta é obtida com uma leitura no soquete e os resultados são exibidos. Para concluir, o soquete é fechado, o que também fecha os fluxos de I/O.

No exemplo anterior, o soquete foi fechado manualmente com uma chamada a **close()**. Se você estiver usando o JDK 7 ou posterior, poderá usar um bloco **try-with-resources** para fechar automaticamente o soquete. Por exemplo, esta é outra maneira de escrever o método **main()** do programa anterior:

```

// Usa o gerenciamento automático de recursos para fechar um soquete.
public static void main(String[] args) {
    int ch;

    // Cria um soquete conectado a internic.net, porta 43. Gerencia
    // esse soquete com um bloco try-with-resources.
    try ( Socket socket = new Socket("whois.internic.net", 43) ) {

        // Obtém os fluxos de entrada e saída.
        InputStream in = socket.getInputStream();
        OutputStream out = socket.getOutputStream();

        // Constrói um string de solicitação.
        String str = (args.length == 0 ? "mcgraw-hill.com" :
                      args[0]) + "\n";
        // Converte em bytes.
        byte[] buf = str.getBytes();

        // Envia a solicitação.
        out.write(buf);

        // Lê e exibe a resposta.
        while ((ch = in.read()) != -1) {

```

```

        System.out.print((char) ch);
    }
} catch(IOException exc) {
    System.out.println(exc);
}
// Agora o soquete é fechado.
}

```

Nessa versão, o soquete é fechado automaticamente quando o bloco **try** termina.

Os exemplos funcionarão com versões de Java anteriores ao JDK 7. Para ilustrar com clareza quando um recurso de rede pode ser fechado, os próximos programas continuarão chamando **close()** explicitamente. No entanto, em seu código, você deve considerar o uso do gerenciamento automático de recursos, já que ele oferece uma abordagem mais otimizada.

Pergunte ao especialista

P Nessa última discussão, você mencionou a classe **ServerSocket**. Pode falar algo sobre ela?

R A classe **ServerSocket** é usada na criação de aplicativos de servidor. Quando você criar um **ServerSocket**, ele se registrará no sistema como tendo interesse em conexões de cliente em uma porta especificada. **ServerSocket** tem um método chamado **accept()** que espera um cliente iniciar comunicações. Quando isso ocorre, **accept()** retorna um **Socket**, que é então usado na comunicação com o cliente.

Verificação do progresso

1. O que um objeto **InetAddress** encapsula?
2. O que é um “soquete”?
3. Os **Sockets** enviam e recebem dados para e de **ServerSockets** usando _____.

A CLASSE URL

URL é a sigla de Uniform Resource Locator. É claro que você conhece os URLs, porque elas fornecem um modo de identificar ou endereçar recursos na Internet. Quando você insere um site em um navegador, está especificando o URL desse site. Java dá suporte aos URLs com a classe **URL**. Após criar uma instância de **URL**, você poderá usá-la para acessar a Internet.

Todos os URLs compartilham o mesmo formato básico, embora alguma variação seja permitida. Veja dois exemplos: <http://www.mhhe.com/> e <http://www.mhhe.com:80/index.html>. A especificação de um URL se baseia em quatro compo-

Respostas:

1. Tanto o endereço IP numérico quanto o nome de domínio desse endereço.
2. É a extremidade de uma conexão entre um cliente e um servidor via rede.
3. **InputStreams** e **OutputStreams**.

nentes principais. O primeiro é o protocolo a ser usado, separado do resto do localizador por dois-pontos (:). Protocolos comuns seriam o HTTP e o FTP. O segundo componente é o nome ou o endereço IP do host a ser usado; ele é definido à esquerda por barras duplas (/) e à direita por uma barra (/) ou, opcionalmente, por dois-pontos (:). O terceiro componente, o número da porta, é um parâmetro opcional, delimitado à esquerda do nome do host por dois-pontos (:) e à direita por uma barra (/). (Para o HTTP, o número de porta padrão é 80, a porta predefinida para o protocolo; logo, no HTTP, o ":80" é desnecessário. Outros padrões são usados para outros protocolos.) A quarta parte é o caminho do recurso, como o de um arquivo.

URL tem vários construtores. Examinaremos três, começando com o mais simples, mostrado abaixo:

```
URL(String especificadorURL) throws MalformedURLException
```

Aqui, *especificadorURL* é um string que especifica um URL completo, semelhante ao que seria inserido em um navegador. Os dois construtores a seguir permitem a divisão do URL nas partes que o compõem:

```
URL(String nomeProtocolo, String nomeHost, int porta, String caminho)
    throws MalformedURLException
```

```
URL(String nomeProtocolo, String nomeHost, String caminho)
    throws MalformedURLException
```

Aqui, cada parte do URL é especificado por um string individual. Nos três construtores, uma **MalformedURLException** será lançada se o protocolo for inválido ou um argumento for **null**.

Há métodos definidos por URL que permitem a obtenção dos componentes individuais de um URL. Vejamos alguns. Para obter o protocolo, podemos chamar **getProtocol()**, e para obter o nome do host, **getHost()**. Para obter o nome do arquivo, chame **getFile()**. Os métodos são mostrados abaixo:

```
String getProtocol()
String getHost()
String getFile()
```

Todos retornam um string contendo a informação desejada. Você pode recuperar o número da porta chamando o método **getPort()**, mostrado aqui:

```
int getPort()
```

Ele retorna a porta associada ao URL ou -1 se a porta não for especificada.

O exemplo a seguir cria um URL para **www.mhhe.com:80/index.html** e exibe seus componentes:

```
// Demonstra URL.

import java.net.*;

class URLDemo {
    public static void main(String[] args) {

        try {
            URL url = new URL("http://www.mhhe.com:80/index.html");
        }
    }
}
```

```
        System.out.println("Protocol: " + url.getProtocol());
        System.out.println("Port: " + url.getPort());

        System.out.println("Host: " + url.getHost());
        System.out.println("File: " + url.getFile());
    } catch (MalformedURLException exc) {
        System.out.println("Invalid URL: " + exc);
    }
}
```

Quando você executá-lo, obterá a saída abaixo:

Protocol: http
Port: 80
Host: www.mhhe.com
File: /index.html

Além dos métodos que acabamos de demonstrar, **URL** define vários outros. Um particularmente interessante é **openConnection()**. Ele retorna uma instância de **URLConnection**, que encapsula informações sobre uma conexão. O método **openConnection()** é mostrado aqui:

URLConnection openConnection() throws IOException

Observe que ele pode lançar uma **IOException**. Você verá esse método em ação na próxima seção.

A CLASSE URLConnection

URLConnection é uma classe abstrata que encapsula uma conexão baseada em URL. Podemos obter um **URLConnection** usando o método `openConnection()` de **URL**, como descrito na seção anterior. Uma vez tendo uma conexão com um servidor remoto, podemos usar **URLConnection** para verificar as propriedades do recurso. Também podemos abrir um fluxo de entrada para baixá-lo. **URLConnection** define vários métodos. Um resumo é mostrado na Tabela 26-1.

Observe que **URLConnection** define muitos métodos que tratam informações de cabeçalho. Um cabeçalho é composto por pares de chaves e valores representados como strings. Usando **getHeaderField()**, você pode obter o valor associado à chave de um cabeçalho, e chamando **getHeaderFields()**, pode obter um mapa contendo todos os cabeçalhos. Vários campos de cabeçalho padrão estão disponíveis diretamente por meio de métodos como **getDate()** e **getContentType()**.

O exemplo a seguir cria um **URLConnection** usando o método **openConnection()** de um objeto **URL**, e então o usa para examinar as propriedades e o conteúdo do documento:

```
// Demonstra URLConnection.  
import java.net.*;  
import java.io.*;  
import java.util.*;  
  
class UCDemo
```

Tabela 26-1 Resumo dos métodos definidos por URLConnection

Método	Descrição
int getContentType()	Retorna o tipo de conteúdo encontrado no recurso. Retorna null se o tipo de conteúdo não estiver disponível.
long getDate()	Retorna a hora e a data da resposta representadas em milissegundos desde primeiro de janeiro de 1970 GMT. Zero é retornado se a hora e a data não estiverem disponíveis.
long getExpiration()	Retorna a hora e a data de expiração do recurso representadas em milissegundos desde primeiro de janeiro de 1970 GMT. Zero é retornado se a data de expiração não estiver disponível.
String getHeaderField(int <i>índice</i>)	Retorna o valor do campo de cabeçalho do índice <i>índice</i> . (Os índices dos campos de cabeçalho começam em 0.) Retorna null se o valor de <i>índice</i> exceder o número de campos.
String getHeaderField(String <i>nomeCampo</i>)	Retorna o valor do campo de cabeçalho cujo nome é especificado por <i>nomeCampo</i> . Retorna null se o nome especificado não for encontrado.
String getHeaderFieldKey(int <i>índice</i>)	Retorna a chave do campo de cabeçalho do índice <i>índice</i> . (Os índices dos campos de cabeçalho começam em 0.) Retorna null se o valor de <i>índice</i> exceder o número de campos
Map<String, List<String>> getHeaderFields()	Retorna um mapa contendo todos os valores e campos de cabeçalho.
long getLastModified()	Retorna a hora e a data, representadas em milissegundos desde primeiro de janeiro de 1970 GMT, da última modificação feita no recurso. Zero é retornado se a data da última modificação não estiver disponível.
InputStream getInputStream() throws IOException	Retorna um InputStream vinculado à conexão.
OutputStream getOutputStream() throws IOException	Retorna um OutputStream vinculado à conexão.

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

        InputStream in = null;
        URLConnection connection = null;

        try {
            URL url = new URL("http://www.mcgraw-hill.com");
        }
    }
}
```

```

connection = url.openConnection(); ← Abre uma conexão.

// obtém a data
long d = connection.getDate(); ←
if(d==0)
    System.out.println("No date information.");
else
    System.out.println("Date: " + new Date(d));

// obtém o tipo do conteúdo
System.out.println("Content-Type: " +
                    connection.getContentType());

// obtém a data de expiração
d = connection.getExpiration(); ←
if(d==0)
    System.out.println("No expiration information.");
else
    System.out.println("Expires: " + new Date(d));

// obtém a data da última modificação
d = connection.getLastModified(); ←
if(d==0)
    System.out.println("No last-modified information.");
else
    System.out.println("Last-Modified: " + new Date(d));

// obtém o tamanho do conteúdo
long len = connection.getContentLengthLong(); ←
if(len == -1)
    System.out.println("Content length unavailable.");
else
    System.out.println("Content-Length: " + len);

if(len != 0) {
    System.out.println("== Content ==");
    in = connection.getInputStream(); ← Obtém o fluxo de entrada.

    int ch;
    while (((ch = in.read()) != -1)) { ← Lê e exibe o conteúdo.
        System.out.print((char) ch);
    }
} else {
    System.out.println("No content available.");
}
} catch(IOException exc) {
    System.out.println("Connection Error: " + exc);
} finally {
    try {
        if(in != null) in.close();
    }
}

```

Obtém vários valores.

Obtém o fluxo de entrada.

Lê e exibe o conteúdo.

```
        } catch(IOException exc) {
            System.out.println("Error closing connection: " + exc);
        }
    }
}
```

O programa estabelece uma conexão HTTP com www.mcgraw-hill.com. Em seguida, exibe vários valores de cabeçalho e recupera o conteúdo. Estas são as primeiras linhas da saída (a saída exata deve mudar com o tempo):

```
Date: Mon Jul 18 10:59:42 CDT 2011
Content-Type: text/html; charset=ISO-8859-1
No expiration information.
No last-modified information.
Content-Length: 50631
== Content ==
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
.
.
```

TENTE ISTO 26-1 Baixe um arquivo da Internet

GetFileFromSite.java

Neste projeto, você criará um programa para baixar um arquivo da Internet e salvá-lo em seu computador. O programa se chama **GetFileFromSite** e recebe como entrada o URL de um arquivo e o nome que será dado a ele em seu computador. Por exemplo, para usar o programa para baixar a home page de **www.mhhe.com** e armazená-la em um arquivo chamado "mhhehomepage.html", você usaria o comando

```
java GetFileFromSite http://www.mhhe.com mhhehomepage.html
```

O programa usa um **URLConnection** para baixar o conteúdo do arquivo.

PASSO A PASSO

1. Crie um novo arquivo chamado **GetFileFromSite.java** e insira o código a seguir:

```
import java.net.*;
import java.io.*;

class GetFileFromSite {
    public static void main(String[] args) {
        if(args.length != 2) {
            System.out.println("Usage: java GetFileFromSite url file");
        }
    }
}
```

```
        return;
    }

InputStream in = null;
URLConnection connection = null;
FileOutputStream fout = null;

try {
    URL url = new URL(args[0]);
    connection = url.openConnection();
    in = connection.getInputStream();
    fout = new FileOutputStream(args[1]);

    // Baixa e salva o arquivo.
    int b;
    while (((b = in.read()) != -1)) {
        fout.write(b);
    }
} catch (IOException exc) {
    System.out.println("Connection Error: " + exc);
} finally {
    try {
        if(in != null) in.close();
        if(fout != null) fout.close();
    } catch (IOException exc) {
        System.out.println("Error closing stream: " + exc);
    }
}
```

2. Examinemos com detalhes como esse programa funciona. Primeiro ele verifica se há a quantidade correta de argumentos de linha de comando. Em seguida, cria três referências. A primeira é um **InputStream** chamado **in**. A segunda é um **URLConnection** chamado **connection**. A terceira é um **FileOutputStream** chamado **fout**. A principal ação do programa ocorre dentro do bloco **try**.

O bloco **try** começa criando um objeto **URL** chamado **url** para o URL especificado na linha de comando. Em seguida, uma conexão com o URL é aberta com uma chamada a **openConnection()** em **url**. Essa conexão é atribuída a **connection**. Agora, um **InputStream** é obtido com uma chamada a **getInputStream** em **connection** e atribuído a **in**. Logo, **in** referencia o arquivo que será baixado. Um **FileOutputStream** associado ao nome de arquivo especificado como arquivo de destino na linha de comando é atribuído a **fout**. Para concluir, o programa copia os bytes recebidos do fluxo de entrada de rede no fluxo de saída de arquivo. Isso resulta no arquivo sendo baixado e armazenado no computador.

3. Compile o programa e, para testá-lo, execute-o com o comando a seguir (todo em uma linha):

```
| java GetFileFromSite http://highered.mcgraw-hill.com/sites/dl/free/0072974168/584690/SourceCode.zip Projects.zip
```

O URL do comando corresponde a um arquivo zip criado por um dos autores para uso com um livro de design orientado a objetos. Quando o programa terminar, você verá um novo arquivo chamado “Projects.zip” no mesmo diretório de seu programa. Se descompactar esse arquivo, verá pastas contendo código-fonte Java.

A CLASSE HttpURLConnection

Java fornece uma subclasse de **URLConnection** que dá suporte adicional a conexões HTTP. Essa classe se chama **HttpURLConnection**. Você pode obter um **HttpURLConnection** do modo como acabamos de mostrar chamando **openConnection()** em um objeto **URL**, mas deve converter o resultado para **HttpURLConnection**. É claro que deve se certificar se está realmente abrindo uma conexão HTTP.

Uma vez que tiver obtido uma referência a um objeto **HttpURLConnection**, você poderá usar qualquer um dos métodos herdados de **URLConnection**. Também poderá usar os diversos métodos definidos por **HttpURLConnection**. O próximo exemplo usa três. O primeiro é **getRequestMethod**, mostrado aqui:

```
String getRequestMethod()
```

Ele retorna um string representando como são feitas solicitações HTTP. O padrão é GET. Outras opções, como POST, estão disponíveis.

O próximo método é **getResponseCode()**, mostrado abaixo:

```
int getResponseCode() throws IOException
```

Ele retorna o código de resposta HTTP. Se nenhum código de resposta for obtido, -1 será retornado. Uma **IOException** será lançada se a conexão falhar. Um código de resposta no intervalo 200 indica sucesso. Um código de resposta no intervalo 300 indica redirecionamento. Uma resposta no intervalo 400 indica um erro de solicitação de algum tipo, como, por exemplo, sintaxe inválida. Respostas no intervalo 500 indicam erro de servidor. Normalmente, quando tudo dá certo, 200 é retornado para indicar sucesso.

O terceiro método que usaremos é **getResponseMessage()**:

```
String getResponseMessage() throws IOException
```

Ele retorna a mensagem de resposta associada ao código de resposta. Se nenhuma mensagem estiver disponível, **null** será retornado. Uma **IOException** será lançada se a conexão falhar.

O programa a seguir demonstra **HttpURLConnection**. Primeiro ele estabelece uma conexão com **www.mcgraw-hill.com**. Em seguida, exibe o método de solicitação, o código de resposta e a mensagem de resposta. Para concluir, exibe as chaves e valores do cabeçalho de resposta.

```

// Demonstra HttpURLConnection.

import java.net.*;
import java.io.*;
import java.util.*;

class HttpURLConnectionDemo
{
    public static void main(String[] args) {

        try {
            URL url = new URL("http://www.mcgraw-hill.com");
            HttpURLConnection connection =
                (HttpURLConnection) url.openConnection();

            // Exibe o método de solicitação.
            System.out.println("Request method is " +
                               connection.getRequestMethod());

            // Exibe o código de resposta.
            System.out.println("Response code is " +
                               connection.getResponseCode());

            // Exibe a mensagem de resposta.
            System.out.println("Response Message is " +
                               connection.getResponseMessage());

            // Obtém uma lista dos campos de cabeçalho
            // e um conjunto das chaves de cabeçalho.
            Map<String, List<String>> hdrMap = connection.getHeaderFields();
            Set<String> hdrKeys = hdrMap.keySet(); ←
                                         Observe como um conjunto
                                         das chaves é obtido.

            System.out.println("\nHere is the header:");

            // Exibe todas as chaves e valores de cabeçalho.
            for(String k : hdrKeys) {
                System.out.println("Key: " + k + ← Agora, exibe as chaves e os valores.
                                  " Value: " + hdrMap.get(k));
            }
        } catch(IOException exc) {
            System.out.println(exc);
        }
    }
}

```

A saída produzida pelo programa é mostrada aqui. (É claro que a resposta exata retornada por **www.mhhe.com** mudará com o tempo.)

```

Request method is GET
Response code is 200
Response Message is OK

```

```
Here is the header:
Key: null Value: [HTTP/1.1 200 OK]
Key: Date Value: [Mon, 18 Jul 2011 16:15:47 GMT]
Key: Content-Length Value: [50631]
Key: Keep-Alive Value: [timeout=5, max=100]
Key: Content-Type Value: [text/html; charset=ISO-8859-1]
Key: Connection Value: [Keep-Alive]
Key: X-Powered-By Value: [Servlet/2.5 JSP/2.1]
```

Observe como as chaves e valores de cabeçalho são exibidos. Primeiro, um mapa das chaves e valores é obtido com uma chamada a `getHeaderFields()` (que é herdado de `URLConnection`). Em seguida, um conjunto das chaves é recuperado com uma chamada a `keySet()` no mapa. O conjunto é então percorrido com o uso de um laço `for` de estilo for-each. O valor associado a cada chave é obtido com uma chamada a `get()` no mapa.

Pergunte ao especialista

P Na tabela de classes de `java.net` mostrada anteriormente, notei a classe `URI`. Que classe é essa?

R A classe `URI` encapsula um *Identificador de Recurso Uniforme* (`URI`, Uniform Resource Identifier). Os `URIs` são semelhantes aos URLs. Na verdade, os URLs são um subconjunto dos `URIs`. Um `URI` representa uma maneira padrão de identificar um recurso. Um URL também descreve como acessar o recurso. Você pode converter uma instância de `URI` em uma instância de `URL` usando o método `toURL()`. Para converter um `URL` em um `URI`, use `toURI()`.

Verificação do progresso

1. Os quatro componentes de um URL são _____.
2. Você pode criar um `URLConnection` chamando _____.
3. Cite dois métodos adicionados pela classe `HttpURLConnection`.

DATAGRAMAS

Os datagramas fornecem uma alternativa à rede de estilo TCP/IP que acabamos de discutir. Eles são conjuntos de informações passados entre as máquinas. Os datagramas diferem da rede TCP/IP em um aspecto muito importante: um datagrama não tem garantias de que alcançará seu destino. Uma vez que o datagrama é enviado para seu destino, não há garantias de que chegará ou nem mesmo de que o destino estará lá

Respostas:

1. protocolo, nome do host, porta e caminho do arquivo
2. o método `openConnection()` da classe `URL`
3. `getRequestMethod()` e `getResponseCode()`

para recebê-lo. Além disso, quando o datagrama é recebido, não há garantias de que não tenha sido danificado em trânsito ou de que quem o enviou ainda esteja presente para receber uma resposta. Os datagramas também podem alcançar seu destino em uma ordem diferente daquela em que são enviados.

Os datagramas têm suporte principalmente em duas classes. A primeira é **DatagramPacket**, que contém os dados. A segunda é **DatagramSocket**, que é a classe de soquete usada para enviar ou receber os **DatagramPackets**. Veremos as duas aqui.

Nota: Outra classe que dá suporte aos datagramas é **MulticastSocket**. Ela é usada para datagramas multicast. O multicasting permite que um servidor envie um datagrama para vários clientes.

DatagramSocket

DatagramSocket define vários construtores. O que usaremos é:

```
DatagramSocket(int porta) throws SocketException
```

Ele cria um **DatagramSocket** para o host local que usa a porta especificada por *porta*. Observe que o método pode lançar uma **SocketException** se um erro ocorrer na criação do soquete.

DatagramSocket define muitos métodos, mas só precisamos de três para o exemplo deste capítulo. Os dois primeiros são **send()** e **receive()**, mostrados abaixo:

```
void send(DatagramPacket pacote) throws IOException  
void receive(DatagramPacket pacote) throws IOException
```

O método **send()** envia um pacote para a porta e o endereço especificados por *pacote*. O método **receive()** espera um pacote ser recebido proveniente da porta especificada por *pacote*. Em seguida, insere o pacote recebido em *pacote*. O terceiro método que usaremos é **close()**, que fecha o soquete. A partir do JDK 7, **DatagramSocket** passou a implementar **AutoCloseable**, ou seja, agora um **DatagramSocket** pode ser gerenciado por um bloco **try-with-resources**.

DatagramPacket

DatagramPacket define vários construtores. Os dois que usaremos são:

```
DatagramPacket(byte[ ] dados, int tamanho)  
DatagramPacket(byte[ ] dados, int tamanho, InetAddress enderecoIp, int porta)
```

O primeiro construtor é usado para receber pacotes. Nele, *dados* especifica o buffer que receberá os dados e *tamanho* é a extensão dos dados, que não deve exceder o tamanho do buffer. O segundo construtor é usado para enviar pacotes. Nessa versão, *dados* especifica o buffer que contém os dados a serem enviados e *tamanho* indica o número de bytes do envio. Ela também especifica um endereço e uma porta de destino, que são usados para o **DatagramSocket** determinar para onde os dados do pacote serão enviados.

DatagramPacket define vários métodos que dão acesso ao endereço e número de porta de um pacote e aos dados brutos e seu tamanho. Em geral, os métodos **get**

são usados em pacotes que são recebidos e os métodos **set** em pacotes que serão enviados. Veja um resumo dos métodos definidos por **DatagramPacket**:

InetAddress getAddress()	Retorna o endereço da origem (para datagramas sendo recebidos) ou do destino (para datagramas sendo enviados).
byte[] getData()	Retorna o array byte que contém o buffer de dados. Mais usado na recuperação de dados do datagrama após ele ter sido recebido.
int getLength()	Retorna o número de bytes de dados contido no buffer. Pode ser menor do que o tamanho do array byte subjacente.
int getOffset()	Retorna o índice inicial dos dados do buffer.
int getPort()	Retorna o número de porta usado pelo host na outra extremidade da conexão.
void setData(byte[] <i>dados</i>)	Define os dados do pacote com <i>dados</i> , o deslocamento com zero e o tamanho com o número de bytes de <i>dados</i> .
void setData(byte[] <i>dados</i> , int <i>idc</i> , int <i>tamanho</i>)	Define os dados do pacote com <i>dados</i> , o deslocamento com <i>idc</i> e o tamanho com <i>tamanho</i> .
void setLength(int <i>tamanho</i>)	Define o tamanho do pacote com <i>tamanho</i> . Esse valor mais o deslocamento não devem exceder o tamanho do array byte subjacente.

Um exemplo de datagrama

O exemplo a seguir demonstra os datagramas com uma implementação muito simples de um servidor e um cliente. Nesse exemplo, o servidor lê strings inseridos a partir do teclado e os envia para o cliente. O cliente apenas espera até receber um pacote e então exibe o string. O processo continua até o string "stop" ser inserido. Nesse caso, tanto o cliente quanto o servidor são encerrados. No programa, os números de porta foram selecionados um pouco arbitrariamente porque são portas não usadas no sistema do autor. Portanto, você pode ter que selecionar portas diferentes. Se uma porta já estiver vinculada a um soquete, a tentativa de criar um soquete de datagrama falhará. Outra coisa: o uso de datagramas pode ser proibido por seu sistema, talvez por um firewall. Se for esse o caso, o exemplo a seguir não funcionará.

Esse exemplo é composto por duas classes. A primeira é **DGServer**, que é a classe que serve dados. Ela é mostrada aqui:

```
// Demonstra os datagramas - lado do servidor.

import java.net.*;
import java.io.*;

class DGServer {
    // Essas portas foram escolhidas arbitrariamente. Você
    // deve empregar portas não usadas em sua máquina.
```

```

public static int clientPort = 50000;
public static int serverPort = 50001;

public static DatagramSocket ds;

public static void dgServer() throws IOException {
    byte[] buffer;
    String str;

    BufferedReader conin = new BufferedReader(
        new InputStreamReader(System.in));

    System.out.println("Enter characters. Enter 'stop' to quit.");
    for(;;) {
        // lê um string a partir do teclado
        str = conin.readLine();

        // converte o string em um array de bytes para transmissão
        buffer = str.getBytes();

        // envia um novo pacote contendo o string
        ds.send(new DatagramPacket(buffer, buffer.length, ← Envia um pacote.
            InetAddress.getLocalHost(), clientPort));

        // encerra quando "stop" é inserida
        if(str.equals("stop")) {
            System.out.println("Server Quits.");
            return;
        }
    }
}

public static void main(String[] args) {
    ds = null;

    try {
        ds = new DatagramSocket(serverPort);
        dgServer();
    } catch(IOException exc) {
        System.out.println("Communication error: " + exc);
    } finally {
        if(ds != null) ds.close();
    }
}

```

Sempre que um novo string é inserido, um **DatagramPacket** é criado contendo o string como um array de **bytes**. Esse pacote é então enviado para o cliente via uma chamada a **send()**. O processo se repete até "stop" ser inserida.

O segundo arquivo, chamado **DGClient**, recebe os dados enviados pelo servidor. Ele é mostrado abaixo:

```
// Demonstra os datagramas - lado do cliente.

import java.net.*;
import java.io.*;

class DGClient {
    // Essa porta foi escolhida arbitrariamente. Você deve
    // empregar uma porta não usada em sua máquina.
    public static int clientPort = 50000;
    public static int buffer_size = 1024;

    public static DatagramSocket ds;

    public static void dgClient() throws IOException {
        String str;
        byte[] buffer = new byte[buffer_size];

        System.out.println("Receiving Data");
        for(;;) {
            // cria um novo pacote para receber os dados
            DatagramPacket p = new DatagramPacket(buffer, buffer.length);

            // espera um pacote
            ds.receive(p); ← Recebe um pacote.

            // converte buffer em String
            str = new String(p.getData(), 0, p.getLength());

            // exibe o string no cliente
            System.out.println(str);

            // encerra quando "stop" é recebida.
            if(str.equals("stop")) {
                System.out.println("Client Stopping.");
                break;
            }
        }
    }

    public static void main(String[] args) {
        ds = null;

        try {
            ds = new DatagramSocket(clientPort);
            dgClient();
        } catch(IOException exc) {
            System.out.println("Communication error: " + exc);
        } finally {
```

```
        if(ds != null) ds.close();  
    }  
}
```

Aqui, um novo **DatagramPacket** é criado para receber os dados. Em seguida, **receive()** é chamado. Ele espera até um pacote ser recebido. O buffer é então convertido em um string e exibido. O processo se repete até "stop" ser recebida.

O construtor de **DatagramSocket** restringe a execução desse exemplo a duas portas na máquina local. Para usar o programa, primeiro execute

```
| java DGClient
```

em uma janela; esse será o cliente. Em seguida, execute

```
| java DGServer
```

Esse será o servidor. Qualquer coisa que for digitada na janela do servidor será enviada para a janela do cliente após uma nova linha ser recebida. Para encerrar o programa, insira "stop".

Verificação do progresso

1. O método `receive()` da classe **DatagramSocket** usa um **DatagramPacket** como parâmetro. Para que esse método usa seu parâmetro?
 2. As classes **DatagramPacket** e **DatagramSocket** usam o protocolo TCP. Verdadeiro ou falso?

EXERCÍCIOS

1. Os **Sockets** foram projetados para enviar e receber dados usando o protocolo HTTP. Verdadeiro ou falso?
 2. Os **URLConnections** foram projetados para enviar e receber dados usando o protocolo HTTP. Verdadeiro ou falso?
 3. Quais são as duas informações que um **Socket** precisa ter para se conectar com outro computador na rede?
 4. Quando inserimos um URL usando o HTTP em um navegador Web, por que não somos solicitados a inserir um número de porta?
 5. Em que enviar dados usando datagramas difere, em relação às garantias de chegada, do envio de dados com um método que use o protocolo TCP?
 6. Modifique o programa **SocketDemo** para que receba qualquer número de nomes de domínio como argumentos na linha de comando e os envie para a porta

Respostas:

1. Os dados recebidos são carregados no **DatagramPacket**. Eles podem ser extraídos pelo método **getData()**.
 2. Falso. Elas usam o UDP.

"whois" do servidor **whois.internic.net**. Para cada argumento de nome de domínio, ele deve exibir os dados recebidos do soquete. Se não houver argumentos, não exibirá nada. Observe que você terá que fechar o soquete e criar um novo após receber cada resposta.

7. Crie um método **getIPSegments()** que use um nome de host como parâmetro e retorne um array de strings onde cada string (separados por pontos) contenha um segmento de um endereço IP numérico. Por exemplo, se o endereço IP do nome de host for 111.222.333.444, então o método deve retornar o array {"111". "222". "333". "444"}. Seu método deve funcionar para endereços IPv4 e IPv6 e lançar uma **UnknownHostException** se não conseguir encontrar o host. Crie um programa de teste para o método.
8. Crie um programa que exiba o nome de host e o endereço IP numérico de sua máquina host local.
9. Se você chamar um construtor de **URL** com o argumento "http://www.mhhe.com/index.html?a=Jonathon#start" e depois chamar o método **getFile()** de **URL**, o que será retornado, se algo for?
10. Modifique o programa **HttpURLConnectionDemo** para que só exiba um número indicando quantos campos existem no cabeçalho de resposta.
11. Modifique o exemplo **DGClient** do fim do capítulo para que **buffer_size** seja igual a 6 em vez de 1024. Que diferença, se houver alguma, isso faz?
12. No método **dgClient()** do exemplo **DGClient**, há um laço em que um novo **DatagramPacket** é construído para receber cada novo pacote de informações do emitente. Também funcionaria reutilizarmos o mesmo **DatagramPacket**? Isto é, suponhamos que reescrevêssemos o código para que um **DatagramPacket** fosse criado fora do laço e o pacote fosse passado repetidamente como argumento para o método **receive()**. O programa continuará funcionando corretamente?

Os utilitários de concorrência

PRINCIPAIS HABILIDADES E CONCEITOS

- A classe **Semaphore**
- A classe **CountDownLatch**
- A classe **CyclicBarrier**
- A classe **Exchanger**
- A classe **Phaser**
- Usar um **Executor**
- Usar **Callable** e **Future**
- Usar bloqueios
- O Framework Fork/Join

Desde o início, Java dá suporte interno ao multithreading e à sincronização. Por exemplo, novas threads podem ser criadas com a implementação de **Runnable** ou a extensão de **Thread**, a sincronização está disponível com o uso da palavra-chave **synchronized** e a comunicação entre threads é suportada pelos métodos **wait()** e **notify()** definidos por **Object**. Em geral, esse suporte interno ao multithreading foi uma das mais importantes inovações de Java e ainda é uma de suas maiores vantagens.

Em sua forma conceitualmente pura, o suporte original de Java ao multithreading não é ideal para todos os aplicativos – principalmente os que fazem amplo uso de múltiplas threads. Por exemplo, o suporte original ao multithreading não fornece vários recursos de alto nível, como os semáforos, os pools de threads e os gerenciadores de execução, que facilitam a criação de programas intensivamente concorrentes.

É importante explicar desde o início que muitos programas Java fazem uso do multithreading e, portanto, são “concorrentes”. Por exemplo, muitos applets usam o multithreading. Porém, como usado neste capítulo, o termo *programa concorrente* se refere a um programa que faz uso *amplo e integral* de threads sendo executadas correntemente. Um exemplo seria um programa que usasse threads separadas para calcular simultaneamente os resultados parciais de um cálculo maior. Outro seria um que coordenasse as atividades de várias threads, com cada thread tentando acessar

informações em um banco de dados. Nesse caso, acessos somente de leitura podem ser tratados de modo diferente dos que requerem recursos de leitura/gravação.

Para começar a tratar as necessidades de um programa concorrente, o JDK 5 adicionou os *utilitários de concorrência*, também conhecidos como *API de concorrência*. O conjunto original de utilitários de concorrência forneceu vários recursos que há muito tempo eram desejados por programadores que desenvolvem aplicativos concorrentes. Por exemplo, ele ofereceu sincronizadores (como o semáforo), pools de threads, gerenciadores de execução, bloqueios, várias coleções de concorrência e uma maneira otimizada de usar threads na obtenção de resultados computacionais.

Embora a API de concorrência original fosse por si só impressionante, ela foi expandida significativamente pelo JDK 7. Um desses acréscimos foi o *Framework Fork/Join*, o qual facilita a criação de certos tipos de programas que fazem uso de vários processadores (como os encontrados em sistemas *multicore*). Logo, ele otimiza o desenvolvimento de programas em que duas ou mais partes são executadas de maneira realmente simultânea (isto é, com execução realmente paralela). Como esperado, a execução paralela pode aumentar drasticamente a velocidade de certas operações.

A API de concorrência original era bem grande e o Framework Fork/Join aumenta bastante seu tamanho. Não é novidade que várias das questões que envolvem os utilitários de concorrência sejam muito complexas. Não faz parte do escopo deste livro a discussão de todos os seus aspectos. Mas é importante que todos os programadores tenham um conhecimento operacional geral da API de concorrência. Mesmo em programas que não sejam intensivamente paralelos, recursos como sincronizadores, threads chamáveis e executores são aplicáveis a uma ampla variedade de situações. Talvez o mais importante seja que, devido ao surgimento dos computadores *multicore*, soluções envolvendo o Framework Fork/Join serão mais comuns. Portanto, este capítulo apresentará uma visão geral dos utilitários de concorrência e mostrará vários exemplos de seu uso. Ele termina com um exame detalhado do Framework Fork/Join.

OS PACOTES DA API DE CONCORRÊNCIA

Os utilitários de concorrência ficam no pacote **java.util.concurrent** e em seus dois subpacotes: **java.util.concurrent.atomic** e **java.util.concurrent.locks**. Uma visão geral de seus conteúdos será dada aqui.

java.util.concurrent

O pacote **java.util.concurrent** define os recursos básicos que suportam as alternativas às abordagens internas de sincronização e comunicação entre threads. Ele define os recursos-chave a seguir:

- Sincronizadores
- Executores
- Coleções de concorrência
- O Framework Fork/Join

Os *sincronizadores* oferecem meios sofisticados de sincronização das interações entre várias threads. As classes sincronizadoras definidas por **java.util.concurrent** são:

Semaphore	Implementa o semáforo clássico.
CountDownLatch	Espera até um número especificado de eventos ter ocorrido.
CyclicBarrier	Permite que um grupo de threads aguarde em um ponto de execução predefinido.
Exchanger	Troca dados entre duas threads.
Phaser	Sincroniza threads que avançam por várias fases de uma operação.

Observe que cada sincronizador fornece uma solução para um tipo específico de problema de sincronização. Isso permite que cada sincronizador seja otimizado para o uso que se espera dele. No passado, esses tipos de objetos de sincronização tinham que ser construídos manualmente. A API de concorrência os padroniza e disponibiliza para todos os programadores de Java.

Os **executores** gerenciam a execução de threads. No topo da hierarquia de executores está a interface **Executor**, que é usada para iniciar uma thread. A interface **ExecutorService** estende **Executor** e fornece métodos que gerenciam a execução. Há três implementações de **ExecutorService**: **ThreadPoolExecutor**, **ScheduledThreadPoolExecutor** e **ForkJoinPool**. O pacote **java.util.concurrent** também define a classe utilitária **Executors**, com muitos métodos **static** que simplificam a criação de executores.

As interfaces **Future** e **Callable** também têm relação com os executores. Um objeto **Future** contém um valor que é retornado por uma thread após sua execução. Logo, seu valor é definido “no futuro”, quando a thread termina. **Callable** define uma thread que retorna um valor.

O pacote **java.util.concurrent** define várias classes de coleção de concorrência, inclusive **ConcurrentHashMap**, **ConcurrentLinkedQueue** e **CopyOnWriteArrayList**. Elas oferecem alternativas de concorrência às classes relacionadas definidas pelo Collections Framework em **java.util**.

O Framework *Fork/Join* dá suporte à programação paralela. Suas principais classes são **ForkJoinTask**, **ForkJoinPool**, **RecursiveTask** e **RecursiveAction**. Como mencionado, o Framework *Fork/Join* foi adicionado pelo JDK 7.

Por fim, para manipular melhor o tempo das threads, **java.util.concurrent** define a enumeração **TimeUnit**.

java.util.concurrent.atomic

O pacote **java.util.concurrent.atomic** facilita o uso de variáveis em um ambiente de concorrência. Ele fornece um meio de atualizarmos de forma eficiente o valor de uma variável sem usar bloqueios. Isso é feito com o uso de classes, como **AtomicInteger** e **AtomicLong**, e métodos, como **compareAndSet()**, **decrementAndGet()** e **getAndSet()**. Esses métodos são executados como uma operação individual que não pode ser interrompida.

java.util.concurrent.locks

O pacote **java.util.concurrent.locks** fornece uma alternativa ao uso de métodos sincronizados. O que sustenta essa alternativa é a interface **Lock**, que define o mecanismo básico de ganho e abandono de acesso a um objeto. Os métodos-chave são **lock()**, **tryLock()** e **unlock()**. A vantagem do uso desses métodos é um controle maior sobre a sincronização.

O resto deste capítulo fará um exame mais detalhado dos componentes da API de concorrência.

USANDO OBJETOS DE SINCRONIZAÇÃO

Uma parte importante da API de concorrência são seus objetos de sincronização que têm suporte nas classes **Semaphore**, **CountDownLatch**, **CyclicBarrier**, **Exchanger** e **Phaser**. Coletivamente, eles permitem o fácil tratamento de situações de sincronização que de outra forma seriam difíceis de resolver. Também são aplicáveis a uma ampla variedade de programas – até mesmo os que só contêm concorrência limitada. Como os objetos de sincronização são de interesse de quase todos os programadores de Java, examinaremos cada um aqui com algum nível de detalhe.

Semaphore

A classe **Semaphore** implementa um semáforo clássico. Um semáforo controla o acesso a um recurso compartilhado com o uso de um contador. Se o contador for maior do que zero, o acesso será permitido. Se for igual a zero, o acesso será negado. O que o contador conta são as *permissões* que concedem acesso ao recurso compartilhado. Logo, para acessar o recurso, a thread deve receber uma permissão do semáforo.

Em geral, para usar um semáforo, a thread que espera acesso ao recurso compartilhado tenta adquirir uma permissão. Se a contagem do semáforo for maior do que zero, a thread obterá a permissão, o que fará a contagem ser reduzida. Caso contrário, a thread será bloqueada até uma permissão poder ser obtida. Quando a thread não precisar mais acessar o recurso compartilhado, ele liberará a permissão e a contagem do semáforo aumentará. Se houver outra thread esperando uma permissão, ele a obterá nesse momento. A classe **Semaphore** da linguagem Java implementa esse mecanismo.

Semaphore tem os dois construtores a seguir:

```
Semaphore(int num)  
Semaphore(int num, boolean como)
```

Aqui, *num* especifica a contagem inicial de permissões. Logo, *num* especifica o número de threads que podem acessar um recurso compartilhado em um determinado momento. Se *num* for igual a um, só uma thread poderá acessar o recurso. Por padrão, as threads em espera recebem a permissão em uma ordem indefinida. Configurando *como* com **true**, você pode garantir que elas recebam a permissão na ordem em que solicitaram acesso.

Para adquirir uma permissão, chame o método **acquire()**, que tem estas duas formas:

```
void acquire() throws InterruptedException  
void acquire(int num) throws InterruptedException
```

A primeira forma adquire uma permissão. A segunda adquire *num* permissões. Se a permissão não puder ser concedida na hora da chamada, a thread chamadora ficará suspensa até ela ficar disponível.

Para liberar uma permissão, chame **release()**, que tem estas duas formas:

```
void release()  
void release(int num)
```

A primeira forma libera uma permissão. A segunda libera o número de permissões especificado por *num*.

Com o uso de um semáforo para controlar o acesso a um recurso, cada thread que quiser usar esse recurso deve chamar **acquire()** antes de acessá-lo. Quando a thread não precisar mais do recurso, deve chamar **release()**. Veja um exemplo que ilustra o uso de um semáforo:

```
// Um exemplo de semáforo simples.

import java.util.concurrent.*;

class SemDemo {

    public static void main(String[] args) {
        Semaphore sem = new Semaphore(1); ← Cria um semáforo que só tem 1 permissão.

        new IncThread(sem, "A");
        new DecThread(sem, "B");

    }
}

// Um recurso compartilhado.
class Shared {
    static int count = 0;
}

// Uma thread de execução que incrementa count.
class IncThread implements Runnable {
    String name;
    Semaphore sem;

    IncThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }

    public void run() {
        System.out.println("Starting " + name);

        try {
            // Primeiro, obtém uma permissão.
            System.out.println(name + " is waiting for a permit.");
            sem.acquire(); ← Espera até uma permissão poder ser adquirida.
            System.out.println(name + " gets a permit.");
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}
```

```

for(int i=0; i < 5; i++) { _____
    Shared.count++;
    System.out.println(name + ": " + Shared.count);

    // Permite uma mudança de contexto -- se possível.
    Thread.sleep(10);
} _____ Acessa o recurso
} catch (InterruptedException exc) {
    System.out.println(exc);
}

// Libera a permissão.
System.out.println(name + " releases the permit.");
sem.release(); ← Libera a permissão.
}
}

// Uma thread de execução que decrementa count.
class DecThread implements Runnable {
    String name;
    Semaphore sem;

    DecThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Starting " + name);

        try {
            // Primeiro, obtém uma permissão.
            System.out.println(name + " is waiting for a permit.");
            sem.acquire(); ← Espera até uma
            System.out.println(name + " gets a permit."); permissão poder
            ser adquirida.

            // Agora, acessa o recurso compartilhado.
            for(int i=0; i < 5; i++) { _____
                Shared.count--;
                System.out.println(name + ": " + Shared.count);

                // Permite uma mudança de contexto -- se possível.
                Thread.sleep(10);
} _____ Acessa o recurso
} catch (InterruptedException exc) {
    System.out.println(exc);
}

// Libera a permissão.
System.out.println(name + " releases the permit.");
}

```

```

    sem.release() ; ← Libera a permissão.
}
}

```

A saída do programa é mostrada aqui. (A ordem exata em que as threads são executadas pode variar.)

```

Starting A
A is waiting for a permit.
A gets a permit.
A: 1
Starting B
B is waiting for a permit.
A: 2
A: 3
A: 4
A: 5
A releases the permit.
B gets a permit.
B: 4
B: 3
B: 2
B: 1
B: 0
B releases the permit.

```

O programa usa um semáforo para controlar o acesso à variável **count**, que é uma variável estática da classe **Shared**. **Shared.count** é incrementada cinco vezes pelo método **run()** de **IncThread** e decrementada cinco vezes por **DecThread**. Para impedir que essas duas threads accessem **Shared.count** ao mesmo tempo, o acesso só é permitido depois que uma permissão é adquirida no semáforo de controle. Após o acesso terminar, a permissão é liberada. Dessa forma, as threads acessam **Shared.count** uma de cada vez, como a saída mostra.

Tanto em **IncThread** quanto em **DecThread**, observe a chamada a **sleep()** dentro de **run()**. Ela é usada para “comprovarmos” se os acessos a **Shared.count** estão sendo sincronizados pelo semáforo. Em **run()**, a chamada a **sleep()** faz a thread chamadora pausar entre cada acesso a **Shared.count**. Normalmente isso permitiria que a segunda thread fosse executada. No entanto, devido ao semáforo, a segunda thread deve esperar até a primeira liberar a permissão, o que só acontece após o término de todos os acessos da primeira thread. Logo, primeiro **Shared.count** é incrementada cinco vezes por **IncThread** e então decrementada cinco vezes por **DecThread**. Os incrementos e decrementos *não* são intercalados.

Sem o uso do semáforo, os acessos a **Shared.count** pelas duas threads teriam ocorrido simultaneamente e os incrementos e decrementos seriam intercalados. Para confirmar isso, tente desativar as chamadas a **acquire()** e **release()** com um comentário. Quando você executar o programa, verá que o acesso à variável **Shared.count** não é mais sincronizado e que cada thread a acessa assim que obtém uma parcela do tempo.

CountDownLatch

Às vezes, podemos querer que uma thread espere até que um ou mais eventos ocorram. (Aqui, estamos usando o termo *evento* para representar alguma ação do progra-

ma, como a conclusão de uma tarefa por uma thread separada.) Para lidar com essa situação, a API de concorrência fornece **CountDownLatch**. Um **CountDownLatch** é criado inicialmente com uma contagem do número de eventos que devem ocorrer antes de a tranca ser liberada. Sempre que um evento ocorre, a contagem é decrementada. Quando a contagem alcança zero, a tranca é aberta.

CountDownLatch tem o seguinte construtor:

```
CountDownLatch(int num)
```

Aqui, *num* especifica o número de eventos que devem ocorrer para a tranca ser aberta.

Para esperar a tranca, a thread chama **await()**, que tem as formas mostradas abaixo:

```
void await() throws InterruptedException
boolean await(long espera, TimeUnit tu) throws InterruptedException
```

A primeira forma espera até a contagem associada ao **CountDownLatch** chamador alcançar zero. A segunda forma só espera durante o período especificado por *espera*. As unidades representadas por *espera* são especificadas por *tu*, que é um objeto da enumeração **TimeUnit**. (**TimeUnit** será descrita posteriormente neste capítulo.) Ela retorna **false** se o limite de tempo for alcançado e **true** se a contagem regressiva alcançar zero.

Para sinalizar um evento, chame o método **countDown()**, mostrado a seguir:

```
void countDown()
```

Cada chamada a **countDown()** decrementa a contagem associada ao objeto chamador.

O próximo programa demonstra **CountDownLatch**. Ele cria uma tranca que requer que cinco eventos ocorram antes de ser aberta.

```
// Um exemplo de CountDownLatch.

import java.util.concurrent.CountDownLatch;

class CDLDemo {
    public static void main(String[] args) {
        CountDownLatch cdl = new CountDownLatch(5); ← Cria um CountDownLatch
                                                       que espera 5 eventos.
        System.out.println("Starting");

        new MyThread(cdl);

        try {
            cdl.await(); ← Espera até a contagem
                           regressiva terminar.
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        System.out.println("Done");
    }
}

class MyThread implements Runnable {
    CountDownLatch latch;

    MyThread(CountDownLatch c) {
```

```

    latch = c;
    new Thread(this).start();
}

public void run() {
    for(int i = 0; i<5; i++) {
        System.out.println(i);
        latch.countDown(); // decrementa a contagem ← Decrementa a contagem a
                           // cada passagem pelo laço.
    }
}

```

A saída produzida pelo programa é:

```

Starting
0
1
2
3
4
Done

```

Dentro de **main()**, um **CountDownLatch** chamado **cdl** é criado com contagem inicial igual a cinco. Em seguida, é criada uma instância de **MyThread**, que começa a execução de uma nova thread. Observe que **cdl** é passado como parâmetro para o construtor de **MyThread** e armazenado na variável de instância **latch**. A thread principal chama então **await()** em **cdl**, o que faz sua execução pausar até a contagem de **cdl** ser decrementada cinco vezes.

Dentro do método **run()** de **MyThread**, é criado um laço que itera cinco vezes. A cada iteração, o método **countDown()** é chamado em **latch**, que referencia **cdl** em **main()**. Após a quinta iteração, a tranca é aberta, o que permite que a thread principal seja retomada.

CountDownLatch é um objeto de sincronização poderoso, porém fácil de usar, que é apropriado sempre que uma thread deve esperar uma ou mais ações ocorrerem.

Verificação do progresso

- Para obter uma permissão em um **Semaphore**, chame seu método _____, e para liberar a permissão, chame o método _____.
- Uma thread pode obter no máximo uma permissão de cada vez em um **Semaphore**. Verdadeiro ou falso?
- Um **CountDownLatch** é útil quando queremos que uma thread espere até _____.
- Para forçarmos uma thread a esperar até a contagem de um **CountDownLatch** alcançar 0, ela deve chamar o método _____ de **CountDownLatch**.

Respostas:

- acquire()** e **release()**
- Falso. Uma thread pode adquirir qualquer número de permissões (supondo que elas estejam disponíveis) em um semáforo chamando **acquire(*n*)**, em que *n* é o número de permissões desejado.
- que um certo número de eventos ocorram
- await()**

CyclicBarrier

Uma situação comum na programação concorrente ocorre quando um conjunto de duas ou mais threads deve esperar em um ponto predeterminado da execução, chamado *barreira*, até que todas as threads do conjunto o tenham alcançado. Para lidar com essa situação, a API de concorrência fornece a classe **CyclicBarrier**. Ela permite a definição de um objeto de sincronização que fica suspenso até o número especificado de threads ter alcançado a barreira.

CyclicBarrier tem os dois construtores a seguir:

```
CyclicBarrier(int numThreads)
CyclicBarrier(int numThreads, Runnable ação)
```

Aqui, *numThreads* especifica o número de threads que devem alcançar a barreira antes de a execução continuar. Na segunda forma, *ação* especifica uma thread que será executada quando a barreira for alcançada por todas as threads.

Este é o procedimento geral que você seguirá para usar **CyclicBarrier**: primeiro, crie um objeto **CyclicBarrier**, especificando o número de threads que irá esperar. Em seguida, a cada thread que alcançar a barreira, faça-a chamar **await()** nesse objeto. Uma vez que o número especificado de threads tiver alcançado a barreira, **await()** retornará e a execução será retomada. Se você também tiver especificado uma ação, essa thread será executada.

O método **await()** tem as duas formas a seguir:

```
int await() throws InterruptedException, BrokenBarrierException
int await(long espera, TimeUnit tu)
    throws InterruptedException, BrokenBarrierException, TimeoutException
```

A primeira forma espera até todas as threads terem alcançado o ponto da barreira. A segunda só espera durante o período de tempo especificado por *espera*. As unidades representadas por *espera* são especificadas por *tu*. Essa versão lança uma **TimeOutException** quando o período de espera expira. As duas formas retornam um valor que indica a ordem de chegada das threads no ponto da barreira. A primeira thread retorna um valor igual ao número de threads em espera menos um. A última retorna zero.

Vejamos um exemplo que ilustra **CyclicBarrier**. Ele espera até um conjunto de três threads alcançarem a barreira. Quando isso ocorre, a thread especificada por **BarAction** é executada.

```
// Um exemplo de CyclicBarrier.

import java.util.concurrent.*;
class BarDemo {
    public static void main(String[] args) {
        CyclicBarrier cb = new CyclicBarrier(3, new BarAction());
        System.out.println("Starting");
    }
}

class BarAction implements Runnable {
    public void run() {
        System.out.println("Bar crossed!");
    }
}
```

Cria um **CyclicBarrier** para três threads.

```

    new MyThread(cb, "A");
    new MyThread(cb, "B");
    new MyThread(cb, "C");

}

}

// Uma thread de execução que usa um CyclicBarrier.

class MyThread implements Runnable {
    CyclicBarrier cbar;
    String name;

    MyThread(CyclicBarrier c, String n) {
        cbar = c;
        name = n;
        new Thread(this).start();
    }

    public void run() {
        System.out.println(name);

        try {
            cbar.await(); ← Espera todas as threads chamarem await().
        } catch (BrokenBarrierException exc) {
            System.out.println(exc);
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
    }
}

// Um objeto dessa classe é chamado quando o
// CyclicBarrier termina.
class BarAction implements Runnable {
    public void run() {
        System.out.println("Barrier Reached!");
    }
}

```

A saída é mostrada aqui. (A ordem exata em que as threads são executadas pode variar.)

```

Starting
A
B
C
Barrier Reached!

```

O **CyclicBarrier** pode ser reutilizado porque ele libera as threads em espera sempre que o número especificado de threads chama **await()**. Por exemplo, se você alterar **main()** no programa anterior de modo que fique assim:

```
public static void main(String[] args) {
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

    System.out.println("Starting");

    new MyThread(cb, "A");
    new MyThread(cb, "B");
    new MyThread(cb, "C");
    new MyThread(cb, "X");
    new MyThread(cb, "Y");
    new MyThread(cb, "Z");

}
```

a saída a seguir será produzida. (A ordem exata em que as threads são executadas pode variar.)

```
Starting
A
B
C
Barrier Reached!
X
Y
Z
Barrier Reached!
```

Como o exemplo anterior mostra, **CyclicBarrier** oferece uma solução otimizada para o que, de outra forma, seria um problema complicado.

Exchanger

Talvez a mais interessante das classes de sincronização seja **Exchanger**. Ela foi projetada para simplificar a troca de dados entre duas threads. A operação de uma **Exchanger** é surpreendentemente simples: ela apenas espera até duas threads chamarem seu método **exchange()** e troca os dados fornecidos pelas threads quando isso ocorre. Esse mecanismo é ao mesmo tempo elegante e fácil de usar. É fácil imaginar usos para **Exchanger**. Por exemplo, uma thread poderia preparar um buffer para receber informações provenientes de uma conexão de rede. Outra thread preencheria esse buffer com as informações a partir da conexão. As duas threads trabalhariam em conjunto de modo que uma troca fosse feita sempre que um novo buffer fosse necessário.

Exchanger é uma classe genérica declarada como mostrado abaixo:

```
Exchanger<V>
```

Aqui, **V** especifica o tipo dos dados que estão sendo trocados.

O único método definido por **Exchanger** é **exchange()**, que tem as duas formas abaixo:

```
V exchange(V objTroca) throws InterruptedException
V exchange(V objTroca, long espera, TimeUnit tu)
    throws InterruptedException, TimeOutException
```

Aqui, *objTroca* é uma referência aos dados a serem trocados. Os dados recebidos da outra thread são retornados. A segunda forma de **exchange()** permite que um tempo limite seja especificado. O importante em **exchange()** é que ele não é bem-sucedido até ser chamado no mesmo objeto **Exchanger** por duas threads distintas. Logo, **exchange()** sincroniza a troca de dados.

Vejamos um exemplo que demonstra **Exchanger**. Ele cria duas threads que trocam strings uma com outra. Uma thread cria uma referência a um string vazio. A segunda thread cria um string inicializado. Essas referências de string são então trocadas. O resultado final é que a primeira thread troca um string vazio por um cheio.

```
// Um exemplo de Exchanger.

import java.util.concurrent.Exchanger;

class ExgrDemo {
    public static void main(String[] args) {
        Exchanger<String> exgr = new Exchanger<String>(); ← Criar um Exchanger
                                                                para strings.

        new UseString(exgr);
        new MakeString(exgr);
    }
}

// Uma thread que constrói um string inicializado.
class MakeString implements Runnable {
    Exchanger<String> ex;
    String str;

    MakeString(Exchanger<String> c) {
        ex = c;
        str = new String();

        new Thread(this).start();
    }

    public void run() {
        char ch = 'A';

        for(int i = 0; i < 3; i++) {
```

```
// Cria um string.  
for(int j = 0; j < 5; j++)  
    str += ch++;  
  
try {  
    // Troca um string inicializado por um vazio.  
    str = ex.exchange(str); ←————— Troca um string inicializado  
} catch(InterruptedException exc) {  
    System.out.println(exc);  
}  
}  
}  
}  
  
// Uma thread que usa um string.  
class UseString implements Runnable {  
    Exchanger<String> ex;  
    String str;  
    UseString(Exchanger<String> c) {  
        ex = c;  
        new Thread(this).start();  
    }  
  
    public void run() {  
  
        for(int i=0; i < 3; i++) {  
            try {  
                // Troca um string vazio por um inicializado.  
                str = ex.exchange(new String()); ←————— Troca um string vazio por  
                System.out.println("Got: " + str);  
            } catch(InterruptedException exc) {  
                System.out.println(exc);  
            }  
        }  
    }  
}
```

Esta é a saída produzida pelo programa:

```
Got: ABCDE  
Got: FGHIJ  
Got: KLMNO
```

No programa, o método **main()** cria um **Exchanger** para strings. Esse objeto é então usado para sincronizar a troca de strings entre as classes **MakeString** e **UseString**. A classe **MakeString** inicializa um string. **UseString** troca um string vazio por um cheio. Em seguida, o programa exibe o conteúdo do string recém-construído. A troca de strings é sincronizada pelo método **exchange()**, que é chamado pelo método **run()** das duas classes.

Verificação do progresso

1. Um **CyclicBarrier** é útil em uma situação em que _____.
2. Um **Exchanger** fornece uma maneira fácil de _____.

Phaser

O JDK 7 adicionou uma nova classe de sincronização chamada **Phaser**. Sua finalidade básica é permitir a sincronização de threads que representem uma ou mais fases de uma atividade. Por exemplo, você poderia ter um conjunto de threads que implementassem três fases de um aplicativo de processamento de pedidos. Na primeira fase, threads distintas são usadas para validar informações do cliente, verificar o estoque e confirmar o preço. Quando essa fase termina, a segunda fase tem duas threads que calculam os custos de entrega e todos os impostos aplicáveis. Depois disso, uma fase final confirma o pagamento e determina o tempo estimado de entrega. No passado, a sincronização das várias threads que compõem esse cenário exigiria algum trabalho de sua parte. Com a inclusão de **Phaser**, o processo é muito mais fácil.

Para começar, ajuda saber que um **Phaser** funciona de maneira semelhante a um **CyclicBarrier**, descrito anteriormente, exceto por dar suporte a várias fases. Como resultado, **Phaser** permite a definição de um objeto de sincronização que espera até uma fase específica terminar. Então ele avança para a próxima fase, esperando novamente até que ela termine. É importante saber que **Phaser** também pode ser usada para sincronizar uma única fase. Nesse aspecto, age de modo muito parecido a um **CyclicBarrier**. No entanto, seu principal uso é na sincronização de várias fases.

Phaser define quatro construtores. Aqui estão os dois usados nesta seção:

```
Phaser()
Phaser(int numParticipantes)
```

O primeiro cria um phaser que tem contagem de registros igual a zero. O segundo configura a contagem de registros com *numParticipantes*. O termo *participante* se refere aos objetos que se registram em um phaser. Embora geralmente haja uma correspondência um para um entre o número de registrados e o número de threads sincronizadas, isso não é necessário. Nos dois casos, a fase atual é zero. Isto é, quando um **Phaser** é criado, inicialmente ele está na fase zero.

Em geral, é assim que **Phaser** é usada: primeiro, crie uma nova instância de **Phaser**. Em seguida, registre um ou mais participantes no phaser, chamando **register()** ou especificando o número de participantes no construtor. Para cada participante registrado, faça o phaser esperar até que todos os participantes registrados completem

Respostas:

1. todas as threads de um conjunto de threads devem alcançar um certo ponto antes de poder ultrapassá-lo.
2. trocar dados entre duas threads.

uma fase. Um participante sinaliza isso chamando um dos vários métodos fornecidos por **Phaser**, como **arrive()** ou **arriveAndAwaitAdvance()**. Após todos os participantes chegarem, a fase estará concluída e o phaser poderá passar para a próxima fase (se houver uma) ou terminar. As seções a seguir explicam o processo em detalhes.

Para registrar participantes após um **Phaser** ser construído, chame o método **register()**. Podemos vê-lo abaixo:

```
int register()
```

Ele retorna o número da fase para a qual fez o registro.

Para sinalizar que um participante concluiu uma fase, você deve chamar **arrive()** ou alguma variação de **arrive()**. Quando o número de chegadas for igual ao número de participantes registrados, a fase estará concluída e o **Phaser** passará para a próxima fase (se houver uma). O método **arrive()** tem esta forma geral:

```
int arrive()
```

Esse método sinaliza que um participante (normalmente uma thread) concluiu alguma tarefa (ou parte dela). Ele retorna o número da fase atual. Se o phaser terminar, um valor negativo será retornado. O método **arrive()** não suspende a execução da thread chamadora. Ou seja, não espera a fase ser concluída. Ele só deve ser chamado por um participante registrado.

Se quiser indicar a conclusão de uma fase e esperar até que os outros participantes registrados também a concluam, use o método **arriveAndAwaitAdvance()**, mostrado aqui:

```
int arriveAndAwaitAdvance()
```

Ele espera até todos os participantes chegarem e retorna o número da próxima fase ou um valor negativo se o phaser terminar. Esse método só deve ser chamado por um participante registrado.

Uma thread pode chegar e então anular seu registro chamando o método **arriveAndDeregister()**, mostrado abaixo:

```
int arriveAndDeregister()
```

Ele retorna o número da fase atual ou um valor negativo se o phaser terminar. Porém, não espera até a fase ser concluída. Esse método só deve ser chamado por um participante registrado.

Para obter o número da fase atual, chame o método **getPhase()**, mostrado aqui:

```
final int getPhase()
```

Quando um **Phaser** é criado, a primeira fase é 0, a segunda 1, a terceira 2, e assim por diante. Um valor negativo é retornado quando o **Phaser** chamador termina. Para determinar se um **Phaser** terminou, use o método **isTerminated()**. Ele retorna **true** se o phaser tiver terminado; caso contrário, retorna **false**.

Vejamos um exemplo que mostra **Phaser** em ação. Ele cria três threads, cada uma com três fases, e usa um **Phaser** para sincronizar cada fase.

```
// Um exemplo de Phaser.

import java.util.concurrent.*;

class PhaserDemo {
    public static void main(String[] args) {
        Phaser phsr = new Phaser(1); Cria um Phaser com um participante inicial para a thread principal.
        int curPhase;

        System.out.println("Starting");

        new MyThread(phsr, "A");
        new MyThread(phsr, "B");
        new MyThread(phsr, "C");

        // Espera todas as threads terminarem a fase um.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance(); A thread principal chega e espera cada fase ser concluída.
        System.out.println("Phase " + curPhase + " Complete");

        // Espera todos as threads terminarem a fase dois.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance(); A thread principal chega e espera cada fase ser concluída.
        System.out.println("Phase " + curPhase + " Complete");

        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance(); A thread principal chega e espera cada fase ser concluída.
        System.out.println("Phase " + curPhase + " Complete");

        // Anula o registro da thread principal.
        phsr.arriveAndDeregister();

        if(phsr.isTerminated())
            System.out.println("The Phaser is terminated");
    }
}

// Uma thread de execução que usa um Phaser.
class MyThread implements Runnable {
    Phaser phsr;
    String name;

    MyThread(Phaser p, String n) {
        phsr = p;
        name = n;
        phsr.register(); Registra essa thread.
        new Thread(this).start();
    }
}
```

```

public void run() {

    System.out.println("Thread " + name + " Beginning Phase One");
    phsr.arriveAndAwaitAdvance(); // Sinaliza a chegada. ←

    // Faz uma pausa para evitar uma saída confusa. Isso é feito apenas a título
    // de ilustração. Não é necessário para a operação apropriada do phaser.
    try {
        Thread.sleep(10);
    } catch(InterruptedException e) {
        System.out.println(e);
    }

    System.out.println("Thread " + name + " Beginning Phase Two");
    phsr.arriveAndAwaitAdvance(); // Sinaliza a chegada. ←

    // Faz uma pausa para evitar uma saída confusa. Isso é feito apenas a título
    // de ilustração. Não é necessário para a operação apropriada do phaser.
    try {
        Thread.sleep(10);
    } catch(InterruptedException e) {
        System.out.println(e);
    }

    System.out.println("Thread " + name + " Beginning Phase Three");
    phsr.arriveAndDeregister(); // Sinaliza a chegada e anula o registro. ← terminar.
}

}

```

Anula o
registro ao
terminar.

A saída é esta:

```

Starting
Thread A Beginning Phase One
Thread C Beginning Phase One
Thread B Beginning Phase One
Phase 0 Complete
Thread B Beginning Phase Two
Thread C Beginning Phase Two
Thread A Beginning Phase Two
Phase 1 Complete
Thread C Beginning Phase Three
Thread B Beginning Phase Three
Thread A Beginning Phase Three
Phase 2 Complete
The Phaser is terminated

```

Examinemos detalhadamente as principais seções do programa. Primeiro, em **main()**, um **Phaser** chamado **phsr** é criado com contagem de participantes inicial

igual a 1 (que corresponde à thread principal). Três threads são então iniciadas com a criação de três objetos **MyThread**. Observe que **MyThread** recebe uma referência a **phsr** (o phaser). Os objetos **MyThread** usam esse phaser para sincronizar suas atividades. Em seguida, **main()** chama **getPhase()** para obter o número da fase atual (que inicialmente é zero) e então chama **arriveAndAwaitAdvance()**. Esse método faz **main()** ficar suspenso até a fase zero ser concluída, o que só ocorrerá quando todos os **MyThreads** chegarem. Quando isso ocorrer, **main()** voltará a ser executado, exibirá que a fase zero terminou e passará para a fase dois. Esse processo se repete até as três fases terminarem. Então **main()** chama **arriveAndDeregister()**. Nesse momento, os três **MyThreads** também têm seu registro anulado. Como essa ação resulta em não haver participantes registrados quando o phaser passa para a próxima fase, o phaser é encerrado.

Agora, examinemos **MyThread**. Primeiro, observe que o construtor recebe uma referência ao phaser que será usado e então registra a nova thread como participante desse phaser. Logo, cada novo objeto **MyThread** passa a ser um participante registrado no phaser que foi passado. Observe também que cada thread tem três fases. Nesse exemplo, cada fase é composta por um espaço reservado que apenas exibe o nome da thread e o que ela está fazendo. É claro que, em um código do mundo real, a thread executaria ações mais significativas. Entre as duas primeiras fases, a thread chama **arriveAndAwaitAdvance()**. Portanto, cada thread espera até todas as threads concluírem a fase (e a thread principal estar pronta). Após todas as threads chegarem (inclusive a thread principal), o phaser passa para a próxima fase. Depois da terceira fase, cada thread anula seu registro com uma chamada a **arriveAndDeregister()**. Como os comentários de **MyThread** explicam, as chamadas a **sleep()** são usadas com fins ilustrativos para assegurar que a saída não fique desorganizada devido ao multithreading. Elas não são necessárias para o phaser funcionar adequadamente. Se você removê-las, a saída pode ficar um pouco confusa, mas as fases continuarão sincronizadas corretamente.

Mais uma coisa: embora o exemplo anterior tenha usado três threads do mesmo tipo, isso não é obrigatório. Cada participante usuário de um phaser pode ser exclusivo, com cada um executando uma tarefa separada.

É possível assumir o controle do que acontecerá quando ocorrer um avanço do phaser. Para fazer isso, você deve sobrepor o método **onAdvance()**. Esse método é chamado pelo tempo de execução quando um **Phaser** avança de uma fase para outra. Ele é mostrado abaixo:

```
protected boolean onAdvance(int fase, int numParticipantes)
```

Aqui, *fase* conterá o número da fase atual antes do incremento, e *numParticipantes* o número de participantes registrados. Para o phaser ser encerrado, **onAdvance()** deve retornar **true**. Para que continue ativo, deve retornar **false**. A versão padrão de **onAdvance()** retorna **true** (encerrando o phaser) quando não há participantes registrados. Como regra geral, sua sobreposição também deve seguir essa prática. Uma das razões para a sobreposição de **onAdvance()** é permitirmos que o phaser execute um número específico de fases e então pare. Você verá um exemplo dessa abordagem na seção Tente isto 27-1.

TENTE ISTO 27-1 Demonstra o método onAdvance() de Phaser

`StarPhaserDemo.java`

Este projeto demonstra o uso do método **onAdvance()** definido por **Phaser**. Ele desenvolve um programa que usa várias threads e um **Phaser** para desenhar um retângulo de asteriscos de dimensões 4 por 3 no console. Quatro threads são usadas no desenho do retângulo e suas ações são coordenadas pelo phaser. Em cada fase, cada thread desenha um asterisco e estão esperando a próxima fase. O programa usa o método **onAdvance()** de **Phaser** para exibir uma nova linha após a conclusão de cada fase e para encerrar o phaser após o retângulo ter sido desenhado. Logo, o phaser será responsável pela passagem para a próxima linha e o início da nova fase.

PASSO A PASSO

- Crie um novo arquivo chamado **StarPhaserDemo.java** e insira o código a seguir:

```
import java.util.concurrent.Phaser;

public class StarPhaserDemo {
    public static void main(String args[]) {
        Phaser phsr = new NewlinePhaser(4,3);

        new StarThread(phsr);
        new StarThread(phsr);
        new StarThread(phsr);
        new StarThread(phsr);
    }
}
```

Observe que o método **main()** só precisa criar o **NewlinePhaser** e as quatro threads. As threads vão tratar o desenho do retângulo.

- Agora, adicione o código da classe **NewlinePhaser**:

```
class NewlinePhaser extends Phaser {
    int numPhases;

    public NewlinePhaser(int numParties, int phases) {
        super(numParties);
        numPhases = phases;
    }

    public boolean onAdvance(int phase, int numParties) {
        System.out.println(); // exibe uma nova linha
        return phase == numPhases-1; // para após numPhases
    }
}
```

Como você pode ver, **NewlinePhaser** faz duas coisas. Em primeiro lugar, controla o número de fases que permitirá que ocorram. Em segundo lugar,

sobrepõe o método **onAdvance()**. Essa sobreposição exibe uma nova linha após cada fase e interrompe o phaser após a ocorrência do número de fases especificado. (Lembre-se, o **Phaser** é encerrado quando **onAdvance()** retorna **true**).

3. Para concluir, adicione o código da classe **StarThread**:

```
class StarThread implements Runnable {
    Phaser phsr;

    StarThread(Phaser p) {
        phsr = p;
        new Thread(this).start();
    }

    public void run() {
        while (!phsr.isTerminated()) {
            System.out.print('*');
            phsr.arriveAndAwaitAdvance();
        }
    }
}
```

Como se pode ver, a thread exibe repetidamente um asterisco e então espera o fim da fase. Ela se mantém executando essa atividade até o phaser terminar.

4. Aqui está o programa completo:

```
import java.util.concurrent.Phaser;

public class StarPhaserDemo {
    public static void main(String args[]) {
        Phaser phsr = new NewlinePhaser(4,3);

        new StarThread(phsr);
        new StarThread(phsr);
        new StarThread(phsr);
        new StarThread(phsr);
    }
}

class NewlinePhaser extends Phaser {
    int numPhases;

    public NewlinePhaser(int numParties, int phases) {
        super(numParties);
        numPhases = phases;
    }

    public boolean onAdvance(int phase, int numParties) {
        System.out.println(); // exibe uma nova linha
        return phase == numPhases-1; // para após numPhases
    }
}
```

```

class StarThread implements Runnable {
    Phaser phsr;

    StarThread(Phaser p) {
        phsr = p;
        new Thread(this).start();
    }

    public void run() {
        while (!phsr.isTerminated()) {
            System.out.print('*');
            phsr.arriveAndAwaitAdvance();
        }
    }
}

```

5. Quando você executar o programa, verá a saída a seguir:

```

*****
*****
*****

```

Como a saída confirma, três linhas com quatro asteriscos foram desenhadas, com cada asterisco de uma linha sendo exibido por uma thread separada. As ações das threads foram coordenadas pelo phaser.

Verificação do progresso

1. Um **Phaser** usado na sincronização de apenas uma fase é muito semelhante a um _____.
2. Para uma thread sinalizar para um **Phaser** que concluiu uma fase e que precisa esperar que as outras threads a concluem, ela deve chamar o método _____ de **Phaser**.
3. Se quiser que um **Phaser** execute algum código na conclusão de uma fase antes de a próxima fase começar, você deve _____.

USANDO UM EXECUTOR

A API de concorrência fornece um recurso chamado *executor* que inicia e controla a execução de threads. Como tal, um executor oferece uma alternativa ao gerenciamento de threads com a classe **Thread**.

Respostas:

1. **CyclicBarrier**
2. **arriveAndAwaitAdvance()**
3. criar uma subclasse de **Phaser** e sobrepor o método **onAdvance()** herdado por ela para que contenha o código que deseja executar

O que dá suporte ao executor é a interface **Executor**. Ela define o método a seguir:

```
void execute(Runnable tarefa)
```

O código especificado por *tarefa* é executado.

A interface **ExecutorService** estende **Executor** adicionando métodos que ajudam a gerenciar e controlar a execução de threads. Por exemplo, **ExecutorService** define o método **shutdown()**, mostrado aqui, que encerra o **ExecutorService** chamador:

```
void shutdown()
```

ExecutorService também define métodos para a execução de threads que retornam resultados, para a execução de um conjunto de threads e para a definição do status de encerramento.

Também é definida a interface **ScheduledExecutorService**, que estende **ExecutorService** para dar suporte ao agendamento de threads.

A API de concorrência define três classes de executores predefinidas: **ThreadPoolExecutor**, **ScheduledThreadPoolExecutor** e **ForkJoinPool**. **ThreadPoolExecutor** implementa as interfaces **Executor** e **ExecutorService** e dá suporte a um pool de threads gerenciado. **ScheduledThreadPoolExecutor** também implementa a interface **ScheduledExecutorService** para permitir que um pool de threads seja agendado. **ForkJoinPool** implementa as interfaces **Executor** e **ExecutorService** e é usada pelo Framework Fork/Join. Ela será descrita posteriormente neste capítulo.

Um pool de threads fornece um conjunto de threads para ser usado na execução de várias tarefas. Em vez de cada tarefa usar sua própria thread, as threads do pool são usadas. Isso reduz a sobrecarga associada à criação de muitas threads separadas. Embora você possa usar **ThreadPoolExecutor** e **ScheduledThreadPoolExecutor** diretamente, o mais provável é que queira obter um executor chamando um dos métodos **static** factory a seguir definidos pela classe utilitária **Executors**. Vejamos alguns exemplos:

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int numThreads)
static ScheduledExecutorService newScheduledThreadPool(int numThreads)
```

O método **newCachedThreadPool()** cria um pool que adiciona threads conforme necessário, mas as reutiliza se possível. O método **newFixedThreadPool()** cria um pool composto por um número especificado de threads. O método **newScheduledThreadPool()** cria um pool que dá suporte ao agendamento de threads. Todos retornam uma referência a um **ExecutorService** que pode ser usado no gerenciamento do pool.

Um exemplo de executor simples

Antes de avançar, será útil ver um exemplo simples que usa um executor. O programa a seguir cria um pool de threads fixo contendo duas threads. Em seguida, ele usa esse pool para executar quatro tarefas. Logo, quatro tarefas compartilham as duas threads que estão no pool. Após as tarefas serem concluídas, o pool é encerrado e o programa termina.

```
// Um exemplo simples que usa um Executor.

import java.util.concurrent.*;

class SimpExec {
    public static void main(String[] args) {
        CountDownLatch cdl = new CountDownLatch(5);
        CountDownLatch cdl2 = new CountDownLatch(5);
        CountDownLatch cdl3 = new CountDownLatch(5);
        CountDownLatch cdl4 = new CountDownLatch(5);
        ExecutorService es = Executors.newFixedThreadPool(2); ← Cria um executor
                                                    que tem duas
                                                    threads.

        System.out.println("Starting");

        // Inicia as threads.
        es.execute(new MyThread(cdl, "A"));
        es.execute(new MyThread(cdl2, "B"));
        es.execute(new MyThread(cdl3, "C"));
        es.execute(new MyThread(cdl4, "D")); ← Executa quatro threads
                                            usando o pool de threads.

        try {
            cdl.await();
            cdl2.await();
            cdl3.await();
            cdl4.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        es.shutdown(); ← Encerra o executor.
        System.out.println("Done");
    }
}

class MyThread implements Runnable {
    String name;
    CountDownLatch latch;

    MyThread(CountDownLatch c, String n) {
        latch = c;
        name = n;
    }

    public void run() {
        for(int i = 0; i < 5; i++) {
            System.out.println(name + ": " + i);
            latch.countDown();
        }
    }
}
```

A saída do programa é mostrada aqui. (A ordem exata em que as threads são executadas pode variar.)

```
Starting
```

```
B: 0
A: 0
B: 1
A: 1
A: 2
B: 2
B: 3
B: 4
A: 3
C: 0
C: 1
A: 4
C: 2
D: 0
D: 1
C: 3
D: 2
D: 3
C: 4
D: 4
Done
```

Como a saída mostra, mesmo com o pool tendo apenas duas threads, todas as quatro tarefas são executadas. No entanto, só duas podem ser executadas ao mesmo tempo. As outras devem esperar até uma das threads do pool estar disponível para uso.

A chamada a `shutdown()` é importante. Se não estivesse presente no programa, ele não terminaria, porque o executor permaneceria ativo. Para confirmar isso, desative a chamada a `shutdown()` com um comentário e observe o resultado.

USANDO Callable E Future

Um recurso muito interessante da API de concorrência é a interface **Callable**. Essa interface representa uma thread que retorna um valor. O aplicativo pode usar objetos **Callable** para calcular resultados que sejam retornados para a thread chama-dora. Esse é um mecanismo poderoso porque facilita a codificação de muitos tipos de cálculos numéricos em que resultados parciais são obtidos simultaneamente. Ele também pode ser usado na execução de uma thread que retorne um código de status indicando a conclusão bem-sucedida da thread.

Callable é uma interface genérica definida assim:

```
interface Callable<V>
```

Aqui, **V** indica o tipo de dado retornado pela tarefa. **Callable** define só um método, `call()`, mostrado abaixo:

```
V call() throws Exception
```

Dentro de **call()**, você deve definir a tarefa que deseja executar. Após ela ser concluída, o resultado será retornado. Se não puder ser calculado, **call()** deve lançar uma exceção.

Uma tarefa **Callable** é executada por um **ExecutorService**, com uma chamada a seu método **submit()**. Há três formas de **submit()**, mas só uma é usada na execução de um **Callable**. Ela é mostrada abaixo:

```
<T> Future<T> submit(Callable<T> tarefa)
```

Aqui, *tarefa* é o objeto **Callable** que será executado. O resultado é retornado em um objeto de tipo **Future**.

Future é uma interface genérica que representa o valor que será retornado por um objeto **Callable**. Já que esse valor é obtido em algum momento futuro, o nome **Future** é apropriado. **Future** é definida assim:

```
interface Future<V>
```

V especifica o tipo do resultado.

Para obter o valor retornado, você chamará o método **get()** de **Future**, que tem estas duas formas:

```
V get()
throws InterruptedException, ExecutionException
V get(long espera, TimeUnit tu)
throws InterruptedException, ExecutionException, TimeoutException
```

A primeira forma espera o resultado infinitamente. A segunda permite a especificação de um tempo limite em *espera*. As unidades de *espera* são passadas em *tu*, que é um objeto da enumeração **TimeUnit**, ainda a ser descrita neste capítulo.

O programa a seguir ilustra **Callable** e **Future** criando três tarefas que executam três cálculos diferentes. A primeira retorna a soma de um valor, a segunda calcula o tamanho da hipotenusa de um triângulo retângulo dado o tamanho de seus lados e a terceira calcula o fatorial de um valor. Todos os três cálculos podem ocorrer simultaneamente.

```
// Um exemplo que usa um Callable.

import java.util.concurrent.*;

class CallableDemo {
    public static void main(String[] args) {
        ExecutorService es = Executors.newFixedThreadPool(3);
        Future<Integer> f;
        Future<Double> f2;
        Future<Integer> f3;

        System.out.println("Starting");

        f = es.submit(new Sum(10)); —————
        f2 = es.submit(new Hypot(3, 4)); ————— Envia tarefas para execução.
        f3 = es.submit(new Factorial(5)); —————

        try {

```

```
System.out.println(f.get());
System.out.println(f2.get());
System.out.println(f3.get());
```

— Obtém e exibe os resultados.

```
} catch (InterruptedException exc) {
    System.out.println(exc);
}
catch (ExecutionException exc) {
    System.out.println(exc);
}

es.shutdown();
System.out.println("Done");
}
```

// A seguir, temos três threads de cálculo.

```
class Sum implements Callable<Integer> {
    int stop;

    Sum(int v) { stop = v; }

    public Integer call() {
        int sum = 0;
        for(int i = 1; i <= stop; i++) {
            sum += i;
        }
        return sum;
    }
}

class Hypot implements Callable<Double> {
    double side1, side2;

    Hypot(double s1, double s2) {
        side1 = s1;
        side2 = s2;
    }

    public Double call() {
        return Math.sqrt((side1*side1) + (side2*side2));
    }
}

class Factorial implements Callable<Integer> {
    int stop;

    Factorial(int v) { stop = v; }

    public Integer call() {
        int fact = 1;
```

```
    for(int i = 2; i <= stop; i++) {
        fact *= i;
    }
    return fact;
}
```

A saída é esta:

```
Starting
55
5.0
120
Done
```

Verificação do progresso

1. Para criar um **ExecutorService** que gerencie um pool de threads de tamanho fixo, devemos chamar o método factory _____.
2. Se você criar um **Executor** e chamar seu método **execute()** quatro vezes seguidas, passando a cada vez um **Runnable** diferente, cada **Runnable** só será executado quando o **Runnable** anterior tiver terminado. Verdadeiro ou falso?
3. Para executar um objeto **Callable**, você deve chamar o método _____ de um **ExecutorService**.
4. Por que “Future” é um nome apropriado para a interface **Future**?

A ENUMERAÇÃO TimeUnit

A API de concorrência define vários métodos que usam um argumento de tipo **TimeUnit**, que indica um período de tempo limite. **TimeUnit** é uma enumeração usada para especificar a *granularidade* (ou divisão) do tempo. Ela foi definida em **java.util.concurrent** e pode ter um dos valores a seguir:

DAYS
HOURS
MINUTES
SECONDS
MILLISECONDS
MICROSECONDS
NANOSECONDS

Respostas:

1. **Executors.newFixedThreadPool()**
2. Falso, a menos que o **Executor** gerencie um pool de threads fixo de tamanho 1.
3. **submit()**
4. Porque representa um valor que estará disponível em algum momento no futuro.

Embora **TimeUnit** permita a especificação de qualquer um desses valores em chamadas a métodos que usem um argumento de tempo, não há garantias de que o sistema consiga lidar com a granularidade especificada.

Vejamos um exemplo que usa **TimeUnit**. A classe **CallableDemo** da seção anterior foi modificada para usar a segunda forma de **get()** que usa um argumento **TimeUnit**:

```
try {
    System.out.println(f.get(10, TimeUnit.MILLISECONDS));
    System.out.println(f2.get(10, TimeUnit.MILLISECONDS));
    System.out.println(f3.get(10, TimeUnit.MILLISECONDS));
} catch (InterruptedException exc) {
    System.out.println(exc);
}
catch (ExecutionException exc) {
    System.out.println(exc);
} catch (TimeoutException exc) {
    System.out.println(exc);
}
```

Nessa versão, nenhuma chamada a **get()** esperará mais do que 10 milissegundos.

A enumeração **TimeUnit** define vários métodos que fazem a conversão entre unidades. Eles são:

```
long convert(long tval, TimeUnit tu)
long toMicros(long tval)
long toMillis(long tval)
long toNanos(long tval)
long toSeconds(long tval)
long toDays(long tval)
long toHours(long tval)
long toMinutes(long tval)
```

O método **convert()** converte *tval* para a unidade especificada e retorna o resultado. Os métodos **to** executam a conversão indicada e retornam o resultado.

Outros três métodos definidos por **TimeUnit** são:

```
void sleep(long atraso) throws InterruptedException
void timedJoin(Thread thrd, long atraso) throws InterruptedException
void timedWait(Object obj, long atraso) throws InterruptedException
```

Aqui, o método **sleep()** pausa a execução pelo período de retardo especificado, que é estabelecido segundo a constante da enumeração chamadora. Ele é convertido em uma chamada a **Thread.sleep()**. O método **timedJoin()** é uma versão especializada de **Thread.join()** em que *thrd* pausa pelo período de tempo especificado por *atraso*, que é descrito conforme a unidade de tempo chamadora. O método **timedWait()** é uma versão especializada de **Object.wait()** em que *obj* é esperado pelo período de tempo especificado por *atraso*, que é descrito conforme a unidade de tempo chamadora.

AS COLEÇÕES DE CONCORRÊNCIA

Como explicado, a API de concorrência define várias classes de coleção que foram criadas para operação concorrente. Elas são:

ArrayBlockingQueue	ConcurrentHashMap	ConcurrentLinkedDeque
ConcurrentLinkedQueue	ConcurrentSkipListMap	ConcurrentSkipListSet
CopyOnWriteArrayList	CopyOnWriteArraySet	DelayQueue
LinkedBlockingDeque	LinkedBlockingQueue	LinkedTransferQueue
PriorityBlockingQueue	SynchronousQueue	

Essas classes oferecem alternativas concorrentes às classes relacionadas definidas pelo Collections Framework. São coleções que funcionam de maneira semelhante às outras exceto por darem suporte à concorrência.

BLOQUEIOS

O pacote **java.util.concurrent.locks** dá suporte a *bloqueios*, que são objetos que oferecem uma alternativa ao uso de **synchronized** no controle de acesso a um recurso compartilhado. Em geral, é assim que um bloqueio funciona: antes do acesso a um recurso compartilhado, o bloqueio que protege esse recurso é adquirido. Quando o acesso ao recurso termina, o bloqueio é liberado. Se uma segunda thread tentar adquirir o bloqueio quando ele estiver sendo usado por outra thread, a segunda thread ficará suspensa até o bloqueio ser liberado. Dessa forma, o acesso conflituoso a um recurso compartilhado é evitado.

Os bloqueios são particularmente úteis quando várias threads precisam acessar o valor de dados compartilhados. Por exemplo, um aplicativo de controle de estoque poderia ter uma thread que primeiro confirmasse se um item existe no estoque e então diminuísse o número de itens disponíveis a cada venda. Se duas ou mais dessas threads estivessem sendo executadas, se não houver algum tipo de sincronização, seria possível uma thread estar no meio de uma transação e a segunda thread começar a sua. Como resultado, as duas threads poderiam assumir que existe um estoque adequado, mesmo se só houver estoque suficiente para uma venda. Nesse tipo de situação, um bloqueio oferece um meio conveniente de tratar a sincronização necessária.

Todos os bloqueios implementam a interface **Lock**. Os métodos definidos por **Lock** são mostrados na Tabela 27-1. Em geral, para se adquirir um bloqueio, é preciso chamar **lock()**. Se o bloqueio estiver indisponível, **lock()** esperará. Para liberar um bloqueio, devemos chamar **unlock()**. Para ver se um bloqueio está disponível, e adquiri-lo se estiver, chamamos **tryLock()**. Esse método não esperará o bloqueio se ele estiver indisponível. Em vez disso, ele retorna **true** se o bloqueio for adquirido; caso contrário, retorna **false**. O método **newCondition()** retorna um objeto **Condition** associado ao bloqueio. Usando **Condition()**, teremos um controle minucioso sobre o bloqueio por intermédio de métodos como **await()** e **signal()**, que fornecem funcionalidade semelhante à de **Object.wait()** e **Object.notify()**.

O pacote **java.util.concurrent.locks** fornece uma implementação de **Lock** chamada **ReentrantLock**. **ReentrantLock** implementa um *bloqueio reentrante*, que é um bloqueio em que a thread que atualmente o contém pode entrar repetidamente. No caso

Tabela 27-1 Os métodos de Lock

Método	Descrição
void lock()	Espera até o bloqueio chamador poder ser adquirido.
void lockInterruptibly() throws InterruptedException	Espera até o bloqueio chamador poder ser adquirido, a menos que seja interrompido.
Condition newCondition()	Retorna um objeto Condition associado ao bloqueio chamador.
boolean tryLock()	Tenta adquirir o bloqueio. Esse método não esperará se o bloqueio estiver indisponível. Em vez disso, ele retorna true se o bloqueio tiver sido adquirido e false se estiver sendo usado por outra thread.
boolean tryLock(long <i>espera</i> , TimeUnit <i>tu</i>) throws InterruptedException	Tenta adquirir o bloqueio. Se o bloqueio estiver indisponível, esse método não esperará mais do que o período especificado por <i>espera</i> , cujas unidades são estabelecidas por <i>tu</i> . Retorna true se o bloqueio for adquirido e false se ele não puder ser adquirido dentro do período especificado.
void unlock()	Libera o bloqueio, permitindo que seja usado por outra thread.

de uma thread reentrar em um bloqueio, todas as chamadas a **lock()** devem ter um número igual de chamadas a **unlock()**. Caso contrário, o bloqueio não será liberado.

O programa a seguir demonstra o uso de um bloqueio. Ele cria duas threads que acessam um recurso compartilhado chamado **Shared.count**. Antes de uma thread poder acessar **Shared.count**, deve obter um bloqueio. Após a obtenção do bloqueio, **Shared.count** é incrementado e, antes da liberação do bloqueio, a thread entra em suspensão. Isso faz a segunda thread tentar obter o bloqueio. No entanto, já que o bloqueio ainda é mantido pela primeira thread, a segunda thread deve esperar até que a primeira thread saia da suspensão e o libere. A saída mostra que o acesso a **Shared.count** é realmente sincronizado pelo bloqueio.

```
// Um exemplo de bloqueio simples.

import java.util.concurrent.locks.*;

class LockDemo {

    public static void main(String[] args) {
        ReentrantLock lock = new ReentrantLock(); ← Cria um bloqueio reentrante.

        new LockThread(lock, "A");
        new LockThread(lock, "B");
    }
}

// Um recurso compartilhado.
class Shared {
    static int count = 0;
}
```

```

// Uma thread de execução que incrementa count.
class LockThread implements Runnable {
    String name;
    ReentrantLock lock;

    LockThread(ReentrantLock lk, String n) {
        lock = lk;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Starting " + name);

        try {
            // Primeiro, bloqueia count.
            System.out.println(name + " is waiting to lock count.");
            lock.lock(); ← Adquire o bloqueio.
            System.out.println(name + " is locking count.");

            Shared.count++; ← Acessa o recurso.
            System.out.println(name + ": " + Shared.count);

            // Agora, permite uma mudança de contexto -- se possível.
            System.out.println(name + " is sleeping.");
            Thread.sleep(1000);
        } catch (InterruptedException exc) {
            System.out.println(exc);
        } finally {
            // Desbloqueia
            System.out.println(name + " is unlocking count.");
            lock.unlock(); ← Libera o bloqueio.
        }
    }
}

```

A saída é mostrada aqui. (A ordem exata em que as threads são executadas pode variar.)

```

Starting A
A is waiting to lock count.
A is locking count.
A: 1
A is sleeping.
Starting B
B is waiting to lock count.
A is unlocking count.
B is locking count.
B: 2
B is sleeping.
B is unlocking count.

```

O pacote **java.util.concurrent.locks** também define a interface **ReadWriteLock**. Essa interface especifica um bloqueio que mantém bloqueios separados para acesso de leitura e gravação. Isso permite que vários bloqueios sejam concedidos para a leitura de um recurso, contanto que ele não esteja sendo gravado. **ReentrantReadWriteLock** fornece uma implementação de **ReadWriteLock**.

Pergunte ao especialista

P Quais são as principais diferenças entre um **Semaphore**, um **Lock** e o bloqueio de monitor implícito usado em blocos **synchronized** como discutido no Capítulo 12?

R Os três objetos fornecem uma maneira de limitar o número de threads que podem acessar um recurso compartilhado em um determinado momento, mas eles têm propriedades diferentes. Os bloqueios de monitor são adquiridos e liberados implicitamente no começo e no fim de um bloco **synchronized**. Os **Locks** têm mais flexibilidade. Por exemplo, além do método **lock()**, os **Locks** têm uma versão sem bloqueio chamada **tryLock()** e uma versão **tryLock(espera, tu)** com bloqueio temporário. Um **ReentrantLock** (uma classe que implementa a interface **Lock**) é um bloqueio que passa a ser “propriedade” de uma thread quando ela chama **lock()** e deve ser liberado pela mesma thread com uma chamada a **unlock()**. Portanto, é recomendado que a chamada a **unlock()** esteja em um bloco **finally** para assegurarmos que o bloqueio seja liberado mesmo no caso de lançamento de uma exceção. Uma segunda chamada a **lock()** pela mesma thread que possui o bloqueio não será bloqueada, mas é preciso que haja uma chamada a **unlock()** para cada chamada a **lock()**. Um **ReentrantLock** também pode ser consultado em relação a seu proprietário atual e à fila de threads que estão esperando para adquirir o bloqueio.

Um **Semaphore** gerencia um conjunto de “permissões” que ele emite para as threads solicitantes. Se houver mais de uma permissão, várias threads poderão receber-las e acessar o recurso compartilhado ao mesmo tempo. Esse comportamento é útil, por exemplo, quando o recurso compartilhado é um pool de objetos que duas ou mais threads querem usar. Se o pool tiver N objetos, um semáforo com N permissões seria uma maneira de gerenciar o acesso ao pool. Basicamente, o semáforo controla apenas quantas permissões estão disponíveis e não quem recebeu uma permissão. Logo, é possível uma thread adquirir uma permissão, o que diminui quantas ainda estão disponíveis, e uma segunda thread liberar uma permissão, o que aumenta o número de permissões disponíveis, mesmo se ela não tiver adquirido uma. Se uma thread tiver adquirido uma permissão e tentar adquirir outra, será bloqueada se não houver mais permissões disponíveis (esse comportamento é diferente do de um bloqueio reentrante, como mencionado acima, porque a thread que possui o bloqueio pode chamar **lock()** com a frequência que quiser sem ser bloqueada). Um semáforo que gerencia apenas uma permissão é chamado de semáforo **binário** e é mais parecido com um bloqueio.

OPERAÇÕES ATÔMICAS

O pacote **java.util.concurrent.atomic** oferece uma alternativa aos outros recursos de sincronização na leitura ou gravação do valor de alguns tipos de variáveis. Esse pacote oferece métodos de obtenção, definição ou comparação do valor de uma variável em uma operação que não pode ser interrompida (isto é, atômica). Ou seja, nenhum bloqueio ou outro mecanismo de sincronização é necessário. Operações atômicas são

executadas com o uso de classes, como **AtomicInteger** e **AtomicLong**, e métodos como **get()**, **set()**, **compareAndSet()**, **decrementAndGet()** e **getAndSet()**, que executam a ação indicada por seus nomes. Em geral, as operações atômicas oferecem uma alternativa conveniente (e possivelmente mais eficaz) a outros mecanismos de sincronização quando só uma variável está envolvida.

Verificação do progresso

1. Quais são os sete valores da enumeração **TimeUnit**?
2. Em que os **AtomicIntegers** diferem dos **Integers**?
3. Para ter acesso a um recurso compartilhado protegido por um **Lock**, a thread deve chamar o método _____ de **Lock**, e quando não precisar mais do recurso, deve chamar o método _____, também de **Lock**.

PROGRAMAÇÃO PARALELA COM O FRAMEWORK FORK/JOIN

Programação paralela é o nome normalmente dado às técnicas que se beneficiam de computadores que contêm dois ou mais processadores (ou núcleos). Nos últimos tempos, os computadores *multicore* se tornaram lugar comum. A vantagem que os ambientes multiprocessadores oferecem é a possibilidade de uma melhoria significativa no desempenho. Como resultado, cresceu a expectativa por um mecanismo que dê aos programadores de Java uma forma simples, porém eficaz, de fazer uso de vários processadores de maneira hábil e escalável. Para atender a essa necessidade, o JDK 7 adicionou várias classes e interfaces novas que dão suporte à programação paralela. Elas são chamadas de *Framework Fork/Join* e estão definidas no pacote **java.util.concurrent**.

O Framework Fork/Join melhora a programação *multithread* em dois aspectos importantes. Em primeiro lugar, simplifica a criação e uso de várias threads. Em segundo lugar, faz uso de vários processadores automaticamente. Em outras palavras, usando o Framework Fork/Join, permitimos que o aplicativo se expanda automaticamente para fazer uso dos vários processadores disponíveis. Esses dois recursos tornam o Framework Fork/Join a abordagem recomendada para o multithreading quando certos tipos de processamento paralelo são desejados.

Antes de continuarmos, é importante destacar a diferença entre o multithreading tradicional e a programação paralela. No passado, a maioria dos computadores tinha apenas uma CPU e o multithreading era usado principalmente para tirar vantagem do tempo ocioso, como quando um programa está esperando entradas do usuário. Com essa abordagem, uma thread pode ser executada enquanto outra está esperando. Em outras palavras, em um sistema de CPU única, o multithreading é

Respostas:

1. DAY, HOUR, MINUTE, SECOND, MILLISECOND, MICROSECOND, NANOSECOND
2. Um **AtomicInteger** é um objeto mutável que armazena um valor inteiro e tem várias operações que não podem ser interrompidas, o que permite que você leia, compare e modifique o valor sem se preocupar com a sincronização.
3. **lock()** e **unlock()**

usado para permitir que duas ou mais tarefas compartilhem a CPU. Normalmente essa forma de multithreading é suportada por um objeto de tipo **Thread** (como descrito no Capítulo 12). Esse tipo de multithreading sempre será útil, mas não é otimizado para situações em que duas ou mais CPUs estão disponíveis (como nos computadores *multicore*).

Quando várias CPUs estão presentes, um segundo tipo de multithreading que dê suporte à execução paralela é necessário. Com duas ou mais CPUs, podemos executar partes de um programa simultaneamente, com cada parte sendo executada em sua própria CPU. Essa abordagem pode ser usada para acelerar significativamente a execução de alguns tipos de operações, como classificação, transformação ou busca em um conjunto grande. Em muitos casos, esses tipos de operações podem ser divididos em partes menores (cada uma atuando sobre uma parcela do conjunto) e cada parte pode ser executada em sua própria CPU. Como era de se esperar, o ganho em eficiência pode ser bem grande. Resumindo: a programação paralela oferece uma maneira de melhorar drasticamente o desempenho dos programas.

AS PRINCIPAIS CLASSES DO FRAMEWORK FORK/JOIN

O Framework Fork/Join fica no pacote **java.util.concurrent**. Estas são suas quatro classes básicas:

ForkJoinTask<V>	Uma classe abstrata que define uma tarefa.
ForkJoinPool	Gerencia a execução de ForkJoinTasks .
RecursiveAction	Uma subclasse de ForkJoinTask<V> para tarefas que não retornam valores.
RecursiveTask<V>	Uma subclasse de ForkJoinTask<V> para tarefas que retornam valores.

Seu relacionamento ocorre assim: um **ForkJoinPool** gerencia a execução de **ForkJoinTasks**. **ForkJoinTask** é uma classe abstrata que é estendida por duas outras classes abstratas: **RecursiveAction** e **RecursiveTask**. Normalmente, o código estende essas classes para criar uma tarefa. Antes de examinar o processo em detalhes, uma visão geral dos aspectos-chaves de cada classe será útil.

ForkJoinTask<V>

ForkJoinTask<V> é uma classe abstrata que define uma tarefa que pode ser gerenciada por um **ForkJoinPool**. O parâmetro de tipo **V** especifica o tipo de resultado da tarefa. **ForkJoinTask** difere de **Thread** por representar a abstração leve de uma tarefa, em vez de uma thread de execução. Os **ForkJoinTasks** são executados por threads gerenciadas por um pool de threads de tipo **ForkJoinPool**. Esse mecanismo permite que um grande número de tarefas seja gerenciado por um pequeno número de threads reais. Logo, os **ForkJoinTasks** são muito eficientes se comparados às threads.

ForkJoinTask define muitos métodos. Os principais são **fork()** e **join()**, mostrados aqui:

```
final ForkJoinTask<V> fork()
final V join()
```

O método **fork()** envia a tarefa chamadora para execução assíncrona. Ou seja, a thread que chama **fork()** continua a ser executada. O método **fork()** retorna **this** após a tarefa ser agendada para execução e só pode ser executado de dentro da parte computacional de outro **ForkJoinTask** em execução dentro de um **ForkJoinPool**. (Você verá como fazer isso em breve.) O método **join()** espera até a tarefa em que foi chamado terminar. O resultado da tarefa é retornado. Logo, com o uso de **fork()** e **join()**, você pode iniciar uma ou mais tarefas e esperá-las terminar.

Outro método importante de **ForkJoinTask** é **invoke()**. Ele combina as operações fork e join em uma única chamada porque inicia uma tarefa e então a espera terminar. Podemos vê-lo abaixo:

```
final V invoke()
```

O resultado da tarefa chamadora é retornado.

Você pode chamar mais de uma tarefa ao mesmo tempo usando **invokeAll()**. Duas de suas formas são mostradas aqui:

```
static void invokeAll(ForkJoinTask<?> tarefaA, ForkJoinTask<?> tarefaB)  
static void invokeAll(ForkJoinTask<?> ... listaTarefas)
```

No primeiro caso, *tarefaA* e *tarefaB* são executadas. No segundo, todas as tarefas especificadas são executadas. Nos dois casos, a thread chamadora espera até todas as tarefas especificadas terminarem. O método **invokeAll()** só pode ser executado de dentro da parte computacional de outro **ForkJoinTask** em execução dentro de um **ForkJoinPool**.

RecursiveAction

Uma das subclasses de **ForkJoinTask** é **RecursiveAction**. Essa classe encapsula uma tarefa que não retorna um resultado. Normalmente, o código estende **RecursiveAction** para criar uma tarefa que tenha um tipo de retorno **void**. **RecursiveAction** especifica quatro métodos, mas só um nos interessa aqui: o método abstrato chamado **compute()**. Quando estendemos **RecursiveAction** para criar uma classe concreta, inserimos o código que define a tarefa dentro de **compute()**. Ele representa a parte *computacional* da tarefa.

O método **compute()** é definido por **RecursiveAction** desta forma:

```
protected abstract void compute()
```

Observe que **compute()** é abstrato. Ou seja, deve ser implementado por uma subclasse (a menos que essa subclasse também seja abstrata). Não há implementação padrão.

Em geral, **RecursiveAction** é usada na implementação de uma estratégia recursiva de dividir e conquistar para tarefas que não retornam resultados. (Consulte “A estratégia de dividir e conquistar” mais à frente neste capítulo.)

RecursiveTask<V>

Outra subclasse de **ForkJoinTask** é **RecursiveTask<V>**. Essa classe encapsula uma tarefa que retorna um resultado. O tipo do resultado é especificado por **V**. Normalmente, o código estende **RecursiveTask<V>** para criar uma tarefa que retorne um valor. Como **RecursiveAction**, ela também especifica quatro métodos, mas só usaremos o método abstrato **compute()**, que representa a parte computacional da tarefa.

Quando estendemos **RecursiveTask**<V> para criar uma classe concreta, inserimos o código que representa a tarefa dentro de **compute()**. Esse código também deve retornar o resultado da tarefa.

O método **compute()** é definido por **RecursiveTask**<V> como descrito a seguir:

```
protected abstract V compute()
```

Observe que **compute()** também é abstrato, ou seja, deve ser implementado por uma subclasse. Quando implementado, deve retornar o resultado da tarefa.

Em geral, **RecursiveTask** é usada na implementação de uma estratégia recursiva de dividir e conquistar para tarefas que retornam resultados. (Consulte “A estratégia de dividir e conquistar” mais à frente neste capítulo.)

ForkJoinPool

A execução de **ForkJoinTasks** ocorre dentro de um **ForkJoinPool**, que também gerencia a execução das tarefas. Portanto, para executar um **ForkJoinTask**, primeiro você deve ter um **ForkJoinPool**.

ForkJoinPool define vários construtores. Estes são os dois mais usados:

```
ForkJoinPool()
ForkJoinPool(int nívelP)
```

O primeiro cria um pool padrão que dá suporte a um nível de paralelismo igual ao número de processadores disponíveis no sistema. O segundo permite a especificação do nível de paralelismo. Seu valor deve ser maior do que zero e não ultrapassar os limites da implementação. O nível de paralelismo determina o número de threads que podem ser executadas concorrentemente. Como resultado, ele determina o número de tarefas que podem ser executadas simultaneamente. (É claro que o número de tarefas executadas simultaneamente não pode exceder o número de processadores.) No entanto, é importante entender que o nível de paralelismo *não* limita o número de tarefas que podem ser gerenciadas pelo pool. Um **ForkJoinPool** pode gerenciar um número de tarefas muito maior do que seu nível de paralelismo. Além disso, o nível de paralelismo é apenas uma meta e não uma garantia.

Após criar uma instância de **ForkJoinPool**, você pode iniciar uma tarefa de várias maneiras. A primeira tarefa iniciada normalmente é considerada a tarefa principal. Com frequência, a tarefa principal inicia subtarefas que também são gerenciadas pelo pool. Uma maneira comum de iniciar uma tarefa principal é chamando o método **invoke()** no **ForkJoinPool**. Ele é mostrado aqui:

```
<T> T invoke(ForkJoinTask<T> tarefa)
```

Esse método inicia a tarefa especificada por *tarefa* e retorna seu resultado. Ou seja, o código chamador espera até **invoke()** retornar.

Para iniciar uma tarefa sem esperar sua conclusão, você pode usar **execute()**. Esta é uma de suas formas:

```
void execute(ForkJoinTask<?> tarefa)
```

Nesse caso, *tarefa* é iniciada, mas o código chamador não espera sua conclusão. Em vez disso, continua a ser executado sem sincronia.

ForkJoinPool gerencia a execução de suas threads usando uma abordagem chamada *roubo de tarefa*. Cada thread mantém uma fila de tarefas. Se a fila de uma thread estiver vazia, ela pegará uma tarefa de outra thread. Isso aumenta a eficiência geral e ajuda a manter a carga balanceada.

Mais uma coisa: **ForkJoinPool** usa threads daemon. Como explicado no Capítulo 12, uma thread daemon é encerrada automaticamente quando todas as threads de usuário terminam. Logo, não há necessidade de encerrar explicitamente um **ForkJoinPool**. No entanto, é possível fazê-lo chamando **shutdown()**.

A ESTRATÉGIA DE DIVIDIR E CONQUISTAR

Muitas aplicações do Framework Fork/Join empregam uma estratégia de *dividir e conquistar** baseada na recursão. É por isso que as duas subclasses de **ForkJoinTask** se chamam **RecursiveAction** e **RecursiveTask**. Fica subentendido que você entenderá uma dessas classes quando criar sua própria tarefa fork/join.

A estratégia de dividir e conquistar é baseada na divisão recursiva de uma tarefa em subtarefas menores até a subtarefa ser suficientemente pequena para ser tratada sequencialmente. Por exemplo, uma tarefa que aplique uma transformação a cada elemento de um array de N inteiros pode ser dividida em duas subtarefas em que cada uma transforme metade dos elementos do array. Isto é, uma subtarefa transforma os elementos de 0 a $N/2$ e a outra transforma os elementos $N/2+1$ a $N-1$. Sucessivamente, cada subtarefa pode ser reduzida em outro conjunto de subtarefas, cada uma transformando metade dos elementos restantes. Esse processo de dividir o array continuará até ser alcançado um limite em que uma solução sequencial seja mais veloz do que a criação de outra divisão.

A vantagem da estratégia de dividir e conquistar é que o processamento pode ocorrer em paralelo. Portanto, em vez de percorrermos um array inteiro usando uma única thread, partes do conjunto podem ser processadas simultaneamente. É claro que a abordagem de dividir e conquistar funciona em muitos casos em que um array (ou coleção) não está presente, mas muitas aplicações envolverão algum tipo de array, coleção ou agrupamento de dados.

Um dos segredos para o emprego mais adequado da estratégia de dividir e conquistar é selecionar corretamente o limite em que o processamento sequencial (em vez da divisão adicional) será usado. Normalmente, um limite ótimo é obtido com a modelagem das características de execução. Mas acelerações significativas continuarão ocorrendo até mesmo quando um limite menor do que o ótimo for usado. É melhor evitar limites muito altos ou muito baixos, no entanto. Quando este texto foi escrito, a documentação da API Java referente à classe **ForkJoinTask<T>** declarava que, como regra prática, uma tarefa sequencial deve ser executada em algum ponto entre 100 e 10.000 etapas computacionais.

Para aplicativos a serem executados em um sistema conhecido, com um número conhecido de processadores, você pode usar o número de processadores para tomar decisões embasadas sobre o valor do limite. No entanto, para aplicativos que serão executados em vários sistemas, cujos recursos não sejam conhecidos antecipadamente, não há como fazer suposições sobre o ambiente de execução.

* N. de R.T.: Do inglês, “divide-and-conquer strategy”.

Mais uma coisa: embora vários processadores possam estar disponíveis em um sistema, outros processos (e o próprio sistema operacional) estarão competindo com seu aplicativo por tempo de CPU. Logo, é importante não supor que seu programa terá acesso irrestrito a todas as CPUs. Além disso, diferentes execuções do mesmo programa podem exibir diferentes características de tempo de execução devido a cargas de tarefa variadas.

Um primeiro exemplo simples do Framework Fork/Join

A essa altura, será útil vermos um exemplo simples que demonstre o Framework Fork/Join e a estratégia de dividir e conquistar. Abaixo temos um programa que transforma os elementos de um array de **doubles** em suas raízes quadradas. Ele faz isso via uma subclasse de **RecursiveAction**.

```
// Um exemplo simples da estratégia básica de dividir e conquistar.
// Nesse caso, RecursiveAction é usada.
import java.util.concurrent.*;
import java.util.*;

// Um ForkJoinTask (via RecursiveAction) que transforma os elementos
// de um array de doubles em suas raízes quadradas. Uma tarefa baseada
// em RecursiveAction.
class SqrtTransform extends RecursiveAction { ←
    // O valor do limite é configurado arbitrariamente com 1.000 neste
    exemplo.
    // Em códigos do mundo real, seu valor ótimo pode ser
    // determinado pela modelagem e experimentação.
    final int seqThreshold = 1000;

    // Array a ser acessado.
    double[] data;

    // Determina que parte dos dados será processada.
    int start, end;

    SqrtTransform(double[] vals, int s, int e) {
        data = vals;
        start = s;
        end = e;
    }

    // Este é o método em que a computação paralela ocorrerá.
    protected void compute() { ←
        // Aqui que a computação ocorre.

        // Se o número de elementos estiver abaixo do limite sequencial,
        // processa sequencialmente.
        if((end - start) < seqThreshold) { ←
            // Transforma cada elemento em sua raiz quadrada.
            for(int i = start; i < end; i++) { ←
                data[i] = Math.sqrt(data[i]);
            }
        } ←
    else {
        // Caso contrário, continua a dividir os dados em partes menores.
    }
}
```

Processa sequencialmente.

```

    // Encontra o ponto intermediário.
    int middle = (start + end) / 2;

    // Chama novas tarefas, usando os dados subdivididos.
    invokeAll(new SqrtTransform(data, start, middle), ← Divide a tarefa
              new SqrtTransform(data, middle, end));      em duas.
}
}

// Demonstra a execução paralela.
class ForkJoinDemo {
    public static void main(String[] args) {
        // Cria um pool de tarefas.
        ForkJoinPool fjp = new ForkJoinPool();

        double[] nums = new double[100000];

        // Dá a nums alguns valores.
        for(int i = 0; i < nums.length; i++)
            nums[i] = (double) i;

        System.out.println("A portion of the original sequence:");

        for(int i=0; i < 10; i++)
            System.out.print(nums[i] + " ");
        System.out.println("\n");                                Cria uma tarefa.

        SqrtTransform task = new SqrtTransform(nums, 0, nums.length); ←

        // Inicia o ForkJoinTask principal.
        fjp.invoke(task); ← Executa a tarefa.

        System.out.println("A portion of the transformed sequence" +
                           " (to four decimal places):");
        for(int i=0; i < 10; i++)
            System.out.format("%.4f ", nums[i]);
        System.out.println();
    }
}

```

A saída do programa é mostrada aqui:

```

A portion of the original sequence:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

A portion of the transformed sequence (to four decimal places):
0.0000 1.0000 1.4142 1.7321 2.0000 2.2361 2.4495 2.6458 2.8284 3.0000

```

Como você pode ver, os valores dos elementos do array foram transformados em suas raízes quadradas.

Examinemos detalhadamente como esse programa funciona. Primeiro, observe que **SqrtTransform** é uma classe que estende **RecursiveAction**. Como explicado,

RecursiveAction estende **ForkJoinTask** para tarefas que não retornam resultados. Em seguida, observe a variável **final seqThreshold**. Esse é o valor que determina quando o processamento sequencial ocorrerá. Ele é configurado (um pouco arbitrariamente) com 1.000. Agora repare que uma referência ao array a ser processado é armazenada em **data** e que os campos **start** e **end** são usados para indicar os limites dos elementos a serem acessados.

A principal ação do programa ocorre no método **compute()**. Ele começa verificando se o número de elementos a serem processados é menor do que o limite do processamento sequencial. Se for, os elementos serão processados (com o cálculo de sua raiz quadrada nesse exemplo) e ocorrerá o retorno. Se o limite do processamento sequencial não tiver sido alcançado, duas novas tarefas serão iniciadas com uma chamada a **invokeAll()**. Nesse caso, cada subtarefa processará metade dos elementos. Como explicado antes, **invokeAll()** espera até as duas tarefas retornarem. Após todas as chamadas recursivas retornarem, cada elemento do array terá sido modificado, com grande parte da ação ocorrendo em paralelo (se vários processadores estiverem disponíveis).

Entendendo o impacto do nível de paralelismo

Antes de prosseguir, é importante entendermos o impacto que o nível de paralelismo tem sobre o desempenho de uma tarefa fork/join e como o paralelismo e o limite interagem. O programa mostrado nesta seção nos permite fazer testes com diferentes graus de paralelismo e valores de limite. Se usarmos um computador *multicore*, podemos observar interativamente o efeito desses valores.

No exemplo anterior, já que o construtor padrão de **ForkJoinPool** foi usado, o nível padrão de paralelismo é empregado, que é igual ao número de processadores do sistema. No entanto, você pode especificar o nível de paralelismo que quiser. Uma maneira já mostrada é especificá-lo ao criar um **ForkJoinPool** usando este construtor:

`ForkJoinPool(int nívelP)`

Aqui, *nívelP* especifica o nível de paralelismo, que deve ser maior do que zero e menor do que o limite definido pela implementação.

O próximo programa cria uma tarefa fork/join que transforma um array de **doubles**. A transformação é arbitrária, mas foi projetada para consumir vários ciclos de CPU. Isso foi feito para assegurar que os efeitos da alteração do limite ou do nível de paralelismo fossem exibidos mais claramente. Para usar o programa, especifique o valor do limite e o nível de paralelismo na linha de comando e o programa executará as tarefas. Ele também exibe o tempo que leva sua execução. Para fazê-lo, usa **System.nanoTime()**, que retorna o valor do timer de alta resolução da JVM.

```
// Um programa simples que nos permite fazer testes com os efeitos da
// alteração do limite e do paralelismo de um ForkJoinTask.
import java.util.concurrent.*;

// Um ForkJoinTask (via RecursiveAction) que executa uma
// transformação nos elementos de um array de doubles.
class Transform extends RecursiveAction {

    // Limite sequencial, que é definido pelo construtor.
    int seqThreshold;
```

```
// Array a ser acessado.
double[] data;

// Determina que parte dos dados será processada.
int start, end;

Transform(double[] vals, int s, int e, int t) {
    data = vals;
    start = s;
    end = e;
    seqThreshold = t;
}

// Este é o método em que a computação paralela ocorrerá.
protected void compute() {

    // Se o número de elementos estiver abaixo do limite sequencial,
    // processa sequencialmente.
    if((end - start) < seqThreshold) {
        // O código a seguir atribui a um elemento de um índice par a raiz
        // quadrada de seu valor original. Um elemento de um índice ímpar
        // recebe sua raiz cúbica. Esse código foi projetado para consumir
        // tempo de CPU suficiente apenas para os efeitos da execução
        // concorrente serem observados com mais rapidez.
        for(int i = start; i < end; i++) {
            if((data[i] % 2) == 0)
                data[i] = Math.sqrt(data[i]);
            else
                data[i] = Math.cbrt(data[i]);
        }
    } else {
        // Caso contrário, continua a dividir os dados em partes menores.

        // Encontra o ponto intermediário.
        int middle = (start + end) / 2;

        // Chama novas tarefas, usando os dados subdivididos.
        invokeAll(new Transform(data, start, middle, seqThreshold),
                  new Transform(data, middle, end, seqThreshold));
    }
}

// Demonstra a execução paralela.
class FJExperiment {

    public static void main(String[] args) {
        int pLevel;
        int threshold;

        if(args.length != 2) {
            System.out.println("Usage: FJExperiment parallelism threshold ");
        }
    }
}
```

```

        return;
    }

    pLevel = Integer.parseInt(args[0]);
    threshold = Integer.parseInt(args[1]);           ← Define o nível de paralelismo
                                                    e o limite sequencial como
                                                    especificado na linha de comando.

    // Essas variáveis são usadas para cronometrar a tarefa.
    long beginT, endT;

    // Cria um pool de tarefas. Observe que o nível de paralelismo é
    definido.
    ForkJoinPool fjp = new ForkJoinPool(pLevel);

    double[] nums = new double[1000000];

    for(int i = 0; i < nums.length; i++)
        nums[i] = (double) i;

    Transform task = new Transform(nums, 0, nums.length, threshold);

    // Começa a cronometrar.
    beginT = System.nanoTime();

    // Inicia o ForkJoinTask principal.
    fjp.invoke(task);

    // Termina a cronometragem.
    endT = System.nanoTime();

    System.out.println("Level of parallelism: " + pLevel);
    System.out.println("Sequential threshold: " + threshold);
    System.out.println("Elapsed time: " + (endT - beginT) + " ns");
    System.out.println();
}
}

```

Para usar o programa, especifique o nível de paralelismo, seguido do limite. Tente fazer testes com valores diferentes para cada um, observando os resultados. Lembre-se, para que funcione, você deve executar o código em um computador com pelo menos dois processadores. Também é bom ressaltar que duas execuções podem (e é quase certo que isso aconteça) produzir resultados diferentes devido a outros processos do sistema estarem consumindo tempo da CPU.

Para ter um ideia da diferença que o paralelismo provoca, faça este teste. Primeiro, execute o programa assim:

```
| java FJExperiment 1 1000
```

Essa linha solicita um nível de paralelismo (basicamente uma execução sequencial) com limite igual a 1.000. Veja um exemplo da execução produzido em um computador dual-core:

Level of parallelism: 1
Sequential threshold: 1000
Elapsed time: 259677487 ns

Agora, especifique dois níveis de paralelismo desta forma:

```
| java FJExperiment 2 1000
```

Aqui está um exemplo da saída dessa execução, produzida pelo mesmo computador dual-core:

```
| Level of parallelism: 2
| Sequential threshold: 1000
| Elapsed time: 169254472 ns
```

Como ficou claro, aumentar o paralelismo diminui significativamente o tempo de execução, aumentando assim a velocidade do programa. Você deve fazer testes variando o limite e o paralelismo em seu computador. Os resultados podem surpreendê-lo.

Há dois outros métodos que você pode achar úteis ao fazer testes com as características de execução de um programa fork/join. Em primeiro lugar, você pode obter o nível de paralelismo chamando o método **getParallelism()**, mostrado abaixo, que é definido por **ForkJoinPool**:

```
int getParallelism()
```

Esse método retorna o nível de paralelismo que está sendo usado atualmente. Lembre-se, por padrão, ele será igual ao número de processadores disponíveis. Em segundo lugar, pode obter o número de processadores disponibilizados para uso pela JVM chamando o método **availableProcessors()**, que é definido pela classe **Runtime**. Ele é mostrado aqui:

```
int availableProcessors()
```

Um exemplo que usa RecursiveTask<V>

Os dois exemplos anteriores se basearam em **RecursiveAction**, ou seja, executam concorrentemente tarefas que não retornam resultados. Para criar uma tarefa que retorna um resultado, usamos **RecursiveTask**. Em geral, as soluções são projetadas da mesma forma. A principal diferença é que o método **compute()** retorna um resultado. Logo, temos que agregar os resultados de modo que quando a primeira chamada terminar, ela retorne o resultado geral. Outra diferença é que normalmente iniciamos uma subtarefa chamando **fork()** e **join()** de maneira explícita (em vez de implicitamente chamando **invokeAll()**, por exemplo).

O programa a seguir demonstra **RecursiveTask**. Ele cria uma tarefa chamada **Sum** que retorna a soma dos valores de um array de **doubles**. Nesse exemplo, o array é composto por 5.000 elementos, que contêm os valores de 0 a 5.000. No entanto, valores positivos e negativos são intercalados. Logo, os primeiros valores do array são 0, -1, 2, -3, 4 e assim por diante.

```
// Um exemplo simples que usa RecursiveTask<V>.
import java.util.concurrent.*;

// Um RecursiveTask que calcula a soma de um array de doubles.
class Sum extends RecursiveTask<Double> { ←— Uma tarefa baseada em RecursiveTask.

    // O valor do limite sequencial.
    final int seqThresHold = 500;
```

```

// Array a ser acessado.
double[] data;

// Determina que parte dos dados será processada.
int start, end;

Sum(double[] vals, int s, int e) {
    data = vals;
    start = s;
    end = e;
}

// Encontra a soma de um array de doubles.
protected Double compute() {
    double sum = 0;

    // Se o número de elementos estiver abaixo do limite sequencial,
    // processa sequencialmente.
    if((end - start) < seqThresHold) {
        // Soma os elementos.
        for(int i = start; i < end; i++) sum += data[i];
    }
    else {
        // Caso contrário, continua a dividir os dados em partes menores.

        // Encontra o ponto intermediário.
        int middle = (start + end) / 2;

        // Chama novas tarefas, usando os dados subdivididos.
        Sum subTaskA = new Sum(data, start, middle);
        Sum subTaskB = new Sum(data, middle, end);

        // Inicia cada subtarefa com fork().
        subTaskA.fork(); ← Usa fork() para iniciar as tarefas.
        subTaskB.fork();

        // Espera as subtarefas retornarem e agraga os resultados.
        sum = subTaskA.join() + subTaskB.join(); ← Agrega os resultados parciais.
    }
    // Retorna a soma final.
    return sum;
}

// Demonstra a execução paralela.
class RecurTaskDemo {
    public static void main(String[] args) {
        // Cria um pool de tarefas.
        ForkJoinPool fjp = new ForkJoinPool();

```

```
double[] nums = new double[5000];

// Inicializa nums com valores que se alternam entre
// positivo e negativo.
for(int i=0; i < nums.length; i++)
    nums[i] = (double) (((i%2) == 0) ? i : -i);

Sum task = new Sum(nums, 0, nums.length);

// Inicia os ForkJoinTasks. Observe que, nesse caso,
// invoke() retorna um resultado.
double summation = fjp.invoke(task);

System.out.println("Summation " + summation);
}
```

Esta é a saída do programa:

```
| Summation -2500.0
```

Há alguns itens interessantes nesse programa. Primeiro, observe que as duas subtarefas são executadas com uma chamada a **fork()**, como mostrado aqui:

```
subTaskA.fork();
subTaskB.fork();
```

Nesse caso, o método **fork()** é usado porque ele inicia uma tarefa mas não a espera terminar. (Logo, executa a tarefa sem sincronia.) O resultado de cada tarefa é obtido com uma chamada a **join()**, como vemos abaixo:

```
| sum = subTaskA.join() + subTaskB.join();
```

Essa instrução espera cada tarefa terminar. Em seguida, soma os resultados de cada uma e atribui o total a **sum**. Logo, a soma de cada subtarefa é adicionada ao subtotal. Para concluir, **compute()** termina retornando **sum**, que será o total final quando a primeira chamada retornar.

Há outras maneiras de tratarmos a execução assíncrona das subtarefas. Por exemplo, a sequência a seguir usa **fork()** para iniciar **subTaskA()** e **invoke()** para iniciar e esperar **subTaskB()**:

```
subTaskA.fork();
sum = subTaskB.invoke() + subTaskA.join();
```

Outra alternativa seria **subTaskB()** chamar **compute()** diretamente, como mostrado aqui:

```
subTaskA.fork();
sum = subTaskB.compute() + subTaskA.join();
```

Executando uma tarefa de forma assíncrona

Os programas anteriores chamaram **invoke()** em um **ForkJoinPool** para iniciar uma tarefa. Normalmente essa abordagem é usada quando a thread chamadora deve espe-

rar até a tarefa ser concluída (o que costuma ser o caso) porque `invoke()` não retorna até a tarefa terminar. No entanto, você pode iniciar uma tarefa de forma assíncrona. Nessa abordagem, a thread chamadora continua sendo executada. Logo, tanto a thread chamadora quanto a tarefa são executados simultaneamente. Para iniciar uma tarefa de forma assíncrona, use `execute()`, que também é definido por `ForkJoinPool`. Ele tem as duas formas a seguir:

```
void execute(ForkJoinTask<?> tarefa)
void execute(Runnable tarefa)
```

Nas duas formas, *tarefa* especifica a tarefa a ser executada. Observe que a segunda forma permite a especificação de uma tarefa **Runnable** em vez de **ForkJoinTask**. Logo, forma uma ponte entre a abordagem Java tradicional do multithreading e o novo Framework Fork/Join. É importante entender que as threads usadas por um **ForkJoinPool** são daemons. Portanto, elas terminarão quando a thread principal terminar. Como resultado, você pode ter que manter a thread principal ativa até as tarefas terminarem.

OS UTILITÁRIOS DE CONCORRÊNCIA VERSUS A ABORDAGEM TRADICIONAL JAVA

Com o poder e a flexibilidade encontrados nos utilitários de concorrência, é natural que se faça a seguinte pergunta: eles substituem a abordagem tradicional que Java usa para o multithreading e a sincronização? A resposta é não! O suporte original ao multithreading e os recursos internos de sincronização ainda são o mecanismo que deve ser empregado para muitos programas Java. Por exemplo, `synchronized`, `wait()` e `notify()` oferecem soluções elegantes para uma ampla variedade de problemas. No entanto, quando um controle extra é necessário, os utilitários de concorrência estão disponíveis para desempenhar a tarefa. Além disso, o Framework Fork/Join oferece um meio poderoso de integração de técnicas de programação paralela aos aplicativos mais sofisticados.

Verificação do progresso

- Em que um **RecursiveAction** difere de um **RecursiveTask**?
- Em um **ForkJoinTask**, o método _____ agenda a tarefa para execução e o método _____ espera até a tarefa terminar e retorna o resultado.
- O método _____ da classe **ForkJoinTask** combina chamadas a `fork()` e `join()`.

Respostas:

- Um **RecursiveTask** retorna um resultado; um **RecursiveAction**, não.
- `fork()` e `join()`
- `invoke()`

EXERCÍCIOS

1. Como funciona um semáforo?
2. No programa **SemDemo** deste capítulo, o que aconteceria se você adicionasse uma chamada a **sem.release()** imediatamente antes das duas chamadas a **sem.acquire()**? Explique.
3. No programa **SemDemo** deste capítulo, o que aconteceria se você substituisse cada uma das duas chamadas a **sem.acquire()** por **sem.acquire(2)**? Explique.
4. Considere uma corrida de cavalos em que os cavalos e jóqueis tivessem que ficar alinhados no portão e esperar até todos estarem prontos antes de o portão abrir e os cavalos começarem a corrida. De que ferramenta Java de sincronização essa situação lhe faz lembrar?
5. Suponhamos que você adicionasse a instrução **System.out.println(name)**; como a última do método **run()** da classe **MyThread** no programa **Bar-Demo**. Nesse caso, antes de os nomes "A", "B" e "C" serem exibidos uma segunda vez, o string "Barrier reached!" seria exibido. Verdadeiro ou falso? Explique.
6. Com o uso de um **Exchanger**, é possível um **String** ser trocado por um **Integer**. Verdadeiro ou falso?
7. Se uma thread tentar obter o valor de um objeto **Future** antes de ele ter sido calculado, uma exceção será lançada. Verdadeiro ou falso?
8. Um objeto **Callable** é como um objeto **Runnable**, exceto por representar uma thread que _____.
9. Um **ForkJoinPool** usa a abordagem chamada “roubo de tarefa” para executar tarefas em espera. O que é o “roubo de tarefa”?
10. Uma estratégia muito apropriada para uso com o Framework Fork/Join é a de dividir e conquistar. Como ela funciona?
11. No programa **RecurTaskDemo**, o método **compute()** da classe **Sum** inclui as três linhas a seguir:

```
subTaskA.fork();
subTaskB.fork();
sum = subTaskA.join() + subTaskB.join();
```

Já que o método **invoke()** é uma combinação de fork e join, por que não chamar apenas **invoke()** para cada subtarefa? Isto é, por que não substituir essas três linhas pela linha a seguir?

```
| sum = subtaskA.invoke() + subtaskB.invoke();
```

12. Modifique o programa **CDLDemo** deste capítulo para que ele crie dois objetos **MyThread** em vez de apenas um. Em seguida, execute o programa e explique a saída obtida.
13. Suponhamos que você quisesse modificar o programa **SemDemo** deste capítulo para que ele criasse dois **Semaphores** com contagens de permissões igual a 1 e cada uma das duas threads tivesse que adquirir os dois semáforos antes de po-

der acessar o recurso compartilhado. O que pode dar errado se as duas threads tentarem adquirir os dois semáforos, porém na ordem oposta?

14. Abaixo temos um programa simples que cria e inicia duas threads da mesma classe. No método `run()` das threads, há duas instruções `print`, a primeira exibindo "before" e a segunda exibindo "after". Há duas saídas possíveis: "before after before after" e "before before after after". Modifique o programa usando um `CountDownLatch` para assegurar que a primeira saída possível, a saber, "before after before after", nunca ocorra.

```
class Exercise {
    public static void main(String[] args) {
        new MyThread();
        new MyThread();
    }
}

class MyThread implements Runnable {
    MyThread() {
        new Thread(this).start();
    }
    public void run() {
        System.out.print("before ");
        System.out.print("after ");
    }
}
```

15. Refaça o exercício anterior usando um `CyclicBarrier` em vez de um `CountDownLatch`.
16. No programa `BarDemo` deste capítulo, o que aconteceria se você substituísse o valor 3 pelo valor 4 como primeiro argumento do construtor de `CyclicBarrier`? Explique.
17. No programa `ExgrDemo`, o método `run()` da classe `UseString` contém a linha:
- ```
| str = ex.exchange(new String());
```
- O que aconteceria se você substituísse essa linha pela linha
- ```
| str = ex.exchange(str);
```
- que aparece no método `run()` da classe `MakeString`? Explique.
18. Um `Exchanger` é muito útil quando duas threads têm que trocar valores, mas se três threads tiverem que trocar valores? Mais precisamente, suponhamos que você tivesse três threads, A, B e C, e quisesse trocar valores entre elas para que o valor de A fosse passado para B, o valor de B fosse passado para C e o valor de C fosse passado para A. Explique como poderia fazê-lo.
19. Suponhamos que você tivesse três threads, A, B e C, com tarefas demoradas, como o download de arquivos grandes. Suponhamos que quisesse iniciar todas as threads ao mesmo tempo, mas que elas executassem suas tarefas sequencialmente na ordem A, B, C em vez de concorrentemente. Como poderia usar um `Phaser` para atingir seu objetivo?

20. Modifique o programa da seção Tente isto 27-1 para que, em vez de um retângulo, ele exiba um triângulo de asteriscos como este:

```
*****
 ***
 **
 *
```

Ele deve usar quatro threads como antes, cada uma sendo responsável pelo desenho de no máximo um asterisco por fase.

21. Modifique o programa **SimpExec** deste capítulo para que o método **run()** de **MyThread** entre em suspensão por 10 milissegundos entre cada chamada a **countDown()**. Em seguida, execute o programa quatro vezes, usando a cada vez um valor diferente (1, 2, 3 ou 4) como argumento do método **newFixedThreadPool()**. Explique os resultados.
22. No programa **CallableDemo** deste capítulo, adicione um quarto **Callable** cujo construtor use um array de inteiros como parâmetro e cujo método **call()** retorne o maior valor do array. Ele lança uma **Exception** se o array for nulo ou estiver vazio.
23. Procure na documentação os métodos **get()** e **set()** da classe **AtomicInteger**. Suponhamos que **ai** fosse um **AtomicInteger**. A natureza atômica das operações de **ai** garantem que, com a execução de **ai.set(ai.get() - ai.get())**, **ai** terá o valor 0?
24. Modifique o programa **RecurTaskDemo** para que, em vez de encontrar e exibir a soma dos elementos de um array de **doubles**, ele encontre e exiba o maior valor do array.

Apêndice A

Usando comentários de documentação da linguagem Java

Como explicado na Parte I, Java dá suporte a três tipos de comentários. Os dois primeiros são `//` e `/* */`. O terceiro tipo se chama *comentário de documentação*. Ele começa com a sequência de caracteres `/**` e termina com `*/`. Os comentários de documentação permitem que você embuta informações sobre o programa no próprio programa. Você então pode usar o programa utilitário **javadoc** (fornecido com JDK) para extrair as informações e inseri-las em um arquivo HTML. Os comentários de documentação tornam conveniente documentar programas. É provável que você já tenha visto documentação gerada com o **javadoc**, já que essa é a maneira como a biblioteca de APIs Java foi documentada.

TAGS DE javadoc

O utilitário **javadoc** reconhece as seguintes tags:

Tag	Significado
<code>@author</code>	Identifica o autor.
<code>{ @code }</code>	Exibe as informações como se encontram, em fonte de código, sem processar estilos HTML.
<code>@deprecated</code>	Especifica que um elemento do programa foi substituído.
<code>{ @docRoot }</code>	Especifica o caminho do diretório raiz da documentação atual.
<code>@exception</code>	Identifica uma exceção lançada por um método ou construtor.
<code>{ @inheritDoc }</code>	Herda um comentário da superclasse imediata.
<code>{ @link }</code>	Insere um link in-line conduzindo a outro tópico.
<code>{ @linkplain }</code>	Insere um link in-line conduzindo a outro tópico, mas o link é exibido em fonte de texto simples.
<code>{ @literal }</code>	Exibe as informações como se encontram, sem processar estilos HTML.
<code>@param</code>	Documenta um parâmetro.
<code>@return</code>	Documenta o valor de retorno de um método.

@see	Especifica um link que conduz a outro tópico.
@serial	Documenta um campo padrão que pode ser serializado.
@serialData	Documenta os dados gravados pelos métodos writeObject() ou writeExternal() .
@serialField	Documenta um componente ObjectStreamField .
@since	Informa a versão em que uma alteração específica foi introduzida.
@throws	Igual a @exception.
{@value}	Exibe o valor de uma constante, que deve ser um campo static .
@version	Especifica a versão de uma classe.

As tags de documentação que começam com um sinal de “arroba” (@) são chamadas de tags *autônomas* e devem ser usadas em sua própria linha. Tags que começam com uma chave, como {@code}, são chamadas de tags *in-line* e podem ser usadas dentro de uma descrição maior. Você também pode usar tags HTML padrão em um comentário de documentação. No entanto, algumas tags, como os cabeçalhos, não devem ser usadas, porque desorganizam a aparência do arquivo HTML produzido pelo **javadoc**.

Na documentação do código-fonte, você pode utilizar os comentários para descrever classes, interfaces, campos, construtores e métodos. Em todos os casos, o comentário deve vir imediatamente antes do item que está sendo documentado. Algumas tags, como **@see**, **@since** e **@deprecated**, podem ser usadas para documentar qualquer elemento. Outras só podem ser aplicadas a elementos relevantes. Examinaremos cada tag a seguir.

NOTA: Os comentários de documentação também podem ser usados para documentar um pacote e preparar uma visão geral, mas os procedimentos diferem dos usados na documentação do código-fonte. Consulte a documentação do **javadoc** para ver detalhes dessas aplicações. Este apêndice se refere apenas aos comentários de documentação de um programa Java.

@author

A tag **@author** documenta o autor de uma classe ou interface. Sua sintaxe é:

@author *descrição*

Aqui, geralmente *descrição* é o nome do autor. Você deverá especificar a opção **-author** quando executar **javadoc** para o campo **@author** ser incluído na documentação HTML.

{@code}

A tag {@code} permite que você embuta texto, como um fragmento de código, em um comentário. Esse texto é então exibido literalmente, em fonte de código, sem nenhum processamento adicional, como a geração em HTML. Sua sintaxe é:

{@code *fragmento-código*}

@deprecated

A tag **@deprecated** especifica que um elemento do programa não é mais usado. É recomendável que você inclua a tag **@see** ou **{@link}** para informar ao programador sobre as alternativas disponíveis. A sintaxe é a seguinte:

```
@deprecated descrição
```

Aqui, *descrição* é a mensagem que descreve a substituição. A tag **@deprecated** pode ser usada na documentação de campos, métodos, construtores, classes e interfaces.

{@docRoot}

{@docRoot} especifica o caminho do diretório raiz da documentação atual.

@exception

A tag **@exception** descreve uma exceção lançada por um método. Sua sintaxe é:

```
@exception nome-exceção explicação
```

Aqui, o nome da exceção é especificado por *nome-exceção*, e *explicação* é um string que descreve como a exceção pode ocorrer. A tag **@exception** só pode ser usada na documentação de um método ou construtor.

{@inheritDoc}

Essa tag herda um comentário da superclasse imediata.

{@link}

A tag **{@link}** fornece um link in-line que conduz a informações adicionais. Sua sintaxe é:

```
{@link pct.classe#membro texto}
```

Aqui, *pct.classe#membro* especifica o nome de uma classe ou método para o qual um link é adicionado, e *texto* é o string exibido.

{@linkplain}

Insere um link in-line que conduz a outro tópico. O link é exibido em fonte sem formatação. Caso contrário, seria igual a **{@link}**.

{@literal}

A tag **{@literal}** permite que você embuta texto em um comentário. Esse texto é então exibido como se encontra, sem nenhum processamento adicional, como a geração em HTML. Sua sintaxe é:

```
{@literal descrição}
```

Aqui, *descrição* é o texto que é embutido.

@param

A tag **@param** documenta um parâmetro. Sua sintaxe é:

@param *nome-parâmetro* *explicação*

Nesse caso, *nome-parâmetro* especifica o nome de um parâmetro. O significado desse parâmetro é descrito por *explicação*. A tag **@param** só pode ser usada na documentação de um método, de um construtor ou de uma classe ou interface genérica.

@return

A tag **@return** descreve o valor de retorno de um método. Sua sintaxe é:

@return *explicação*

Aqui, *explicação* descreve o tipo e o significado do valor retornado por um método. A tag **@return** só pode ser usada na documentação de um método.

@see

A tag **@see** fornece uma referência a informações adicionais. Suas duas formas mais usadas são estas:

@see *âncora*

@see *pct.classe#membro* *texto*

Na primeira forma, *âncora* é o link de um URL absoluto ou relativo. Na segunda forma, *pct.classe#membro* especifica o nome do item e *texto* é o texto exibido para esse item. O parâmetro de texto é opcional e, se não for usado, o item especificado por *pct.classe#membro* será exibido. O nome do membro também é opcional, logo, você pode especificar uma referência a um pacote, classe ou interface além da referência a um determinado método ou campo. O nome pode ser total ou parcialmente qualificado. No entanto, o ponto que precede o nome do membro (se ele existir) deve ser substituído por um caractere hash.

@serial

A tag **@serial** define o comentário de um campo padrão que pode ser serializado. Sua sintaxe é:

@serial *descrição*

Aqui, *descrição* é o comentário do campo.

@serialData

A tag **@serialData** documenta os dados gravados pelos métodos **writeObject()** e **writeExternal()**. Sua sintaxe é:

@serialData *descrição*

Nesse caso, *descrição* é o comentário dos dados.

@serialField

Para uma classe que implemente **Serializable**, a tag **@serialField** fornece comentários de um componente **ObjectStreamField**. Sua sintaxe é:

@serialField nome tipo descrição

Aqui, *nome* é o nome do campo, *tipo* é o tipo e *descrição* é o comentário do campo.

@since

A tag **@since** informa que um elemento foi introduzido em uma versão específica. Sua sintaxe é:

@since versão

Nessa tag, *versão* é o string que designa o lançamento ou a versão em que esse recurso foi disponibilizado.

@throws

A tag **@throws** tem o mesmo significado da tag **@exception**.

{@value}

{@value} tem duas formas. A primeira exibe o valor da constante que ela precede, que deve ser um campo **static**. A forma é dada a seguir:

{ @value }

A segunda forma exibe o valor de um campo **static** especificado. A forma é esta:

{ @value pct.classe#campo }

Aqui, *pct.classe#campo* especifica o nome do campo **static**.

@version

A tag **@version** especifica a versão de uma classe ou interface. Sua sintaxe é:

@version info

Aqui, *info* é um string contendo informações de versão, normalmente um número de versão, como 2.2. Você necessitará especificar a opção **-version** quando executar **javadoc** para o campo **@version** ser incluído na documentação HTML.

FORMA GERAL DE UM COMENTÁRIO DE DOCUMENTAÇÃO

Após o `/**` inicial, a primeira linha ou linhas são a descrição principal da classe, interface, campo, construtor ou método. Depois, você pode incluir uma ou mais das diversas tags **@**. Cada tag **@** deve começar no início de uma nova linha ou vir após um ou mais asteriscos (*) dispostos no início da linha. Se houver várias tags do mesmo tipo, elas devem ser agrupadas. Por exemplo, se você tiver três tags **@see**, insira-as uma

depois da outra. As tags in-line (as que começam com uma chave) podem ser usadas dentro de qualquer descrição.

Veja um exemplo de um comentário de documentação de uma classe:

```
/**  
 * Esta classe desenha um gráfico de barras.  
 * @author Herbert Schildt  
 * @version 3.2  
 */
```

O QUE javadoc GERA

O programa **javadoc** recebe como entrada o arquivo-fonte do programa Java e gera vários arquivos HTML contendo a documentação do programa. Informações sobre cada classe estarão em seu próprio arquivo HTML. O **javadoc** também gerará um índice e uma árvore hierárquica. Outros arquivos HTML podem ser gerados.

EXEMPLO QUE USA COMENTÁRIOS DE DOCUMENTAÇÃO

A seguir, temos um exemplo de programa que usa comentários de documentação. Observe a maneira como cada comentário vem imediatamente antes do item que descreve. Após ser processada por **javadoc**, a documentação sobre a classe **SquareNum** poderá ser encontrada em **SquareNum.html**.

```
import java.io.*;  
/**  
 * Esta classe demonstra comentários de documentação.  
 * @author Herbert Schildt  
 * @version 1.2  
 */  
public class SquareNum {  
    /**  
     * Este método retorna o quadrado de num.  
     * Esta é uma descrição de várias linhas.  
     * Você pode usar quantas linhas quiser.  
     * @param num O valor do qual queremos o quadrado.  
     * @return o quadrado de num como um double.  
    */  
    public double square(double num) {  
        return num * num;  
    }  
  
    /**  
     * Este método pega um número com o usuário.  
     * @return O valor da entrada como um double.  
     * @exception IOException Em caso de erro de entrada.  
     * @see IOException  
    */
```

```
public double getNumber() throws IOException {
    // cria um BufferedReader usando System.in
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader inData = new BufferedReader(isr);
    String str;

    str = inData.readLine();
    return (new Double(str)).doubleValue();
}

/**
 * Este método demonstra square().
 * @param args Não utilizado.
 * @exception IOException Em caso de erro.
 * @see IOException
 */
public static void main(String[] args)
    throws IOException
{
    SquareNum ob = new SquareNum();
    double val;

    System.out.println("Enter value to be squared: ");
    val = ob.getNumber();
    val = ob.square(val);

    System.out.println("Squared value is " + val);
}
```

Apêndice B

Introdução às expressões regulares

Este apêndice apresentará uma breve introdução a um recurso poderoso e às vezes complicado: a expressão regular. Como o termo é usado aqui, a *expressão regular* é um string de caracteres que descreve uma forma geral, chamada *padrão*. Esse padrão pode ser então usado na busca de ocorrências em outras sequências de caracteres. Com o uso de expressões regulares você pode executar operações sofisticadas de comparação de padrões. Por exemplo, as expressões regulares podem especificar um caractere curinga, conjuntos de caracteres e vários quantificadores. Esses e outros elementos permitem que uma expressão regular compare muitas sequências diferentes com grande flexibilidade.

Java dá amplo suporte às expressões regulares no pacote `java.util.regex`. Ela define duas classes que facilitam o processamento de expressões regulares: **Pattern** e **Matcher**. Essas classes funcionam em conjunto. **Pattern** define uma expressão regular. O padrão é comparado com outra sequência com o uso de **Matcher**.

A CLASSE Pattern

A classe **Pattern** não define construtores. Em vez disso, um padrão é criado com uma chamada a seu método **compile()**. Uma de suas formas é esta:

```
static Pattern compile(String padrão)
```

Aqui, *padrão* é a expressão regular que você deseja usar. O método **compile()** transforma o string de *padrão* em um padrão que possa ser usado na comparação de padrões pela classe **Matcher**. Ele retorna um objeto **Pattern** contendo o padrão. Se *padrão* não representar uma expressão válida, uma **PatternSyntaxException** será lançada.

Uma vez que você tiver criado um objeto **Pattern**, poderá usá-lo para criar um **Matcher**. Isso é feito com uma chamada ao método **matcher()** definido por **Pattern**. Ele é mostrado abaixo:

```
Matcher matcher(CharSequence str)
```

Aqui, *str* é a sequência de caracteres com a qual o padrão será comparado. Ela é chamada de *sequência de entrada*. Como mencionado no Capítulo 23, **CharSequence** é uma interface que define um conjunto de caracteres somente leitura. Ela é implementada pela classe **String**, entre outras. Logo, você pode passar um string para **Matcher**.

A CLASSE Matcher

A classe **Matcher** não tem construtores. Em vez disso, você pode criar um **Matcher** chamando o método **matcher()** definido por **Pattern**, como acabamos de explicar. Uma vez que você tiver criado um **Matcher**, poderá usar seus métodos para executar várias operações de comparação de padrões.

O método mais simples de comparação de padrões é **matches()**, que determina apenas se a sequência de caracteres coincide com o padrão. Podemos vê-lo abaixo:

```
boolean matches()
```

Ele retorna **true** se a sequência e o padrão coincidirem; caso contrário, retorna **false**. É bom ressaltar que a sequência inteira deve coincidir com o padrão e não apenas uma subsequência dela.

Para determinar se uma subsequência da sequência de entrada coincide com o padrão, use **find()**. Uma versão é mostrada aqui:

```
boolean find()
```

Ele retorna **true** se houver uma subsequência coincidente; caso contrário, retorna **false**. Esse método pode ser chamado repetidamente para encontrar todas as subsequências coincidentes. Cada chamada a **find()** começa onde a anterior terminou.

Você pode obter um string contendo a sequência coincidente atual chamando **group()**. Uma de suas formas é esta:

```
String group()
```

O string coincidente é retornado. Se não houver ocorrências, uma **IllegalStateException** será lançada.

Podemos obter o índice da ocorrência atual dentro da sequência de entrada chamando **start()**. O índice que fica uma unidade após o fim da ocorrência atual é obtido com uma chamada a **end()**. Abaixo temos uma forma de cada método:

```
int start()  
int end()
```

Os dois lançam uma **IllegalStateException** se não houver ocorrências.

Você pode substituir todas as ocorrências de uma sequência coincidente por outra sequência chamando o método **replaceAll()**, mostrado a seguir:

```
String replaceAll(String novaStr)
```

Aqui, *novaStr* especifica a nova sequência de caracteres que substituirá as que coincidem com o padrão. A sequência de entrada atualizada é retornada como um string.

ASPECTOS BÁSICOS DA SINTAXE DAS EXPRESSÕES REGULARES

Antes de demonstrarmos **Pattern** e **Matcher**, é necessário explicar como construir uma expressão regular. Embora não haja uma regra complicada, há muitas delas e uma discussão completa não faz parte do escopo deste livro. No entanto, algumas das estruturas básicas serão descritas aqui.

Em geral, uma expressão regular é composta por caracteres normais, classes de caracteres (conjuntos de caracteres), o caractere curinga e quantificadores. Um caractere normal é comparado na forma como se encontra. Logo, se um padrão for composto por "xy", a única sequência de entrada que coincidirá com ele é "xy". Caracteres como os de nova linha e tabulação são especificados com o uso das sequências de escape padrão, que começam com uma barra invertida. Por exemplo, uma nova linha é especificada por `\n`. Na linguagem das expressões regulares, um caractere normal também é chamado de *literal*.

Uma classe de caracteres é um conjunto de caracteres. Ela é especificada com a inserção dos caracteres da classe entre colchetes. Por exemplo, a classe `[wxyz]` apresentaria correspondência com w, x, y ou z. Para especificar um conjunto inverso, preceda os caracteres com um acento circunflexo. Por exemplo, `[^wxyz]` apresentaria correspondência com qualquer caractere exceto w, x, y ou z. Você pode especificar um intervalo de caracteres usando um hífen. Por exemplo, para especificar uma classe de caracteres que corresponda aos dígitos de 1 a 9, use `[1-9]`. Java também fornece várias classes predefinidas. Por exemplo, a classe `\d` corresponde aos dígitos 0 a 9, e a classe `\w` corresponde aos caracteres que podem fazer parte de uma palavra.

Outra classe predefinida é o ponto. Ele é, essencialmente, um *caractere curinga*, porque corresponde a qualquer caractere exceto um finalizador de linha. (É possível alterar o comportamento do ponto para que também capture finalizadores de linha, mas não faremos isso aqui.) Logo, um padrão composto por `".` corresponderá a essas (e outras) sequências de entrada: "A", "a", "X" e assim por diante.

Um quantificador determina quantas vezes uma expressão pode ocorrer. Estes são os quantificadores básicos:

<code>+</code>	Ocorre uma ou mais vezes.
<code>*</code>	Ocorre zero ou mais vezes.
<code>?</code>	Ocorre zero ou uma vez.

Por exemplo, o padrão `"x+"` apresentará correspondência com "x", "xx" e "xxx", entre outras. Como você verá em breve, esses quantificadores são chamados de *vorazes* devido ao seu comportamento.

DEMONSTRANDO A CORRESPONDÊNCIA DE PADRÕES

A melhor maneira de se entender como a comparação de padrões de expressões regulares opera é por meio de alguns exemplos. O primeiro, mostrado aqui, procura uma coincidência com um padrão literal.

```
// Uma demonstração simples da comparação de padrões.
import java.util.regex.*;

class RegExpr {
    public static void main(String[] args) {
        Pattern pat;
        Matcher mat;
        boolean found;

        pat = Pattern.compile("Alpha");
        mat = pat.matcher("Alpha");
        found = mat.find();
        System.out.println(found);
    }
}
```

```
mat = pat.matcher("Alpha");
found = mat.matches(); // procura uma ocorrência

System.out.println("Testing Alpha against Alpha.");
if(found) System.out.println("Matches");
else System.out.println("No Match");

System.out.println();

System.out.println("Testing Alpha against Alpha Beta Gamma.");
mat = pat.matcher("Alpha Beta Gamma"); // cria um novo comparador

found = mat.matches(); // procura uma ocorrência

if(found) System.out.println("Matches");
else System.out.println("No Match");
}
}
```

A saída do programa é esta:

```
Testing Alpha against Alpha.
Matches

Testing Alpha against Alpha Beta Gamma.
No Match
```

Examinemos esse programa em detalhes. Ele começa criando o padrão que contém a sequência "Alpha". Em seguida, um **Matcher** é criado para esse padrão contendo a sequência de entrada "Alpha". O método **matches()** é então chamado para determinar se a sequência de entrada coincide com o padrão. Já que a sequência e o padrão são iguais, **matches()** retorna **true**. Agora, um novo **Matcher** é criado com a sequência de entrada "Alpha Beta Gamma", e **matches()** é chamado novamente. Nesse caso, o padrão e a sequência de entrada diferem e nenhuma ocorrência é encontrada. Lembre-se, a função **matches()** só retorna **true** quando a sequência de entrada coincide exatamente com o padrão. Ela não retornará **true** só porque uma subsequência apresenta coincidência.

Você pode usar **find()** para determinar se a sequência de entrada contém uma subsequência que coincida com o padrão. Considere o programa a seguir:

```
// Usa find() para encontrar uma subsequência.
import java.util.regex.*;

class RegExpr2 {
    public static void main(String[] args) {
        Pattern pat = Pattern.compile("Alpha");
        Matcher mat = pat.matcher("Alpha Beta Gamma");

        System.out.println("Looking for Alpha in Alpha Beta Gamma.");

        if(mat.find()) System.out.println("subsequence found");
        else System.out.println("No Match");
    }
}
```

A saída é mostrada abaixo:

```
| Looking for Alpha in Alpha Beta Gamma.
| subsequence found
```

Nesse caso, **find()** encontra a subsequência "Alpha".

O método **find()** pode ser usado para encontrar ocorrências repetidas do padrão na sequência de entrada porque cada chamada a **find()** começa onde a anterior terminou. Por exemplo, o próximo programa encontra duas ocorrências do padrão "Beta":

```
// Usa find() para encontrar várias subsequências.
import java.util.regex.*;

class RegExpr3 {
    public static void main(String[] args) {
        Pattern pat = Pattern.compile("Beta");
        Matcher mat = pat.matcher("Alpha Beta Gamma Beta Theta");

        while(mat.find()) {
            System.out.println("Beta found at index " + mat.start());
        }
    }
}
```

A saída é esta:

```
| Beta found at index 6
| Beta found at index 17
```

Como a saída mostra, duas ocorrências foram encontradas. O programa usa o método **start()** para obter o índice de cada ocorrência.

USANDO O CARACTERE CURINGA E QUANTIFICADORES

Embora os programas anteriores mostrem a técnica geral do uso de **Pattern** e **Matcher**, eles não mostram seu poder. O benefício real do processamento de expressões regulares não é visto até o curinga e os quantificadores serem usados. Para começar, considere o exemplo a seguir, que usa o quantificador **+** para encontrar qualquer sequência longa e arbitrária de Ws. Lembre-se, **+** procura uma expressão uma ou mais vezes.

```
// Usa um quantificador.
import java.util.regex.*;

class RegExpr4 {
    public static void main(String[] args) {
        Pattern pat = Pattern.compile("W+");
        Matcher mat = pat.matcher("W WW WWW");

        while(mat.find())
            System.out.println("Match: " + mat.group());
    }
}
```

A saída do programa é:

```
|Match: W  
|Match: WW  
|Match: WWW
```

Como a saída mostra, o padrão "W+" da expressão regular coincide com qualquer sequência longa e arbitrária de Ws.

O próximo programa usa um curinga para criar um padrão que procure qualquer sequência de tamanho 3 ou mais que comece com *e* e termine com *d*. Para fazê-lo, ele usa o caractere ponto junto com o quantificador +.

```
// Usa o curinga e o quantificador.*.  
import java.util.regex.*;  
  
class RegExpr5 {  
    public static void main(String[] args) {  
        Pattern pat = Pattern.compile("e.+d");  
        Matcher mat = pat.matcher("extend cup end table");  
  
        while(mat.find())  
            System.out.println("Match: " + mat.group());  
    }  
}
```

Você pode ficar surpreso com a saída produzida pelo programa, que é mostrada aqui:

```
|Match: extend cup end
```

Só uma ocorrência foi encontrada e ela é a sequência mais longa que começa com *e* e termina com *d*. Você pode ter achado que seriam duas ocorrências: "extend" e "end". A razão para a sequência mais longa ter sido encontrada é que, por convenção, **find()** procura a sequência mais longa que se ajusta ao padrão. Isso se chama *comportamento voraz*. Todos os quantificadores básicos exibem o comportamento voraz; portanto, são chamados de *quantificadores vorazes*.

Você pode usar dois outros tipos de comportamento dos quantificadores. Eles são o *relutante* e o *possessivo*. O comportamento relutante é especificado com o acréscimo de um ? ao quantificador. Portanto, os quantificadores relutantes são +?, *? e ???. Os quantificadores relutantes fazem o padrão coincidente mais curto ser obtido. Por exemplo, se você substituir "**e.+?d**" como expressão regular no programa anterior, verá a saída a seguir:

```
|Match: extend  
|Match: end
```

Como a saída mostra, o padrão "**e.+?d**" procura a sequência mais curta de tamanho 3 ou mais que comece com *e* e termine com *d*. Logo, duas ocorrências são encontradas.

Um quantificador possessivo procurará a sequência mais longa que puder, mesmo se uma sequência mais curta permitisse que a comparação com a expressão regular inteira fosse bem-sucedida. Ele é criado com o acréscimo de um +. Logo, os quantificadores possessivos são ++, *+ e ?+. Por exemplo, se você substituir "**e.++d**" no programa anterior como padrão, ele não relatará ocorrências porque .++ consumirá todos os caracteres até o fim do string. Portanto, não há ocorrência do **d**.

Mais uma coisa: há formas dos quantificadores que nos permitem especificar quantas vezes queremos encontrar a ocorrência de um padrão. Isso é feito com a inserção de uma contagem após o quantificador, entre chaves. Outras variações permitem a especificação do número mínimo de vezes que uma ocorrência deve ser procurada, ou de um intervalo de vezes.

TRABALHANDO COM CLASSES DE CARACTERES

Você poderia querer encontrar qualquer sequência que tivesse um ou mais caracteres, em qualquer ordem, que fizessem parte de um conjunto de caracteres. Por exemplo, digamos que para buscar palavras inteiras, você quisesse comparar qualquer sequência das letras do alfabeto. Uma das maneiras mais fáceis de fazê-lo é usando uma classe de caracteres, que define um conjunto de caracteres. Lembre-se, uma classe de caracteres é criada com a inserção dos caracteres procurados entre colchetes. Por exemplo, para procurar os caracteres minúsculos de a a z, use **[a-z]**. O programa a seguir demonstra essa técnica:

```
// Usa uma classe de caracteres.
import java.util.regex.*;

class RegExpr6 {
    public static void main(String[] args) {
        // Procura palavras em minúsculas.
        Pattern pat = Pattern.compile("[a-z]+");
        Matcher mat = pat.matcher("this is a test.");

        while(mat.find())
            System.out.println("Match: " + mat.group());
    }
}
```

A saída é mostrada aqui:

```
Match: this
Match: is
Match: a
Match: test
```

USANDO replaceAll()

O método **replaceAll()** fornecido por **Matcher** permite a execução de poderosas operações de busca e substituição com o uso de expressões regulares. Por exemplo, o programa a seguir substitui por "Eric" todas as ocorrências de sequências que começem com "Jon".

```
// Usa replaceAll().
import java.util.regex.*;

class RegExpr7 {
    public static void main(String[] args) {
        String str = "Jon Jonathan Frank Ken Todd";
        System.out.println(str.replaceAll("Jon", "Eric"));
    }
}
```

```
Pattern pat = Pattern.compile("Jon.*? ");
Matcher mat = pat.matcher(str);

System.out.println("Original sequence: " + str);

str = mat.replaceAll("Eric ");

System.out.println("Modified sequence: " + str);
}
}
```

A saída é:

```
|Original sequence: Jon Jonathan Frank Ken Todd
|Modified sequence: Eric Eric Frank Ken Todd
```

Já que a expressão regular "Jon.*?" procura relutantemente qualquer string que comce com Jon, seguido por zero ou mais caracteres e terminando em um espaço, ela pode ser usada para procurar e substituir tanto Jon quanto Jonathan pelo nome Eric. Seria mais difícil fazer essa substituição sem os recursos de comparação de padrões.

A CONEXÃO COM A CLASSE String

Ao trabalhar com **Strings**, você pode executar uma comparação de padrões sem criar um **Matcher** ou um **Pattern** diretamente. Para fazê-lo, use o método **matches()** fornecido por **String**. Ele é mostrado abaixo:

```
boolean matches(String padrão)
```

Se o string chamador coincidir com a expressão regular de *padrão*, **matches()** retornará **true**. Caso contrário, retornará **false**. Se *padrão* não representar uma expressão válida, uma **PatternSyntaxException** será lançada. O método **matches()** de **String** é muito conveniente quando só uma comparação ocasional é necessária.

ASSUNTOS A EXPLORAR

Esta breve introdução às expressões regulares apresentou vários recursos-chave. No entanto, há muitos outros recursos fornecidos pelas expressões regulares. Eis algumas áreas que podem ser exploradas: o trabalho com grupos, o operador OR, o uso de comparadores limitados e o uso de vários flags. Tanto **Pattern** quanto **Matcher** também definem métodos adicionais além dos mencionados aqui que você pode achar úteis em alguns casos. Veja dois exemplos: **Pattern** define o método **split()**, que permite a redução de um string a suas partes individuais; e o método **replaceFirst()** de **Matcher** substitui a primeira ocorrência de uma expressão regular por um string especificado.

Apêndice C

Respostas de exercícios selecionados

CAPÍTULO 1

1. Cite as três partes essenciais de um computador.

A CPU, a memória e o dispositivo de entrada/saída.

2. O que é código-fonte? E código-objeto?

Código-fonte é a forma de um programa legível por humanos. Código-objeto é a forma executável do programa.

3. Como fica o valor 14 em binário? Qual é o equivalente decimal ao número binário 1010 0110?

14 em binário é 1110. O valor 1010 0110 em decimal é 166.

4. Como regra geral, um byte é composto por _____ bits.

8

5. O que é bytecode e por que ele é importante para o uso de Java em programação na Internet?

Bytecode é um conjunto de instruções altamente otimizado que é executado pela Máquina Virtual Java. Ele ajuda Java a fornecer portabilidade e segurança.

6. Quais são os três princípios básicos da programação orientada a objetos?

Encapsulamento, polimorfismo e herança.

7. Onde os programas Java começam a ser executados?

Os programas Java começam a ser executados em **main()**.

8. O que é uma variável?

Uma variável é um local nomeado na memória. O conteúdo de uma variável pode ser alterado durante a execução de um programa.

9. Qual dos nomes de variável a seguir é inválido?

A variável inválida é a da opção **D**. Nomes de variável não podem começar com um dígito.

10. Como se cria um comentário de linha única? E um comentário de várias linhas?

Um comentário de linha única começa com // e termina no fim da linha. Um comentário de várias linhas começa com /* e termina com */.

11. Mostre a forma geral da instrução **if**. Mostre também a do laço **for**.

Forma geral de **if**:

```
if(condição) instrução;
```

Forma geral de **for**:

```
for(inicialização; condição; iteração) instrução;
```

12. Como se cria um bloco de código?

Um bloco de código começa com uma chave de abertura e termina com uma chave de fechamento.

13. A gravidade da Lua é cerca de 17% da gravidade da Terra. Crie um programa que calcule seu peso efetivo na Lua.

```
/*
 * Calcula seu peso na Lua.

 * Chame este arquivo de Moon.java.
 */
class Moon {
    public static void main(String[] args) {
        double earthWeight; // peso na Terra
        double moonWeight; // peso na Lua

        earthWeight = 165;

        moonWeight = earthWeight * 0.17;

        System.out.println(earthWeight +
                           " earth-pounds is equivalent to " +
                           moonWeight + " moon-pounds.");
    }
}
```

14. Adapte o código da seção Tente isto 1-2 para que ele exiba uma tabela de conversões de polegadas para metros. Exiba 12 pés de conversões, polegada a polegada. Gere uma linha em branco a cada 12 polegadas. (Um metro é igual a aproximadamente 39,37 polegadas.)

```
/*
 * Este programa exibe uma tabela de conversão
 * de polegadas para metros.

 * Chame-o de InchToMeterTable.java.
 */
class InchToMeterTable {
    public static void main(String[] args) {
```

```

double inches, meters;
int counter;

counter = 0;
for(inches = 1; inches <= 144; inches++) {
    meters = inches / 39.37; // converte para metros
    System.out.println(inches + " inches is " +
                        meters + " meters.");

    counter++;
    // a cada 12 linhas, exibe uma linha em branco
    if(counter == 12) {
        System.out.println();
        counter = 0; // zera o contador de linhas
    }
}
}
}

```

- 15.** Se você cometer um engano de digitação ao inserir seu programa, isso vai resultar em que tipo de erro?

Erro de sintaxe.

- 16.** É importante o local onde inserimos uma instrução em uma linha?

Não, Java é uma linguagem de forma livre.

CAPÍTULO 2

- 1.** Por que Java especifica rigorosamente o intervalo e o comportamento de seus tipos primitivos?

Java especifica rigorosamente o intervalo e o comportamento de seus tipos primitivos para assegurar portabilidade entre as plataformas.

- 2.** Qual é o tipo de caractere usado em Java e em que ele é diferente do tipo de caractere usado por outras linguagens de programação?

O tipo de caractere de Java é o **char**. Os caracteres Java são Unicode em vez de ASCII, que é usado por outras linguagens de programação.

- 3.** Um valor **boolean** pode ter o valor que você quiser, já que qualquer valor diferente de zero é verdadeiro. Verdadeiro ou falso?

Falso. Um valor booleano deve ser **true** ou **false**.

- 4.** Dada esta saída,

One
Two
Three

usando um único string, mostre a instrução **println()** que a produziu.

```
| System.out.println("One\nTwo\nThree");
```

5. O que está errado no fragmento a seguir?

```
| for(i = 0; i < 10; i++) {  
|     int sum;  
  
|     sum = sum + i;  
| }  
System.out.println("Sum is: " + sum);
```

Há três erros básicos no fragmento. Em primeiro lugar, **sum** é criada sempre que o bloco gerado pelo laço **for** é alcançado e destruída na saída. Logo, ela não manterá seu valor entre as iterações. Não adianta tentar usar **sum** para armazenar um acumulado das iterações. Em segundo lugar, **sum** não será conhecida fora do bloco em que é declarada, portanto, a referência a ela na instrução **println()** é inválida. Para concluir, **sum** não é inicializada.

6. Explique a diferença entre as formas prefixada e pós-fixada do operador de incremento.

Quando o operador de incremento preceder seu operando, Java executará a operação correspondente antes de obter o valor do operando para que seja usado pelo resto da expressão. Se o operador vier após o operando, Java obterá seu valor antes de incrementar.

7. Mostre como um AND de curto-circuito pode ser usado para impedir um erro de divisão por zero.

```
| if((b != 0) && (val / b == 1)) ...
```

8. Em uma expressão, a que tipo são promovidos **byte** e **short**?

Em uma expressão, **byte** e **short** são promovidos a **int**.

9. Em geral, quando uma coerção é necessária?

Uma coerção é necessária quando uma conversão redutora está ocorrendo.

10. Escreva um programa que encontre todos os números primos entre 2 e 100.

```
// Encontra os números primos entre 2 e 100.  
class Prime {  
    public static void main(String[] args) {  
        int i, j;  
        boolean isprime;  
  
        for(i=2; i < 100; i++) {  
            isprime = true;  
  
            // vê se o número tem divisão exata  
            for(j=2; j <= i/j; j++)  
                // se tiver, não é primo  
                if((i%j) == 0) isprime = false;  
  
            if(isprime)  
                System.out.println(i + " is prime.");  
        }  
    }  
}
```

- 11.** O uso de parênteses adicionais afeta o desempenho do programa?

Não.

- 12.** Um bloco define um escopo?

Sim.

CAPÍTULO 3

- 1.** Escreva um programa que leia caracteres do teclado até um ponto ser recebido. Faça-o contar o número de espaços. Relate o total no fim do programa.

```
// Conta espaços.
class Spaces {
    public static void main(String[] args)
        throws java.io.IOException {

        char ch;
        int spaces = 0;

        System.out.println("Enter a period to stop.");

        do {
            ch = (char) System.in.read();
            if(ch == ' ') spaces++;
        } while(ch != '.');

        System.out.println("Spaces: " + spaces);
    }
}
```

- 2.** Mostre a forma geral da escada **if-else-if**.

```
if(condição)
    instrução;
else if(condição)
    instrução;
else if(condição)
    instrução;
.
.
.
else
    instrução;
```

3. Dado o código

```
if(x < 10)
    if(y > 100) {
        if(!done) x = z;
        else y = z;
    }
else System.out.println("error"); // que if?
```

a que **if** o último **else** está associado?

O último **else** está associado a **if(y > 100)**.

4. Mostre a instrução **for** de um laço que conte de 1.000 a 0 em intervalos de -2.

```
| for(int i = 1000; i >= 0; i -= 2) // ...
```

5. O fragmento a seguir é válido?

```
for(int i = 0; i < num; i++)
    sum += i;

count = i;
```

Não, **i** não é conhecida fora do laço **for** em que é declarada.

6. Explique o que **break** faz. Certifique-se de explicar suas duas formas.

Um **break** sem rótulo causa o encerramento da instrução **switch** ou do laço imediatamente externo.

Um **break** com rótulo faz o controle ser transferido para o fim da instrução ou do bloco rotulado.

7. No fragmento a seguir, após a instrução **break** ser executada, o que é exibido?

```
for(i = 0; i < 10; i++) {
    while(running) {
        if(x<y) break;
        // ...
    }
    System.out.println("after while");
}
System.out.println("After for");
```

Após **break** ser executada, o string "after while" é exibido.

8. O que o fragmento abaixo exibe?

```
for(int i = 0; i<10; i++) {
    System.out.print(i + " ");
    if((i%2) == 0) continue;
    System.out.println();
}
```

A resposta é esta:

```
0 1
2 3
4 5
6 7
8 9
```

9. Nem sempre a expressão de iteração de um laço **for** necessita alterar a variável de controle de laço adicionando ou subtraindo um valor fixo. Em vez disso, a variável de controle pode mudar de alguma maneira arbitrária. Usando esse conceito, escreva um programa que use um laço **for** para gerar e exibir a progressão 1, 2, 4, 8, 16, 32 e assim por diante.

```
/* Usa um laço for para gerar a progressão
   1 2 4 8 16 32 64
*/
class Progress {
    public static void main(String[] args) {

        for(int i = 1; i < 100; i += i)
            System.out.print(i + " ");
    }
}
```

10. As letras minúsculas ASCII ficam separadas das maiúsculas por um intervalo igual a 32. Logo, para converter uma letra minúscula em maiúscula, temos que subtrair 32 dela. Use essa informação para escrever um programa que leia caracteres do teclado. Ele deve converter todas as letras minúsculas em maiúsculas e todas as letras maiúsculas em minúsculas, exibindo o resultado. Não faça alterações em nenhum outro caractere. O programa será encerrado quando o usuário inserir um ponto. No fim, ele deve exibir quantas alterações ocorreram na caixa das letras.

```
// Altera a caixa das letras.
class CaseChg {
    public static void main(String[] args)
        throws java.io.IOException {

        char ch;
        int changes = 0;

        System.out.println("Enter period to stop.");

        do {
            ch = (char) System.in.read();
            if(ch >= 'a' & ch <= 'z') {
                ch -= 32;
                changes++;
                System.out.println(ch);
            }
        }
```

```
        else if(ch >= 'A' & ch <= 'Z') {
            ch += 32;
            changes++;
            System.out.println(ch);
        }
    } while(ch != '.');
System.out.println("Case changes: " + changes);
}
}
```

11. O que é um laço infinito?

Um laço infinito é aquele que é executado incessantemente.

12. No uso de **break** com um rótulo, este deve estar em uma instrução ou bloco que contenha **break**?

Sim.

CAPÍTULO 4

1. Qual é a diferença entre uma classe e um objeto?

Classe é uma abstração lógica que descreve a forma e o comportamento de um objeto. Objeto é uma instância física da classe.

2. Como uma classe é definida?

Uma classe é definida com o uso da palavra-chave **class**. Dentro da instrução **class**, você deve especificar o código e os dados que compõem a classe.

3. Cada objeto tem sua própria cópia de quê?

Cada objeto de uma classe tem sua própria cópia das variáveis de instância da classe.

4. Usando duas instruções separadas, mostre como declarar um objeto de nome **counter** de uma classe chamada **MyCounter** e atribua a ela um novo objeto dessa classe.

```
MyCounter counter;
counter = new MyCounter();
```

5. Mostre como um método chamado **myMeth()** será declarado se tiver um tipo de retorno **double** e dois parâmetros **int** chamados **a** e **b**.

```
double myMeth(int a, int b) { // ...
```

6. Como um método deve retornar se um valor for retornado?

Um método que retorna um valor deve retornar por intermédio da instrução **return**, passando o valor de retorno ao fazer isso.

7. Que nome tem um construtor?

Um construtor tem o mesmo nome de sua classe.

8. O que **new** faz?

O operador **new** aloca memória para um objeto e o inicializa usando seu construtor.

9. O que é coleta de lixo e como ela funciona? O que é **finalize()**?

Coleta de lixo é o mecanismo que recicla objetos não utilizados para que sua memória possa ser reutilizada. O método **finalize()** de um objeto é chamado imediatamente antes de ele ser reciclado. Ele não é usado pela maioria dos programas Java.

10. O que é **this**?

A palavra-chave **this** é uma referência ao objeto em que um método é chamado. Ela é passada automaticamente para o método.

11. Um construtor pode ter um ou mais parâmetros?

Sim.

12. Se um método não retornar um valor, qual deve ser seu tipo de retorno?

void

CAPÍTULO 5**1.** Mostre duas maneiras de declarar um array unidimensional de 12 **doubles**.

```
| double x[] = new double[12];
| double[] x = new double[12];
```

2. Mostre como inicializar um array unidimensional de inteiros com os valores de 1 a 5.

```
| int[] x = { 1, 2, 3, 4, 5 };
```

3. Escreva um programa que use um array para encontrar a média de 10 valores **double**. Use os 10 valores que quiser.

```
// Encontra a média de 10 valores double.
class Avg {
    public static void main(String[] args) {
        double[] nums = { 1.1, 2.2, 3.3, 4.4, 5.5,
                         6.6, 7.7, 8.8, 9.9, 10.1 };
        double sum = 0;

        for(int i=0; i < nums.length; i++)
            sum += nums[i];

        System.out.println("Average: " + sum / nums.length);
    }
}
```

4. Altere a classificação da seção Tente isto 5-1 para que classifique um array de strings. Demonstre que funciona.

```
// Demonstra a classificação por bolha com strings.
class StrBubble {
    public static void main(String[] args) {
        String[] strs = {
            "this", "is", "a", "test",
            "of", "a", "string", "sort"
        };
        int a, b;
        String t;
        int size;

        size = strs.length; // número de elementos a serem classificados

        // exibe o array original
        System.out.print("Original array is:");
        for(int i=0; i < size; i++)
            System.out.print(" " + strs[i]);
        System.out.println();

        // Esta é a classificação por bolha para strings.
        for(a=1; a < size; a++) {
            for(b=size-1; b >= a; b--) {
                if(strs[b-1].compareTo(strs[b]) > 0) { // se estiver fora de ordem,
                    // troca os elementos
                    t = strs[b-1];
                    strs[b-1] = strs[b];
                    strs[b] = t;
                }
            }
        }

        // exibe o array classificado
        System.out.print("Sorted array is:");
        for(int i=0; i < size; i++)
            System.out.print(" " + strs[i]);
        System.out.println();
    }
}
```

5. Qual é a diferença entre os métodos **indexOf()** e **lastIndexOf()** de **String**?

O método **indexOf()** encontra a primeira ocorrência do substring especificado. **lastIndexOf()** encontra a última ocorrência.

6. Já que todos os strings são objetos de tipo **String**, mostre como chamar os métodos **length()** e **charAt()** no seguinte literal de string: "I like Java".

Mesmo parecendo estranha, esta é uma chamada válida a **length()**:

```
| System.out.println("I like Java".length());
```

A saída exibida é 11. **charAt()** é chamado de forma semelhante.

7. Expandindo a classe **SimpleCipher**, modifique-a para que use um string de oito caracteres como chave.

```
// Codificação XOR melhorada.
class SimpleCipher2 {
    public static void main(String[] args) {
        String msg = "This is a test";
        String encMsg = "";
        String decMsg = "";
        String key = "abcdefgi";
        int j;

        System.out.print("Original message: ");
        System.out.println(msg);

        // codifica a mensagem
        j = 0;
        for(int i=0; i < msg.length(); i++) {
            encMsg = encMsg + (char) (msg.charAt(i) ^ key.charAt(j));
            j++;
            if(j==8) j = 0;
        }

        System.out.print("Encoded message: ");
        System.out.println(encMsg);

        // decodifica a mensagem
        j = 0;
        for(int i=0; i < msg.length(); i++) {
            decMsg = decMsg + (char) (encMsg.charAt(i) ^ key.charAt(j));
            j++;
            if(j==8) j = 0;
        }

        System.out.print("Decoded message: ");
        System.out.println(decMsg);
    }
}
```

8. Os operadores bitwise podem ser aplicados ao tipo **double**?

Não.

9. Mostre como a sequência a seguir pode ser reescrita com o uso do operador **?**.

```
if(x < 0) y = 10;
else y = 20;
```

Esta é a resposta:

```
y = x < 0 ? 10 : 20;
```

10. No fragmento a seguir, **&** é um operador bitwise ou lógico? Por quê?

```
boolean a, b;  
// ...  
if(a & b) ...
```

É um operador lógico porque os operandos são de tipo **boolean**.

11. É um erro ultrapassar o fim de um array?

Sim.

E indexar um array com um valor negativo?

Sim. Os índices de todos os arrays começam em zero.

12. Qual é o símbolo usado para o operador de deslocamento para a direita sem sinal?

>>>

13. Reescreva a classe **MinMax** mostrada anteriormente neste capítulo para que use um laço **for** de estilo for-each.

```
// Encontra os valores mínimo e máximo de um array.  
class MinMax {  
    public static void main(String[] args) {  
        int[] nums = new int[10];  
        int min, max;  
  
        nums[0] = 99;  
        nums[1] = -10;  
        nums[2] = 100123;  
        nums[3] = 18;  
        nums[4] = -978;  
        nums[5] = 5623;  
        nums[6] = 463;  
        nums[7] = -9;  
        nums[8] = 287;  
        nums[9] = 49;  
  
        min = max = nums[0];  
        for(int v : nums) {  
            if(v < min) min = v;  
            if(v > max) max = v;  
        }  
        System.out.println("min and max: " + min + " " + max);  
    }  
}
```

14. Os laços **for** que executam a classificação na classe **Bubble** mostrada na seção Tente isto 5-1 podem ser convertidos em laços de estilo for-each? Em caso negativo, por que não?

Não, os laços **for** da classe **Bubble** que executam a classificação não podem ser convertidos em laços de estilo for-each. No caso do laço externo, o valor atual de seu contador é usado pelo laço interno. No caso do laço interno, valores fora de ordem devem ser trocados, o que implica atribuições. Atribuições ao array subjacente não podem ocorrer com o uso de um laço de estilo for-each.

- 15.** Um **String** pode controlar uma instrução **switch**?

A partir de JDK 7, a resposta é sim.

CAPÍTULO 6

- 1.** Dado este fragmento,

```
class X {
    private int count;
```

o fragmento a seguir está correto?

```
class Y {
    public static void main(String[] args) {
        X ob = new X();
        ob.count = 10;
```

Não, um membro **private** não pode ser acessado fora de sua classe.

- 2.** Um modificador de acesso deve _____ a declaração de um membro.

| preceder

- 3.** Dada esta classe,

```
class Test {
    int a;
    Test(int i) { a = i; }
```

crie um método chamado **swap()** que troque o conteúdo dos objetos referenciados por duas referências de objeto **Test**.

```
void swap(Test ob1, Test ob2) {
    int t;

    t = ob1.a;
    ob1.a = ob2.a;
    ob2.a = t;
}
```

- 4.** O fragmento a seguir está correto?

```
class X {
    int meth(int a, int b) { ... }
    String meth(int a, int b) { ... }
```

Não. Métodos sobrecarregados podem ter diferentes tipos de retorno, mas não tomam parte na definição da sobrecarga. Eles *devem* ter listas de parâmetros diferentes.

- 5.** Crie um método recursivo que exiba o conteúdo de um string de trás para frente.

```
// Exibe um string de trás para frente usando a recursão.
class Backwards {
    String str;
```

```
    Backwards(String s) {
        str = s;
    }

    void backward(int idx) {
        if(idx != str.length()-1) backward(idx+1);

        System.out.print(str.charAt(idx));
    }
}

class BWDemo {
    public static void main(String[] args) {
        Backwards s = new Backwards("This is a test");

        s.backward(0);
    }
}
```

6. Se todos os objetos de uma classe tiverem que compartilhar a mesma variável, como você deve declarar essa variável?

Variáveis compartilhadas são declaradas como **static**.

7. Por que você pode ter que usar um bloco **static**?

Um bloco **static** é usado para executar alguma inicialização relacionada à classe, antes que um objeto seja criado.

8. O que é uma classe interna?

Uma classe interna é uma classe aninhada não estática.

9. Para que um membro só possa ser acessado por outros membros de sua classe, que modificador de acesso deve ser usado?

private

10. O nome de um método mais sua lista de parâmetros compõem a _____ do método.

assinatura

11. Um argumento **int** é passado para um método com o uso da chamada por _____.

valor

12. Crie um método varargs chamado **sum()** que some os valores **int** passados para ele. Faça-o retornar o resultado. Demonstre seu uso.

Há muitas maneiras de criar uma solução. Esta é uma:

```
class SumIt {
    int sum(int ... n) {
        int result = 0;

        for(int i = 0; i < n.length; i++)
```

```

        result += n[i];

        return result;
    }
}

class SumDemo {
    public static void main(String[] args) {

        SumIt siObj = new SumIt();

        int total = siObj.sum(1, 2, 3);
        System.out.println("Sum is " + total);

        total = siObj.sum(1, 2, 3, 4, 5);
        System.out.println("Sum is " + total);
    }
}

```

- 13.** Um método varargs pode ser sobrecarregado?

Sim.

- 14.** Mostre um exemplo de um método varargs sobrecarregado que seja ambíguo.

Aqui está um exemplo de um método varargs sobrecarregado que é ambíguo:

```

double myMeth(double ... v) { // ...
double myMeth(double d, double ... v) { // ...

```

Se você tentar chamar **myMeth()** com um argumento, desta forma,

```
| myMeth(1.1);
```

o compilador não conseguirá determinar que versão do método deve chamar.

CAPÍTULO 7

- 1.** Uma superclasse tem acesso aos membros de uma subclasse? E a subclasse pode acessar os membros de uma superclasse?

Não, uma superclasse não conhece suas subclasses, mas a subclasse tem acesso a todos os membros não privados de sua superclasse.

- 2.** Crie uma subclasse de **TwoDShape** chamada **Circle**. Inclua um método **area()** que calcule a área do círculo e um construtor que use **super** para inicializar a parte referente a **TwoDShape**.

```

// Subclasse de TwoDShape para círculos.
class Circle extends TwoDShape {
    // Construtor padrão.
    Circle() {
        super();
    }
}

```

```
// Constrói Circle
Circle(double x) {
    super(x, "circle"); // chama o construtor da superclasse
}

// Constrói um objeto a partir de outro.
Circle(Circle ob) {
    super(ob); // passa o objeto para o construtor de TwoDShape
}

double area() {
    return (getWidth() / 2) * (getWidth() / 2) * 3.1416;
}
```

3. Como impedir que uma subclasse tenha acesso a um membro de uma superclasse?

Para impedir que uma subclasse tenha acesso a um membro da superclasse, declare esse membro como **private**.

4. Descreva a finalidade e a aplicação das duas versões de **super**.

A palavra-chave **super** tem duas formas. A primeira é usada para chamar um construtor da superclasse. A forma geral dessa aplicação é

`super(lista-parâm);`

A segunda forma de **super** é usada para acessar um membro da superclasse. Esta é sua forma geral:

`super.membro`

5. Dada a hierarquia a seguir, em que ordem os construtores dessas classes são executados quando um objeto **Gamma** é instanciado?

```
class Alpha { ... }

class Beta extends Alpha { ... }

Class Gamma extends Beta { ... }
```

Os construtores são sempre chamados em ordem de derivação. Logo, quando um objeto **Gamma** é criado, a ordem é **Alpha, Beta, Gamma**.

6. Uma referência da superclasse pode referenciar um objeto da subclasse. Explique por que isso é importante no âmbito da sobreposição de métodos.

Quando um método sobreposto é chamado por intermédio de uma referência da superclasse, é o tipo de objeto que está sendo referenciado que determina qual versão do método será chamada.

7. O que é uma classe abstrata?

Uma classe com pelo menos um método abstrato é abstrata.

- 8.** Como impedir que um método seja sobreposto? E que uma classe seja herdada?

Para impedir que um método seja sobreposto, declare-o como **final**. Para impedir que uma classe seja herdada, declare-a também como **final**.

- 9.** Explique como a herança, a sobreposição de métodos e as classes abstratas são usadas para dar suporte ao polimorfismo.

A herança, a sobreposição de métodos e as classes abstratas dão suporte ao polimorfismo permitindo a criação de uma estrutura de classes generalizada que possa ser estendida por várias classes. Logo, a classe abstrata define uma interface coerente que é compartilhada por todas as classes que a implementam. Essa abordagem personifica o conceito de “uma interface, vários métodos”.

- 10.** Que classe é superclasse de todas as outras classes?

A classe **Object**.

- 11.** Uma classe que contém pelo menos um método abstrato deve ser declarada como abstrata. Verdadeiro ou falso?

Verdadeiro.

- 12.** Que palavra-chave é usada para criar uma constante nomeada?

final

CAPÍTULO 8

- 1.** “Uma interface, vários métodos” é um princípio-chave de Java. Que recurso o exemplifica melhor?

A interface exemplifica melhor o princípio “uma interface, vários métodos” da programação orientada a objetos.

- 2.** Quantas classes podem implementar uma interface?

Uma interface pode ser implementada por um número ilimitado de classes.

- 3.** Quantas interfaces uma classe pode implementar?

Uma classe pode implementar quantas interfaces quiser.

- 4.** Uma classe declara que implementa uma interface usando uma _____ cláusula **implements**

- 5.** As interfaces podem ser estendidas?

Sim, as interfaces podem ser estendidas por outras interfaces.

- 6.** Crie uma interface para a classe **Vehicle** do Capítulo 7. Chame-a de **IVehicle**.

```
interface IVehicle {
    // Retorna a autonomia.
    int range();
```

```
// Calcula o combustível necessário para percorrer a distância dada.  
double fuelNeeded(int miles);  
  
// Métodos acessadores.  
int getPassengers();  
void setPassengers(int p);  
int getFuelCap();  
void setFuelCap(int f);  
int getMpg();  
void setMpg(int m);  
}
```

7. As variáveis declaradas em uma interface são implicitamente **static** e **final**. Para que servem?

As variáveis de interface criam constantes nomeadas que podem ser compartilhadas por todas as classes de um programa. Elas ganham visibilidade implementando sua interface.

8. Uma interface pode ser membro de outra?

Sim.

9. Dadas duas interfaces chamadas **Alpha** e **Beta**, mostre como uma classe chamada **MyClass** especificaria que implementa as duas.

```
| class MyClass implements Alpha, Beta { // ...
```

CAPÍTULO 9

1. Usando o código da seção Tente isto 8-1, insira a interface **ISimpleStack** e suas duas implementações em um pacote chamado **stackpack**. Mantendo a classe de demonstração de pilha **ISimpleStackDemo** no pacote padrão, mostre como importar e usar as classes de **stackpack**.

Para inserir **ISimpleStack** e suas implementações no pacote **stackpack**, torne **public** cada classe da implementação e adicione essa instrução ao início de cada arquivo.

```
| package stackpack;
```

Após fazer isso, você poderá usar **stackpack** adicionando essa instrução **import** a **ISimpleStackDemo**.

```
| import stackpack.*;
```

2. O que é espaço de nome? Por que é importante Java permitir que você divida o espaço de nome?

Um espaço de nome é uma região declarativa. Ao dividir o espaço de nome, você poderá evitar colisões de nomes.

3. Os pacotes são armazenados em _____.
diretórios

- 4.** Explique a diferença entre **protected** e o acesso padrão.

Um membro com acesso **protected** pode ser usado dentro de seu pacote e por uma subclasse de qualquer pacote. Um membro com acesso padrão só pode ser usado dentro de seu pacote.

- 5.** Explique as duas maneiras pelas quais os membros de um pacote podem ser acessados por outros pacotes.

Para usar um membro de um pacote, você pode qualificar totalmente o seu nome ou importá-lo usando **import**.

- 6.** Um pacote é, basicamente, um contêiner para classes. Verdadeiro ou falso?

Verdadeiro.

- 7.** Que pacote Java padrão é importado automaticamente para um programa?

java.lang

- 8.** Diga em suas próprias palavras o que faz a importação estática.

A importação estática traz para o espaço de nome global os membros estáticos de uma classe ou interface. Ou seja, os membros estáticos podem ser usados sem precisarmos qualificá-los com o nome de sua classe ou interface.

- 9.** O que a instrução a seguir faz?

```
| import static somepack.SomeClass.myMethod;
```

A instrução traz para o espaço de nome global o método **SomeClass.myMethod()**, que faz parte do pacote **somepack**.

- 10.** A importação estática foi projetada para situações especiais ou é boa prática dar visibilidade a todos os membros estáticos de todas as classes?

A importação estática foi projetada para casos especiais. Dar visibilidade a muitos membros estáticos levará a colisões de espaço de nome e desestruturará o código.

CAPÍTULO 10

- 1.** Que classe fica no topo da hierarquia de exceções?

Throwable fica no topo da hierarquia de exceções.

- 2.** Explique resumidamente como **try** e **catch** são usados.

As instruções **try** e **catch** funcionam em conjunto. As instruções do programa que você quiser monitorar em busca de exceções ficarão dentro de um bloco **try**. Uma exceção é capturada com o uso de **catch**.

- 3.** O que está errado no fragmento a seguir?

```
// ...
vals[18] = 10;
catch (ArrayIndexOutOfBoundsException exc) {
    // trata erro
}
```

Não há um bloco **try** antes da instrução **catch**.

4. O que acontece quando uma exceção não é capturada?

Se uma exceção não for capturada, isso resultará no encerramento anormal do programa.

5. O que está errado no fragmento a seguir?

```
class A extends Exception { ... }

class B extends A { ... }

// ...

try {
    // ...
}
catch (A exc) { ... }
catch (B exc) { ... }
```

No fragmento, um **catch** da superclasse precede um **catch** da subclasse. Como o **catch** da superclasse também captura todas as subclasses, cria-se um código inalcançável.

6. Um **catch** interno pode relançar uma exceção para um **catch** externo?

Sim, uma exceção pode ser relançada.

7. O bloco **finally** é a última parte do código executada antes de o programa terminar. Verdadeiro ou falso? Explique sua resposta.

Falso. O bloco **finally** é o código executado quando um bloco **try** termina.

8. Que tipo de exceções deve ser declarado explicitamente na cláusula **throws** de um método?

Todas as exceções, exceto as de tipo **RuntimeException** e **Error**, devem ser declaradas em uma cláusula **throws**.

9. O que está errado neste fragmento?

```
class MyClass { // ... }
// ...
throw new MyClass();
```

MyClass não estende **Throwable**. Só subclasses de **Throwable** podem ser lançadas por **throw**.

10. Quais são as três maneiras pelas quais uma exceção pode ser gerada?

Uma exceção pode ser gerada por um erro na JVM, por um erro no programa ou explicitamente via uma instrução **throw**.

11. Quais são as duas subclasses diretas de **Throwable**?

Error e **Exception**

12. O que é o recurso multi-catch?

O recurso multi-catch permite que uma cláusula **catch** capture duas ou mais exceções.

13. Normalmente um código deve capturar exceções de tipo **Error**?

Não.

CAPÍTULO 11

- 1.** Por que Java define fluxos tanto de bytes quanto de caracteres?

Os fluxos de bytes são os fluxos originais definidos por Java. Eles são úteis principalmente para I/O binário e dão suporte a arquivos de acesso aleatório. Os fluxos de caracteres são otimizados para caracteres Unicode.

- 2.** Já que a entrada e a saída do console são baseadas em texto, por que Java ainda usa fluxos de bytes para esse fim?

Os fluxos predefinidos, **System.in**, **System.out** e **System.err**, foram criados antes de Java adicionar os fluxos de caracteres.

- 3.** Mostre como abrir um arquivo para a leitura de bytes.

Esta é uma maneira de abrir um arquivo para a leitura de bytes:

```
| FileInputStream fin = new FileInputStream("test");
```

- 4.** Mostre como abrir um arquivo para a leitura de caracteres.

Esta é uma maneira de abrir um arquivo para a leitura de caracteres:

```
| FileReader fr = new FileReader("test");
```

- 5.** Mostre como abrir um arquivo para I/O de acesso aleatório.

Esta é uma maneira de abrir um arquivo para o acesso aleatório:

```
| RandomAccessFile randfile = new RandomAccessFile("test", "rw");
```

- 6.** Como podemos converter um string numérico como "123.23" em seu equivalente binário?

Uma maneira de converter strings numéricos em seus equivalentes binários é usando os métodos de análise definidos pelos encapsuladores de tipo, como **Integer** ou **Double**.

- 7.** Escreva um programa que copie um arquivo de texto. No processo, faça-o converter todos os espaços em hífens. Use as classes de fluxos de bytes de arquivos. Use a abordagem tradicional para fechar um arquivo chamando **close()** explicitamente.

```
/* Copia um arquivo de texto, substituindo espaços por hífens.
```

Esta versão utiliza fluxos de bytes.

Para usar este programa, especifique o nome
do arquivo de origem e do arquivo de destino.

Por exemplo,

```
java Hyphen source target
```

```
*/
```

```
import java.io.*;
```

```
class Hyphen {
    public static void main(String[] args)
    {
        int i;
        FileInputStream fin = null;
        FileOutputStream fout = null;

        // Primeiro verifica se os dois arquivos foram especificados.
        if(args.length != 2 ) {
            System.out.println("Usage: Hyphen From To");
            return;
        }

        // Copia o arquivo e insere os hífens.
        try {
            fin = new FileInputStream(args[0]);
            fout = new FileOutputStream(args[1]);

            do {
                i = fin.read();

                // converte o espaço em um hífen
                if((char)i == ' ') i = '-';

                if(i != -1) fout.write(i);
            } while(i != -1);
        } catch(IOException exc) {
            System.out.println("I/O Error: " + exc);
        } finally {
            try {
                if(fin != null) fin.close();
            } catch(IOException exc) {
                System.out.println("Error closing input file.");
            }

            try {
                if(fout != null) fout.close();
            } catch(IOException exc) {
                System.out.println("Error closing output file.");
            }
        }
    }
}
```

8. Reescreva o programa do Exercício 7 para que use as classes de fluxos de caracteres. Dessa vez, use a instrução **try-with-resources** para fechar automaticamente o arquivo.

```
/* Copia um arquivo de texto, substituindo espaços por hífens.

   Esta versão usa fluxos de caracteres.
```

Para usar este programa, especifique o nome do arquivo de origem e do arquivo de destino. Por exemplo,

```
java Hyphen2 source target
```

Este código requer JDK 7 ou posterior.

```
/*
import java.io.*;

class Hyphen2 {
    public static void main(String[] args)
    {
        int i;

        // Primeiro verifica se os dois arquivos foram especificados.
        if(args.length !=2 ) {
            System.out.println("Usage: CopyFile From To");
            return;
        }

        // Copia o arquivo e insere os hífens.
        // Usa a instrução try-with-resources.
        try (FileReader fin = new FileReader(args[0]);
             FileWriter fout = new FileWriter(args[1]))
        {
            do {
                i = fin.read();

                // converte o espaço em um hífen
                if((char)i == ' ') i = '-';

                if(i != -1) fout.write(i);
            } while(i != -1);
        } catch(IOException exc) {
            System.out.println("I/O Error: " + exc);
        }
    }
}
```

9. Que tipo de fluxo é **System.in**?

InputStream

10. O que o método **read()** de **InputStream** retorna quando o fim do fluxo é alcançado?

-1

11. Que tipo de fluxo é usado na leitura de dados binários?

DataInputStream

12. Reader e Writer estão no topo da hierarquia de classes _____.
de I/O baseado em caracteres.
13. A instrução try-with-resources é usada para _____.
gerenciamento automático de recursos
14. Quando usamos o método tradicional de fechamento de arquivo, geralmente fechar um arquivo dentro de um bloco finally é uma boa abordagem. Verdadeiro ou falso?
Verdadeiro.
15. Que classe dá acesso aos atributos de um arquivo?
File
16. Você pode usar a classe File para excluir um arquivo?
Sim.

CAPÍTULO 12

1. Como o uso de várias threads Java permite escrever programas mais eficientes?

O uso de várias threads permite tirar vantagem do tempo ocioso existente em quase todos os programas. Quando uma thread não pode ser executada, outra pode. Em sistemas *multicore*, duas ou mais threads podem ser executadas simultaneamente.

2. O uso de várias threads é suportado pela classe _____ e pela interface _____.

Thread e **Runnable**

3. Na criação de um objeto executável, por que pode ser melhor estender **Thread** em vez de implementar **Runnable**?

Pode ser melhor estender **Thread** quando você quiser sobrepor um ou mais de seus métodos além de **run()**.

4. Mostre como podemos usar **join()** para esperar um objeto de thread chamado **MyThrd** terminar.

```
| myThrd.join();
```

5. Mostre como configurar uma thread chamada **MyThrd** com três níveis acima da prioridade normal.

```
| myThrd.setPriority(Thread.NORM_PRIORITY+3);
```

6. Qual é o efeito da inclusão da palavra-chave **synchronized** em um método?

A inclusão de **synchronized** em um método permite que apenas uma thread de cada vez use o método para qualquer objeto de sua classe.

7. Os métodos **wait()** e **notify()** são usados na execução da _____.
comunicação entre threads

8. Altere a classe **TickTock** para que ela marque realmente o tempo. Isto é, faça cada tique levar meio segundo e cada taque levar mais meio segundo. Logo, cada tique-taque levará um segundo. (Não se preocupe com o tempo necessário para alternar tarefas, etc.)

Para fazer a classe **TickTock** marcar realmente o tempo, apenas adicione chamadas a `sleep()`, como mostrado aqui:

```
// Faz a classe TickTock marcar realmente o tempo.

class TickTock {

    String state; // contém o estado do relógio

    synchronized void tick(boolean running) {
        if(!running) { // interrompe o relógio
            state = "ticked";
            notify(); // notifica as threads que estiverem esperando
            return;
        }

        System.out.print("Tick ");

        // espera 1/2 segundo
        try {
            Thread.sleep(500);
        } catch(InterruptedException exc) {
            System.out.println("Thread interrupted.");
        }
    }

    state = "ticked"; // configura o estado atual com ticked

    notify(); // deixa tock() ser executado
    try {
        while(!state.equals("tocked"))
            wait(); // espera tock() terminar
    }
    catch(InterruptedException exc) {
        System.out.println("Thread interrupted.");
    }
}

synchronized void tock(boolean running) {
    if(!running) { // interrompe o relógio
        state = "tocked";
        notify(); // notifica as threads que estiverem esperando
        return;
    }

    System.out.println("Tock");
}
```

```
// espera 1/2 segundo
try {
    Thread.sleep(500);
} catch(InterruptedException exc) {
    System.out.println("Thread interrupted.");
}

state = "locked"; // configura o estado atual com locked

notify(); // deixa tick() ser executado
try {
    while(!state.equals("ticked"))
        wait(); // espera tick() terminar
}
catch(InterruptedException exc) {
    System.out.println("Thread interrupted.");
}
}
```

9. Por que você não pode usar **suspend**, **resume()** e **stop()** em programas novos?

Os métodos **suspend()**, **resume()** e **stop()** foram substituídos porque podem causar problemas sérios de tempo de execução.

10. Que método definido por **Thread** obtém o nome de uma thread?

getName()

11. O que **isAlive()** retorna?

Retorna **true** se a thread chamadora ainda estiver em execução, e **false** se ela tiver terminado.

CAPÍTULO 13

1. Diz-se que as constantes de enumeração são *autotipadas*. O que isso significa?

No termo *autotipada*, o “auto” se refere ao tipo da enumeração em que a constante é definida. Logo, uma constante de enumeração é um objeto da enumeração do qual ela faz parte.

2. Que classe todas as enumerações herdam automaticamente?

A classe **Enum** é herdada automaticamente por todas as enumerações.

3. Dada a enumeração a seguir, escreva um programa que use **values()** para exibir uma lista das constantes e seus valores ordinais.

```
enum Tools {
    SCREWDRIVER, WRENCH, HAMMER, PLIERS
}
```

A solução é:

```
enum Tools {
    SCREWDRIVER, WRENCH, HAMMER, PLIERS
}

class ShowEnum {
    public static void main(String[] args) {
        for(Tools d : Tools.values())
            System.out.print(d + " has ordinal value of " +
                d.ordinal() + '\n');
    }
}
```

4. A simulação de semáforo desenvolvida na seção Tente isto 13-1 pode ser melhorada com algumas alterações simples que se beneficiem dos recursos de classe da enumeração. Na versão mostrada, a duração de cada sinal era controlada pela classe **TrafficLightSimulator** com os valores sendo embutidos no método **run()**. Altere isso para que a duração de cada sinal seja armazenada pelas constantes da enumeração **TrafficLightColor**. Para fazê-lo, você terá que adicionar um construtor, uma variável de instância privada e um método chamado **getDelay()**.

A versão melhorada da simulação de semáforo é mostrada abaixo. Há duas melhorias importantes. Em primeiro lugar, agora o retardo do sinal está vinculado ao seu valor na enumeração, o que dá mais estrutura ao código. Em segundo, o método **run()** não precisa mais utilizar uma instrução **switch** para determinar a duração do retardo. Em vez disso, **sleep()** recebe **tlc.getDelay()**, que faz o retardo associado ao sinal atual ser usado automaticamente.

```
// Versão melhorada da simulação de semáforo que
// armazena o retardo do sinal em TrafficLightColor.

// Enumeração com as cores de um semáforo.
enum TrafficLightColor {
    RED(12000), GREEN(10000), YELLOW(2000);

    private int delay;

    TrafficLightColor(int d) {
        delay = d;
    }

    int getDelay() { return delay; }
}

// Semáforo computadorizado.
class TrafficLightSimulator implements Runnable {
    private Thread thrd; // contém a thread que executa a simulação
    private TrafficLightColor tlc; // contém a cor do sinal atual
    boolean stop = false; // configura com true para interromper a simulação
    boolean changed = false; // true quando o sinal mudou
```

```
TrafficLightSimulator(TrafficLightColor init) {
    tlc = init;

    thrd = new Thread(this);
    thrd.start();
}

TrafficLightSimulator() {
    tlc = TrafficLightColor.RED;

    thrd = new Thread(this);
    thrd.start();
}

// Inicia o semáforo.
public void run() {
    while(!stop) {
        // Observe como esse código foi simplificado!
        try {
            Thread.sleep(tlc.getDelay());
        } catch(InterruptedException exc) {
            System.out.println(exc);
        }

        changeColor();
    }
}

// Muda a cor.
synchronized void changeColor() {
    switch(tlc) {
        case RED:
            tlc = TrafficLightColor.GREEN;
            break;
        case YELLOW:
            tlc = TrafficLightColor.RED;
            break;
        case GREEN:
            tlc = TrafficLightColor.YELLOW;
    }

    changed = true;
    notify(); // sinaliza que a cor mudou
}

// Espera até uma mudança de sinal ocorrer.
synchronized void waitForChange() {
    try {
        while(!changed)
            wait(); // espera o sinal mudar
```

```

        changed = false;
    } catch(InterruptedException exc) {
        System.out.println(exc);
    }
}

// Retorna a cor atual.
synchronized TrafficLightColor getColor() {
    return tlc;
}

// Interrrompe o semáforo.
synchronized void cancel() {
    stop = true;
}
}

class TrafficLightDemo {
    public static void main(String[] args) {
        TrafficLightSimulator tl =
            new TrafficLightSimulator(TrafficLightColor.GREEN);

        for(int i=0; i < 9; i++) {
            System.out.println(tl.getColor());
            tl.waitForChange();
        }

        tl.cancel();
    }
}

```

- 5.** Defina boxing e unboxing. Como o *autoboxing/unboxing* afeta essas ações?

Boxing é o processo de armazenar um valor primitivo em um objeto encapsulador de tipo. Unboxing é o processo de recuperar o valor primitivo no encapsulador de tipo. O *autoboxing* encapsula automaticamente um valor primitivo sem ser preciso construir um objeto de maneira explícita. O *autounboxing* recupera automaticamente o valor primitivo em um encapsulador de tipo sem termos que chamar explicitamente um método, como `intValue()`.

- 6.** Altere o fragmento a seguir para que use *autoboxing*.

```
| Short val = new Short(123);
```

A solução é:

```
| Short val = 123;
```

- 7.** Uma anotação é sintaticamente baseada em uma _____.

interface

8. O que é uma anotação marcadora?

Uma anotação marcadora é aquela que não recebe argumentos.

9. Uma anotação só pode ser aplicada a métodos. Verdadeiro ou falso?

Falso. Qualquer tipo de declaração pode ter uma anotação.

CAPÍTULO 14

1. Os genéricos são importantes para Java porque permitem a criação de código

- A. Com segurança de tipos
 - B. Reutilizável
 - C. Confiável
 - D. Todas as alternativas acima
- D. Todas as alternativas acima

2. Um tipo primitivo pode ser usado como argumento de tipo?

Não, os argumentos de tipo devem ser tipos de objeto.

3. Mostre como declarar uma classe chamada **FlightSched** que use dois parâmetros genéricos.

A solução é

```
| class FlightSched<T, V> {
```

4. Usando a resposta à Questão 3, altere o segundo parâmetro de tipo de **FlightSched** para que seja preciso estender **Thread**.

A solução é

```
| class FlightSched<T, V extends Thread> {
```

5. Agora, altere **FlightSched** para que seu segundo parâmetro de tipo seja subclasse do primeiro parâmetro de tipo.

A solução é

```
| class FlightSched<T, V extends T> {
```

6. No que diz respeito aos genéricos, o que é o símbolo ? e o que ele faz?

O símbolo ? é o argumento curinga. Ele equivale a qualquer tipo válido.

7. O argumento curinga pode ser limitado?

Sim, um curinga pode ter um limite superior ou inferior.

8. Um método genérico chamado **MyGen()** tem um parâmetro de tipo. Além disso, **MyGen()** tem um parâmetro cujo tipo é o do parâmetro de tipo. Ele também retorna um objeto desse parâmetro de tipo. Mostre como declarar **MyGen()**.

A solução é

```
| <T> T MyGen(T o) { // ...
```

9. Dada esta interface genérica

```
| interface IGenIF<T, V extends T> { // ...
```

mostre a declaração de uma classe chamada **MyClass** que implemente **IGenIF**.

A solução é

```
| class MyClass<T, V extends T> implements IGenIF<T, V> { // ...
```

10. Dada uma classe genérica chamada **Counter<T>**, mostre como criar um objeto de seu tipo bruto.

Para obter o tipo bruto de **Counter<T>**, simplesmente use seu nome sem nenhuma especificação de tipo, como mostrado aqui:

```
| Counter x = new Counter();
```

11. Existem parâmetros de tipo no tempo de execução?

Não. Todos os parâmetros de tipo são apagados durante a compilação e as conversões apropriadas são feitas. Esse processo se chama erasure.

12. Quando uma classe genérica é herdada, seus parâmetros de tipo também devem ser especificados pela subclasse. Verdadeiro ou falso?

Verdadeiro.

13. O que é **<>**?

O operador losango.

14. Com o uso do JDK 7, como a linha a seguir pode ser simplificada?

```
| MyClass<Double, String> obj = new MyClass<Double, String>(1.1, "Hi");
```

Ela pode ser simplificada com o uso do operador losango, como mostrado aqui:

```
| MyClass<Double, String> obj = new MyClass<>(1.1, "Hi");
```

CAPÍTULO 15

1. Que método é chamado quando um applet é executado pela primeira vez? E qual é chamado quando ele é removido do sistema?

Quando um applet é iniciado, o primeiro método chamado é **init()**. Quando ele é removido, **destroy()** é chamado.

2. Explique por que um applet deve usar várias threads se for executado continuamente.

Um applet deve usar várias threads se for executado continuamente porque os applets não devem entrar em um “modo” de operação. Por exemplo, se **start()** nunca retornar, **paint()** nunca será chamado.

3. Melhore o projeto da seção Tente isto 15-1 para que exiba o string passado como parâmetro. Adicione um segundo parâmetro que especifique o retardo (em milissegundos) existente entre cada giro da mensagem.

```
| /* Applet de banner simples que usa parâmetros.
```

```
/*
import java.awt.*;
import java.applet.*;

/*
<applet code="ParamBanner" width=300 height=50>
<param name=message value=" I like Java! ">
<param name=delay value=500>
</applet>
*/

public class ParamBanner extends Applet implements Runnable {
    String msg;
    int delay;
    Thread t;

    boolean stopFlag;

    // Inicializa t com null.
    public void init() {
        String temp;

        msg = getParameter("message");
        if(msg == null) msg = " Java Rules the Web ";

        temp = getParameter("delay");

        try {
            if(temp != null)
                delay = Integer.parseInt(temp);
            else
                delay = 250; // usa o padrão quando não especificado
        } catch(NumberFormatException exc) {
            delay = 250 ; // usa o padrão em caso de erro
        }

        t = null;
    }

    // Inicia a thread quando o applet é necessário.
    public void start() {
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }

    // Ponto de entrada da thread que executa o banner.
    public void run() {
        // Solicita uma atualização no intervalo especificado.
        for( ; ; ) {
            try {
                repaint();
            }

```

```

        Thread.sleep(delay);
        if(stopFlag) break;
    } catch(InterruptedException exc) {}
}
}

// Pausa o banner.
public void stop() {
    stopFlag = true;
    t = null;
}

// Exibe o banner.
public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;

    g.drawString(msg, 50, 30);
}
}
}

```

4. Desafio extra: crie um applet que exiba a hora atual, atualizada a cada segundo.

```

// Applet de relógio simples.

import java.util.*;
import java.awt.*;
import java.applet.*;
/*
<object code="Clock" width=200 height=50>
</object>
*/

public class Clock extends Applet implements Runnable {
    String msg;
    Thread t;
    Calendar clock;

    boolean stopFlag;

    // Inicializa
    public void init() {
        t = null;
        msg = "";
    }
}

```

```
// Inicia a thread quando o applet é necessário.  
public void start() {  
    t = new Thread(this);  
    stopFlag = false;  
    t.start();  
}  
  
// Ponto de entrada do relógio.  
public void run() {  
    // Solicita uma atualização a cada segundo.  
    for( ; ; ) {  
        try {  
            repaint();  
            Thread.sleep(1000);  
            if(stopFlag) break;  
        } catch(InterruptedException exc) {}  
    }  
}  
  
// Pausa o relógio.  
public void stop() {  
    stopFlag = true;  
    t = null;  
}  
  
// Exibe o relógio.  
public void paint(Graphics g) {  
    clock = Calendar.getInstance();  
  
    msg = "Current time is " +  
          Integer.toString(clock.get(Calendar.HOUR));  
    msg = msg + ":" +  
          Integer.toString(clock.get(Calendar.MINUTE));  
    msg = msg + ":" +  
          Integer.toString(clock.get(Calendar.SECOND));  
  
    g.drawString(msg, 30, 30);  
}
```

5. Para solicitar que a janela de um applet seja reexibida, que método você deve chamar?

repaint()

6. Descreva brevemente a palavra-chave **assert**.

A palavra-chave **assert** cria uma asserção, que é uma condição que deve ser verdadeira durante a execução do programa. Se a asserção for falsa, um **AssertionError** será lançado.

7. Cite uma razão que explique por que um método nativo pode ser útil para alguns tipos de programas.

Um método nativo é útil quando interage com rotinas escritas em linguagens que não sejam Java ou na otimização de um código para um ambiente de tempo de execução específico.

8. Que operador você pode usar para determinar o tipo de um objeto no tempo de execução?

`instanceof`

CAPÍTULO 16

1. Para métodos, classes e variáveis, você deve usar nomes _____ que revelem sua finalidade

2. Todos os princípios discutidos neste capítulo são regras rigorosas que devem ser seguidas para termos um software elegante. Verdadeiro ou falso?

Falso.

3. Um bom programador cria código autodocumentado e, portanto, nunca precisa adicionar comentários internos. Verdadeiro ou falso?

Falso. A documentação interna fornece informações que não estão prontamente disponíveis no código.

4. Geralmente é melhor inserir todos os métodos importantes em uma classe e tornar a maioria das outras classes simples classes secundárias. Verdadeiro ou falso?

Falso.

5. Uma documentação desatualizada é melhor do que nenhuma documentação. Verdadeiro ou falso?

Falso.

6. Um método com encadeamento de métodos na forma `a.getB().getC().getD().doSomething()` não segue a _____.

Lei de Demeter

7. Suponhamos que você tivesse uma classe **Person** com variáveis de instância contendo o nome, a data de nascimento, o endereço, o nome do cônjuge, o nome dos filhos e a ocupação da pessoa. De acordo com o padrão Expert, essa classe deve ter métodos que tratem qualquer tratamento desses dados que outras classes precisem. No entanto, um número ilimitado de tratamentos de um objeto **Person** pode ser executado. Quantos métodos a classe **Person** deve ter para estar correta?

Ela deve incluir todos os métodos essenciais e um conjunto apropriado de métodos de conveniência.

8. Normalmente uma classe tem tanto variáveis de instância quanto métodos. Considere dois casos extremos: uma classe **A** que tem muitas variáveis de instância

e nenhum método e uma classe **B** que tem vários métodos e nenhuma variável de instância. Qual dos princípios discutidos neste capítulo é mais provável que essas classes violem?

O padrão Expert.

9. Quando tentamos falar com um executivo de uma empresa, temos pelo menos três opções:

- Ligar e aguardar até ele estar livre.
 - Continuar ligando em intervalos de minutos até ele estar livre.
 - Deixar uma mensagem pedindo ao executivo que retorne quando estiver livre.
- Qual dessas opções chega mais próxima do padrão Observer?
- O executivo é o assunto/publicador e você é o observador/assinante.

10. Cite uma falha na classe a seguir que a torna deselegante.

```
class NamedObject {  
    private String name;  
    NamedObject(String n) { name = n; }  
    void setName(String n) { name = n; }  
}
```

Não há o método **getName()**, portanto, não há como descobrir qual é o nome. Logo, o nome não tem valor. O problema é que a classe tem uma interface incompleta.

CAPÍTULO 17

1. A maioria dos componentes do AWT é convertida em pares nativos. Por que isso é um problema e como Swing o corrige?

Os pares nativos são problemáticos porque seu uso pode fazer um componente ter uma aparência diferente ou agir de outra forma em plataformas distintas. Já que usam recursos nativos, sua aparência não é alterada facilmente. Eles também têm limitações, como serem opacos. Swing resolve esses problemas usando componentes leves e uma aparência adaptável.

2. A maioria dos componentes de Swing é escrita em código 100% Java. Verdadeiro ou falso?

Verdadeiro.

3. Quais são os quatro contêineres pesados de nível superior?

JFrame, JApplet, JDialog e JWindow

4. Qual é o contêiner de nível superior mais usado para um aplicativo?

JFrame

5. **JFrame** contém vários painéis. A que painel os componentes são adicionados?

Os componentes são adicionados ao painel de conteúdo.

6. Um ouvinte de eventos deve _____ em uma fonte para receber notificações de eventos.

Registrar-se

7. Para receber um evento de ação, uma classe deve implementar que interface?
ActionListener
8. No uso de um **JButton** ou de um **JTextField**, que método deve ser chamado para configurar o comando de ação?
setActionCommand()
9. Cite três gerenciadores de leiaute.

Aqui estão alguns gerenciadores de leiaute: **FlowLayout**, **BorderLayout**, **GridLayout**, **GridBagLayout**, **BoxLayout** e **SpringLayout**.

10. O exemplo do cronômetro da seção Tente isto 17-1 usa dois botões, um para iniciar o cronômetro e o outro para interrompê-lo. No entanto, é possível usar apenas um botão, que se alternaria entre as atividades de iniciar e interromper o cronômetro. Uma maneira de fazer isso é redefinir o texto do botão após cada pressionamento, alternando-o entre Start e Stop. Já que, por padrão, esse texto também é o comando de ação associado ao botão, você pode usar o mesmo botão para dois fins diferentes. Sua missão é reescrever o projeto da seção Tente isto 17-1 para que implemente essa abordagem.

```
// Uma versão do cronômetro da seção Tente isto 17-1 que
// usa um único botão de ação.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

class StopWatch implements ActionListener {

    JLabel jlab;
    long start; // contém a hora inicial em milissegundos
    JButton jbtnStartStop; // um botão que inicia ou interrompe

    StopWatch() {

        // Cria um contêiner JFrame.
        JFrame jfrm = new JFrame("A Simple Stopwatch");

        // Especifica FlowLayout como gerenciador de leiaute.
        jfrm.setLayout(new FlowLayout());

        // Fornece um tamanho inicial para o quadro.
        jfrm.setSize(250, 90);

        // Encerra o programa quando o usuário fecha o aplicativo.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Cria um botão.
        jbtnStartStop = new JButton("Start");
    }
}
```

```
// Adiciona ouvintes de ação.  
jbtnStartStop.addActionListener(this);  
  
// Adiciona os botões ao painel de conteúdo.  
jfrm.add(jbtnStartStop);  
  
// Cria um rótulo baseado em texto.  
jlab = new JLabel("Press Start to begin timing.");  
  
// Adiciona o rótulo ao quadro.  
jfrm.add(jlab);  
  
// Exibe o quadro.  
jfrm.setVisible(true);  
}  
  
// Trata eventos de botão.  
public void actionPerformed(ActionEvent ae) {  
    // obtém a hora atual do sistema  
    Calendar cal = Calendar.getInstance();  
  
    if(ae.getActionCommand().equals("Start")) {  
        // Armazena a hora inicial.  
        start = cal.getTimeInMillis();  
        jlab.setText("Stopwatch is Running...");  
        jbtnStartStop.setText("Stop");  
    }  
    else {  
        // Calcula o tempo decorrido.  
        jlab.setText("Elapsed time is "  
            + (double) (cal.getTimeInMillis() - start)/1000);  
        jbtnStartStop.setText("Start");  
    }  
}  
  
public static void main(String[] args) {  
  
    // Cria o quadro na thread de despacho de evento.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new StopWatch();  
        }  
    });  
}
```

CAPÍTULO 18

1. JLabel gera um ActionEvent?

Não.

2. Que evento é gerado quando um botão de ação é pressionado?

ActionEvent

3. JButton pode incluir um ícone?

Sim.

4. Que controle se alterna entre dois estados: ativo e inativo?

JToggleButton e suas subclasses.

5. No uso de JTextField, o recurso de recortar e colar é suportado por quais métodos?

cut() e paste()

6. Mostre como criar um campo de texto com 32 colunas.

```
| new JTextField(32)
```

7. Podemos definir o comando de ação de um JTextField? Se podemos, como?

Sim, chamando setActionCommand().

8. Que componente de Swing cria uma caixa de seleção? Que evento é gerado quando uma caixa de seleção é marcada ou desmarcada?

JCheckBox cria uma caixa de seleção. Um ItemEvent é gerado quando uma caixa de seleção é marcada ou desmarcada.

9. JRadioButton cria uma lista de botões com a forma de rádios. Verdadeiro ou falso?

Falso.

10. JList exibe uma lista de itens para o usuário selecionar. Verdadeiro ou falso?

Verdadeiro.

11. Que evento é gerado quando o usuário marca ou desmarca um item em um JList?

ListSelectionEvent

12. O que JScrollPane faz?

Rola o conteúdo de outro componente.

13. Que método define o modo de seleção de um JList? Que método obtém o índice do primeiro item selecionado?

setSelectionMode() define o modo de seleção. getSelectedIndex() obtém o índice do primeiro item selecionado.

14. Para exibir informações em formato tabular, você pode usar _____.

JTable

15. Para exibir informações em formato de árvore, você pode usar _____.

JTree

16. O que é JComboBox?

JComboBox é uma combinação de campo de texto com lista suspensa.

CAPÍTULO 19

1. Quais são as principais classes de menu de Swing?

As classes de menu de Swing são **JMenu**, **JMenuItem** e **JMenuBar**.

2. Que classe cria um menu? Para criarmos a barra principal de menus, que classe usaríamos?

JMenu cria um menu. **JMenuBar** cria uma barra de menus.

3. Que evento é gerado quando um item de menu é selecionado?

Um evento de ação.

4. Imagens não são permitidas em menus. Verdadeiro ou falso?

Falso; imagens são permitidas em menus.

5. Que método adiciona uma barra de menus a uma janela?

Para adicionar uma barra de menus a uma janela, chame **setJMenuBar()**.

6. Que método adiciona um mnemônico a um item de menu?

O método **setMnemonic()**.

7. Um ícone pode ser usado como item de menu? Se puder, isso impede o uso de um nome?

Sim, um ícone pode ser usado como item de menu, e não, isso não impede o uso de um nome.

8. Que classe cria um item de menu de botão de rádio?

JRadioButtonMenuItem.

9. Embora sejam permitidos itens de menu de caixa de seleção, seu uso é desencorajado porque eles dão uma aparência estranha ao menu. Verdadeiro ou falso?

Falso. Os itens de menu de caixa de seleção são opções populares de menu padrão.

10. No decorrer deste capítulo, foram sugeridas várias alterações no programa **MenuDemo** que demonstram recursos adicionais dos menus. Exceto pelos itens de menu dinâmicos da seção Tente isto 19-1, integre as outras alterações ao programa **MenuDemo** original. No processo, reorganize o programa para simplificá-lo usando métodos separados para construir os diversos menus.

```
// Programa de demonstração do menu, versão final.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MenuDemo implements ActionListener {

    JLabel jlab;
    JMenuBar jmb;
```



```
JMenuItem jmiClose = new JMenuItem("Close",
                                  KeyEvent.VK_C);
jmiClose.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_C,
                          InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiSave = new JMenuItem("Save",
                                 KeyEvent.VK_S);
jmiSave.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_S,
                          InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiExit = new JMenuItem("Exit",
                                 KeyEvent.VK_E);
jmiExit.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_E,
                          InputEvent.CTRL_DOWN_MASK));

jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
jmb.add(jmFile);

jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);

}

// Cria o menu Options.
void makeOptionsMenu() {

    // Cria o menu Options.
    JMenu jmOptions = new JMenu("Options");

    // Cria o submenu Colors.
    JMenu jmColors = new JMenu("Colors");

    // Usa caixas de seleção para as cores. Isso permite
    // que o usuário selecione mais de uma cor.
    // Observe que inicialmente Red está marcada.
    JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red", true);
    JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
    JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

    jmColors.add(jmiRed);
    jmColors.add(jmiGreen);
    jmColors.add(jmiBlue);
    jmOptions.add(jmColors);
```

```
// Cria o submenu Priority.  
JMenu jmPriority = new JMenu("Priority");  
  
// Usa botões de rádio para a configuração da prioridade.  
// Isso permite que o menu exiba que prioridade está sendo usada,  
// mas também assegura que uma e apenas uma prioridade  
// possa ser selecionada em um determinado momento. Observe que  
// inicialmente o botão de rádio High foi pressionado.  
JRadioButtonMenuItem jmiHigh =  
    new JRadioButtonMenuItem("High", true);  
JRadioButtonMenuItem jmiLow =  
    new JRadioButtonMenuItem("Low");  
  
jmPriority.add(jmiHigh);  
jmPriority.add(jmiLow);  
jmOptions.add(jmPriority);  
  
// Cria grupo de botões para os itens de menu de botão de rádio.  
ButtonGroup bg = new ButtonGroup();  
bg.add(jmiHigh);  
bg.add(jmiLow);  
  
// Cria o item de menu Reset.  
JMenuItem jmiReset = new JMenuItem("Reset");  
jmOptions.addSeparator();  
jmOptions.add(jmiReset);  
  
jmiRed.addActionListener(this);  
jmiGreen.addActionListener(this);  
jmiBlue.addActionListener(this);  
jmiHigh.addActionListener(this);  
jmiLow.addActionListener(this);  
jmiReset.addActionListener(this);  
  
// Para concluir, adiciona o menu Options inteiro  
// à barra de menus  
jmb.add(jmOptions);  
}  
  
// Cria o menu Help.  
void makeHelpMenu() {  
  
    // Cria o menu Help.  
    JMenu jmHelp = new JMenu("Help");  
    ImageIcon iconAbout = new ImageIcon("AboutIcon.gif");  
    JMenuItem jmiAbout = new JMenuItem("About", iconAbout);  
    jmiAbout.setToolTipText("Info about the MenuDemo program.");  
    jmHelp.add(jmiAbout);  
    jmb.add(jmHelp);
```

```
jmiAbout.addActionListener(this);  
}  
  
// Trata eventos de ação dos itens de menu.  
public void actionPerformed(ActionEvent ae) {  
    // Obtém o comando de ação da seleção no menu.  
    String comStr = ae.getActionCommand();  
  
    // Se o usuário selecionar Exit, encerra o programa.  
    if(comStr.equals("Exit")) System.exit(0);  
  
    // Caso contrário, exibe a seleção.  
    jlab.setText(comStr + " Selected");  
}  
  
public static void main(String[] args) {  
    // Cria a GUI na thread de despacho de evento.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new MenuDemo();  
        }  
    });  
}
```

CAPÍTULO 20

1. Uma caixa de diálogo é uma combinação de dois ou mais componentes que interage com o usuário e espera uma resposta. Verdadeiro ou falso?

Verdadeiro.

2. Que método de **JOptionPane** cria uma caixa de diálogo de entrada? E qual cria uma caixa de diálogo de mensagem?

O método **showInputDialog()** cria uma caixa de diálogo de entrada. O método **showMessageDialog()** cria uma caixa de diálogo de mensagem.

3. Que método de **JOptionPane** você usaria para criar uma caixa de diálogo que confirmasse se o usuário deseja salvar alterações em um documento? Mostre como seria a chamada.

O método é **showConfirmDialog()**. Aqui está um exemplo de como chamá-lo:

```
| showConfirmDialog(null, "Save Changes?");
```

4. Com o uso de uma caixa de diálogo de confirmação, que tipo de retorno indica que o usuário clicou no botão Yes?

O tipo de retorno **YES_OPTION** indica que o usuário clicou no botão Yes.

5. Que tipo de opção é usado para exibir só os botões Yes e No em uma caixa de diálogo de confirmação?

YES_NO_OPTION

- 6.** Se você quiser solicitar uma resposta na forma de string ao usuário, que método de **JOptionPane** deve chamar?

Para solicitar ao usuário uma resposta na forma de string, chame o método **showInputDialog()**.

- 7.** O parâmetro de mensagem de qualquer um dos métodos **show** de **JOptionPane** deve ser um string? Explique.

Não. O parâmetro de mensagem de qualquer um dos métodos **show** de **JOptionPane** pode, mas não tem que, ser um string.

- 8.** **JDialog** é um contêiner de nível superior. Verdadeiro ou falso?

Verdadeiro.

- 9.** Quais são as quatro etapas necessárias à criação e exibição de uma caixa de diálogo baseada em **JDialog**?

Estas são as quatro etapas necessárias à criação e exibição de uma caixa de diálogo baseada em **JDialog**:

- Criar um objeto **JDialog**.
- Especificar o gerenciador de leiaute, o tamanho e a política de fechamento padrão da caixa de diálogo.
- Adicionar componentes ao painel de conteúdo da caixa de diálogo.
- Exibir a caixa de diálogo chamando nela **setVisible(true)**.

- 10.** **JDialog** pode criar uma caixa de diálogo não modal?

Sim.

- 11.** Explique a diferença entre **setVisible(false)** e **dispose()** no âmbito das caixas de diálogo.

setVisible(false) remove uma caixa de diálogo da tela. **dispose()** remove a caixa de diálogo e também libera seus recursos.

- 12.** Que método de **JFileChooser** cria um selecionador de arquivos Save? E qual criaria um selecionador de arquivos para usar um título que você escolhesse?

O método **showSaveDialog()** cria um selecionador de arquivos Save. O método **showDialog()** nos permite especificar o título.

- 13.** Quais são os dois métodos que devem ser sobrepostos na implementação de um **FileFilter** para **JFileChooser**?

Os métodos **accept()** e **getDescription()** devem ser sobrepostos na implementação de um **FileFilter** para **JFileChooser**.

- 14.** O Capítulo 19 descreveu os menus. Neste capítulo, os exemplos incluíram um menu File que tinha uma entrada Exit. Altere esse código para que faça o usuário confirmar se deseja realmente encerrar o programa.

Aqui está o código que usa uma caixa de diálogo para confirmar se o usuário deseja realmente encerrar o programa:

```
// Se o usuário selecionar Exit, encerra o programa.
if (comStr.equals("Exit")) {
```

```
int response = JOptionPane.showConfirmDialog(
    null,
    "Exit Now?",
    "Terminate Program",
    JOptionPane.YES_NO_OPTION);

if(response == JOptionPane.YES_OPTION) {
    System.exit(0);
}
```

CAPÍTULO 21

1. Qualquer código que afete um componente de Swing deve ser executado na thread _____.

de despacho de evento

2. Quando usamos **javax.swing.Timer**, que evento é gerado quando o timer expira?

Um **ActionEvent** é acionado quando um **javax.swing.Timer** expira.

3. Que métodos iniciam e interrompem **javax.swing.Timer**?

Os métodos que iniciam e interrompem **javax.swing.Timer** são **start()** e **stop()**.

4. O que é **SwingWorker**?

SwingWorker cria e gerencia threads de segundo plano.

5. Que método um applet deve usar para criar sua GUI?

Um applet deve usar **invokeAndWait()** para criar sua GUI.

6. Seu programa pode pintar diretamente na superfície de um componente?

Sim, o programa pode pintar diretamente na superfície de um componente.

7. Quais são os três métodos de pintura chamados quando um componente de Swing deve gerar sua exibição?

Os três métodos de pintura a seguir são chamados quando um componente de Swing deve gerar sua exibição: **paintComponent()**, **paintBorder()** e **paintChildren()**.

8. Que métodos obtêm a largura e a altura atuais de um componente?

Para obter a largura e a altura atuais de um componente, chame **getWidth()** e **getHeight()**.

9. Refaça o applet da seção Tente isto 21-1 para que a direção da rolagem se inverta em intervalos periódicos. Por exemplo, faça o texto rolar para a esquerda durante algum tempo e depois para a direita por um determinado período, e assim por diante. A duração vai depender de você, mas a resposta mostrada aqui inverte a direção da rolagem a cada 20 segundos.

```
// Resposta do exercício 9.

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
/*
Esta HTML pode ser usada para iniciar o applet:

<object code="ScrollText" width=240 height=100>
</object>

*/

public class ScrollText extends JApplet {

    JLabel jlab;

    String msg = " Swing makes the GUI move! ";

    ActionListener scroller;

    Timer stTimer; // Este timer controla a velocidade da rolagem.
    int counter; // usa para inverter a rolagem.

    // Este valor controla quando a direção da rolagem deve mudar.
    int scrollLimit;

    // Inicializa o applet.
    public void init() {
        counter = 0;
        scrollLimit = 100;

        try {
            SwingUtilities.invokeAndWait(new Runnable () {
                public void run() {
                    guiInit();
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    // Inicia o timer quando o applet é iniciado.
    public void start() {
        stTimer.start();
    }

    // Interrompe o timer quando o applet é interrompido.
    public void stop() {
        stTimer.stop();
    }

    // Interrompe o timer quando o applet é destruído.
    public void destroy() {
        stTimer.stop();
    }
}
```

```
// Inicializa a GUI do timer.  
private void guiInit() {  
  
    // Cria o rótulo que rolará a mensagem.  
    jlab = new JLabel(msg);  
    jlab.setHorizontalAlignment(SwingConstants.CENTER);  
  
    // Cria o ouvinte de ação para o timer.  
    // Esta versão inverte a direção da rolagem  
    // a cada 20 segundos.  
    scroller = new ActionListener() {  
        public void actionPerformed(ActionEvent ae) {  
            if(counter < scrollLimit) {  
                // Rola a mensagem um caractere para a esquerda.  
                char ch = msg.charAt(0);  
                msg = msg.substring(1, msg.length());  
                msg += ch;  
            } else {  
                // Rola a mensagem um caractere para a direita.  
                char ch = msg.charAt(msg.length()-1);  
                msg = msg.substring(0, msg.length()-1);  
                msg = ch + msg;  
                if(counter == scrollLimit*2) counter = 0;  
            }  
            counter++;  
            jlab.setText(msg);  
        }  
    };  
  
    // Cria o timer.  
    stTimer = new Timer(200, scroller);  
  
    // Adiciona o rótulo ao painel de conteúdo do applet.  
    getContentPane().add(jlab);  
}  
}
```

CAPÍTULO 22

1. Crie uma instrução que exiba o substring de um string **s** composto por todos os caracteres de **s** exceto o último.

| System.out.println(s.substring(0, s.length() - 1));

2. Suponhamos que **s** fosse um string e tivéssemos a instrução a seguir:

| boolean b = s.isEmpty();

Encontre uma instrução ou um conjunto de instruções equivalente usando **length()** em vez de **isEmpty()**.

```
| boolean b = (s.length() == 0);
```

- 3.** O que aconteceria se você chamassem **charAt()** e o valor que passasse como índice estivesse fora do intervalo?

Uma **IndexOutOfBoundsException** seria lançada.

- 4.** Suponhamos que você quisesse criar um string composto por duas linhas de texto, cada um em uma linha separada. Você não pode usar uma instrução como

```
| String s = "This is the first line.  
This is the second line";
```

porque ela gerará um erro do compilador. Como pode fazê-lo?

```
| String s = "This is the first line.\nThis is the second line";
```

- 5.** Qual é o valor da expressão `2+2+"ME"`? É "22ME" ou "4ME"?

"4ME"

- 6.** Suponhamos que você quisesse verificar se uma variável de string `s` contém o string "abcdef". É suficiente chamar `s.startsWith("abc")` e `s.endsWith("def")` e então ver se as duas chamadas de método retornam verdadeiro?

Não. Strings que não sejam "abcdef" podem começar com "abc" e terminar com "def". Por exemplo, o string "abczdef" tem essas propriedades.

- 7.** No exemplo **UseTrim** deste capítulo, que demonstrou como usar o método `trim()`, a escada **if-else-if** verificou repetidamente se dois strings eram iguais usando o método `equals()`. Por que a escada não fez a verificação repetidamente usando `==`?

O operador `==` verifica a identidade do objeto, enquanto `equals()` verifica se os conteúdos dos strings são idênticos. No exemplo **UseTrim**, queríamos saber se o string `str` continha a mesma sequência de caracteres de outro string e não se `str` era o mesmo objeto de outro string.

- 8.** Explique a diferença entre `startsWith(substring, index)` e `indexOf(substring, index)`.

Os dois métodos verificam se um string é substring de outro string. No entanto, o método `startsWith()` só verifica se o substring começa no índice dado e retorna **true** ou **false**. O método `indexOf()` verifica se o substring começa no índice dado ou depois dele e retorna o índice em que o substring começa ou `-1` se o substring não fizer parte do string.

CAPÍTULO 23

- As classes usadas para encapsular valores primitivos para que eles possam ser usados onde apenas objetos são permitidos são chamadas de classes _____ encapsuladoras de tipos
- O infinito não é um número. Mais precisamente, para um **double d**, se `isInfinite(d)` for **true**, então `isNaN(d)` será **true**. Verdadeiro ou falso?

Falso.

3. **Double.MIN_VALUE** é o menor valor **double** positivo e **Integer.MIN_VALUE** é o menor valor inteiro positivo (a saber, 1). Verdadeiro ou falso?

A afirmação é verdadeira no que diz respeito a **Double.MIN_VALUE**, mas não em relação a **Integer.MIN_VALUE**. **Integer.MIN_VALUE** é o valor inteiro negativo com maior valor absoluto.

4. Determine os valores das expressões a seguir. As constantes são todas definidas na classe **Double**:

- A. MAX_VALUE / MIN_VALUE
 - B. MAX_VALUE * MAX_VALUE
 - C. POSITIVE_INFINITY / POSITIVE_INFINITY
 - D. POSITIVE_INFINITY / NEGATIVE_INFINITY
 - E. POSITIVE_INFINITY – POSITIVE_INFINITY
 - F. POSITIVE_INFINITY – NEGATIVE_INFINITY
 - G. NaN – NaN
- A. Infinito
 - B. Infinito
 - C. NaN
 - D. NaN
 - E. NaN
 - F. Infinito
 - G. NaN

5. Para qualquer caractere **ch**, a expressão

| Character.isUpperCase(Character.toUpperCase(ch))

produz **true**. Verdadeiro ou falso?

Falso. A expressão só produz **true** se **ch** for uma letra.

6. Para qualquer **double d**, a expressão **Math.ceil(Math.floor(d))** é igual à expressão **Math.floor(Math.ceil(d))**. Verdadeiro ou falso?

Falso. Para a maioria dos valores de **d**, elas não são iguais. Por exemplo, para **d = 1,5**, a primeira expressão tem valor 2,0, e a segunda, 1,0.

7. Para iniciar outro programa (outro processo pesado) em um programa Java, você deve chamar o construtor da classe **Process**. Verdadeiro ou falso?

Falso. Você deve criar um **ProcessBuilder** e chamar seu método **start()**, que retorna um objeto **Process**.

8. O método **nanoTime()** da classe **System** retorna o tempo em nanossegundos tomando como base meia-noite de primeiro de janeiro de 1970. Verdadeiro ou falso?

Falso. Ele retorna o tempo em nanossegundos desde algum ponto de partida não especificado.

9. O método **getName()** de **Class** retorna apenas o nome da classe sem o pacote ao qual ela pertence. Verdadeiro ou falso?

Falso. Ele retorna o nome completo, que inclui o prefixo do pacote.

CAPÍTULO 24

- Quais são as principais diferenças entre a classe **javax.swing.Timer** discutida no Capítulo 12 e a classe **java.util.Timer**?

As duas fornecem a mesma funcionalidade básica de fazer o código ser executado em intervalos regulares. A classe **javax.swing.Timer** aciona **ActionEvents** em intervalos regulares e os ouvintes que recebem os eventos são executados na thread de despacho de evento, o que torna esse timer útil em aplicativos Swing. A classe **java.util.Timer** é mais geral por executar um **TimerTask** arbitrário em intervalos regulares.

- Qual é a diferença entre o valor retornado por uma chamada ao método **System.currentTimeMillis()** mencionado no último capítulo e o retornado por uma chamada ao método **Calendar.getInstance().getTimeInMillis()** usado em capítulos anteriores?

Não há diferença. Os dois retornam quantos milissegundos se passaram desde meia-noite de primeiro de janeiro de 1970.

- Quando um **Formatter** vê um caractere % no string de formato, como ele sabe se esse caractere faz parte de um especificador de formato ou se é apenas um caractere comum que deve ser inserido no string de saída?

Um caractere % sempre faz parte de um especificador de formato em um string de formato. Para inserir o caractere % no string da saída, é usado o especificador de formato "%%".

- Suponhamos que você tivesse um objeto **cal** do tipo **GregorianCalendar** e quisesse exibir, com **System.out.printf()**, a data que ele representa na forma *ano:mês:dia:hora:minuto:segundo* usando a forma longa de ano, mês e dia e o formato de 24 horas. Por exemplo, se **cal** representasse o meio-dia de 11 de março de 1984, a saída seria "1984:março:11:12:00:00". Quais seriam os argumentos da chamada a **printf()**? Use índices relativos onde for possível.

| "%tY:%tB:%te:%tT", cal

- Suponhamos que **x** fosse uma variável **long** a ser exibida com o uso de **System.out.printf()** com largura de campo mínima igual a 20, sinal de adição (+) inicial se for positiva, vírgulas após cada três dígitos a partir da direita e alinhamento à esquerda. Quais seriam os argumentos da chamada a **printf()**?

| "%+, -20d", x

- Suponhamos que você tivesse uma variável **double x**, usasse um **Formatter** para criar um string contendo o valor de **x** e depois usasse um **Scanner** para recuperar esse valor a partir do string. É garantida a obtenção do mesmo valor com o qual começou? Por quê?

Não, o valor de **x** que você acabará obtendo pode ser diferente. Por exemplo, se inicialmente **x = 1,2345** e se você solicitar ao **Formatter** que formate **x** com apenas duas casas decimais de precisão, acabará obtendo **x = 1,23**.

- Qual é a saída da instrução a seguir?

```
| System.out.printf("%%(-11.2E%%", -1.23456789);  
| %(1.23E+00) %
```

8. O que aconteceria se você criasse um objeto **GregorianCalendar** que representasse uma data inválida como 32 de janeiro de 1970? Mais precisamente, o que aconteceria se executasse este segmento de código:

```
| GregorianCalendar cal = new GregorianCalendar(1970, 0, 32);  
e exibisse mês, dia e ano armazenados em cal usando chamadas a getDisplayName()?
```

Ele converteria a data para primeiro de fevereiro de 1970 e a exibiria.

CAPÍTULO 25

1. Qual é a finalidade das seguintes classes de coleção: **AbstractCollection**, **AbstractList**, **AbstractQueue**, **AbstractSequentialList**, **AbstractSet**? Isto é, por que essas classes foram incluídas no Collections Framework?

Elas fornecem implementações parciais que são usadas como pontos de partida para a criação de implementações concretas de várias classes de coleção. Na verdade, também podem ser usadas na criação de nossas próprias classes de coleção.

2. Tanto o método **poll()** quanto o método **remove()** da interface **Queue** removem e retornam o elemento do início da fila. Qual é a diferença entre eles?

O primeiro retorna **null** se **Queue** estiver vazio, e o segundo lança uma **NoSuchElementException**.

3. Um objeto **Collection** representa um grupo arbitrário de dados. Ao contrário, um **Set** representa um grupo de dados em que _____.

todos os elementos são diferentes (isto é, você não pode incluir o mesmo elemento duas vezes)

4. O que aconteceria se você usasse o método **add()** para tentar adicionar a um **Set** um elemento que já fizesse parte dele?

O método **add()** não adicionará o elemento e retornará **false**.

5. Se **m** for um **Map**, então **m.keySet().size() = m.values().size()**. Verdadeiro ou falso?

Verdadeiro.

6. Suponhamos que você tivesse três coleções de strings **c1**, **c2**, **c3** e quisesse descobrir quais strings aparecem em todas as três coleções, se isso ocorrer. Escreva um segmento de código que crie uma coleção **c4** contendo apenas os strings que aparecem nas três coleções. Ele não deve modificar as três coleções originais.

```
| Collection c4 = new ArrayList(c1);  
| c4.retainAll(c2);  
| c4.retainAll(c3);
```

7. Suponhamos que você tivesse três coleções de strings **c1**, **c2**, **c3** e quisesse descobrir quais strings aparecem em **c1**, mas não aparecem em **c2** ou **c3**, se isso ocorrer. Crie um segmento de código que gere uma coleção **c4** contendo apenas os strings desejados. Ele não deve modificar as três coleções originais.

```
Collection<String> c4 = new ArrayList<String>(c1);
c4.removeAll(c2);
c4.removeAll(c3);
```

8. O que será exibido pelo segmento de código a seguir? Explique.

```
Collection<Object> c = new ArrayList<Object>();
c.add(new Object());
c.add(new Object());
System.out.println(c.contains(new Object()));
```

O código exibirá "false". O **ArrayList** contém dois objetos, mas nenhum deles é o objeto cuja inclusão está sendo verificada.

CAPÍTULO 26

1. Os **Sockets** foram projetados para enviar e receber dados usando o protocolo HTTP. Verdadeiro ou falso?

Falso. Eles enviam e recebem dados usando o protocolo TCP.

2. Os **URLConnections** foram projetados para enviar e receber dados usando o protocolo HTTP. Verdadeiro ou falso?

Falso. Eles também podem ser usados com protocolos diferentes como o FTP para se comunicar com outro computador na Web.

3. Quais são as duas informações que um **Socket** precisa ter para se conectar com outro computador na rede?

Um nome de host e um número de porta.

4. Quando inserimos um URL usando o HTTP em um navegador Web, por que não somos solicitados a inserir um número de porta?

Há um número de porta padrão (80) que é usado quando não especificamos um.

5. Em que enviar dados usando datagramas difere, em relação às garantias de chegada, do envio de dados com um método que use o protocolo TCP?

Os datagramas não garantem sua chegada no destino. Se chegarem, não haverá garantias de que não foram danificados ou de que estejam em alguma ordem especial. Pacotes de dados enviados com o uso do TCP têm mais probabilidade de chegar sem danos e em ordem.

CAPÍTULO 27

1. Como funciona um semáforo?

Ele emite permissões para as threads acessarem um recurso compartilhado. Uma thread chama **acquire()** quando quer uma permissão e chama **release()** quando ela não é mais necessária.

2. No programa **SemDemo** deste capítulo, o que aconteceria se você adicionasse uma chamada a **sem.release()** imediatamente antes das duas chamadas a **sem.acquire()**? Explique.

O programa continuaria sendo executado, mas não haveria mais acesso controlado ao recurso compartilhado.

3. No programa **SemDemo** deste capítulo, o que aconteceria se você substituisse cada uma das duas chamadas a **sem.acquire()** por **sem.acquire(2)**? Explique.

As duas threads estarão tentando adquirir duas permissões, mas **sem** só tem uma, logo, ambas esperarão infinitamente.

4. Considere uma corrida de cavalos em que os cavalos e jockeys tivessem que ficar alinhados no portão e esperar até todos estarem prontos antes de o portão abrir e os cavalos começarem a corrida. De que ferramenta Java de sincronização essa situação lhe faz lembrar?

Um **CyclicBarrier**.

5. Suponhamos que você adicionasse a instrução **System.out.println(name);** como a última do método **run()** da classe **MyThread** no programa **BarDemo**. Nesse caso, antes de os nomes "A", "B" e "C" serem exibidos uma segunda vez, o string "Barrier reached" seria exibido. Verdadeiro ou falso? Explique.

Verdadeiro. A thread **BarAction** é executada antes de qualquer outra thread poder ser retomada.

6. Com o uso de um **Exchanger**, é possível um **String** ser trocado por um **Integer**. Verdadeiro ou falso?

Verdadeiro. Nesse caso, o parâmetro de tipo de **Exchanger** teria de ser **Object** e, após a troca, provavelmente os resultados trocados teriam de ser convertidos para o tipo correto.

7. Se uma thread tentar obter o valor de um objeto **Future** antes de ele ter sido calculado, uma exceção será lançada. Verdadeiro ou falso?

Falso. A thread solicitante do valor de **Future** esperará até o valor estar disponível.

8. Um objeto **Callable** é como um objeto **Runnable**, exceto por representar uma thread que _____.

retorna um valor

9. Um **ForkJoinPool** usa a abordagem chamada “roubo de tarefa” para executar tarefas em espera. O que é o “roubo de tarefa”?

Cada thread mantém uma fila de tarefas em espera. Se uma thread esvaziar sua fila, roubará um ou mais tarefas de outra fila.

10. Uma estratégia muito apropriada para uso com o Framework Fork/Join é a de dividir e conquistar. Como ela funciona?

Uma tarefa é dividida recursivamente em duas ou mais subtarefas até todas as subtarefas serem suficientemente pequenas para ser executadas em sequência por uma thread.

Índice

SÍMBOLOS

& (E comercial)
operador && (AND de curto-círcuito), 59, 61, 67
operador &= (atribuição com AND lógico), 64
operador AND bitwise, 186-188
operador AND lógico, 59, 67
<> (colchetes angulares)
operador < (menor que), 29, 59, 60, 67
operador << (bitwise de deslocamento para a esquerda), 191
operador <= (menor ou igual a), 29, 59, 60, 67
operador > (maior que), 29, 59, 60, 67
operador >= (maior ou igual a), 29, 59, 60, 67
operador >> (bitwise de deslocamento para a direita), 191
operador >>> (bitwise de deslocamento para a direita sem sinal), 191
operador losango, 530
parâmetros de tipo dentro, 499
* (asterisco)
operador *= (multiplicação e atribuição), 64
operador de multiplicação, 24, 57, 67
@ (sinal de arroba), em nomes de anotações, 489
\ (barra invertida), em sequências de escape, 51
^ (circunflexo)
operador XOR bitwise, 186, 187, 189
operador XOR lógico, 59, 67

operador ^= (XOR com atribuição), 64, 193
, (vírgula), em listas separadas por vírgulas, 30
{ } (chaves)
delimitando blocos de código, 32, 33
delimitando definições de classe, 20
delimitando o corpo do método, 21
= (sinal de igualdade)
operador == (igual a), 29, 59, 60, 181, 813
operador de atribuição, 63, 67
! (ponto de exclamação)
operador ! (NOT lógico), 59, 67
operador != (diferente de), 29, 59, 60, 67
/ (barra comum)
/* */ delimitando comentários de várias linhas, 20
/** */ delimitando comentários de documentação, 1041
// em comentários de linha única, 20
operador /= (divisão e atribuição), 64
operador de divisão, 24, 57, 67
- (sinal de subtração)
menos unário, 67
operador – (decremento), 32, 57, 58, 67
operador -= (subtração e atribuição), 64
operador de subtração, 24, 57, 67
() (parênteses)
após nomes de métodos, 123
em expressões, 71
. (ponto)
separando nomes de pacotes, 323
... (três pontos), especificando varargs, 242
| (símbolo pipe)
operador OR bitwise, 186-188
operador OR lógico, 59, 67
operador |= (OR lógico com atribuição), 64
operador || (OR de curto-círcuito), 59, 61, 67
% (sinal de porcentagem)
operador %= (módulo e atribuição), 64
especificadores %n e %%, 880
operador de módulo, 57, 67
+ (sinal de adição)

A

abs(), 218, 508, 839
Abstract Window Toolkit (AWT), 539, 610
 herança de funcionalidade por Applet, 553
 importando, 540
 Swing e, 611
 accept(), 412, 756, 757
 aceleradores, adicionando a itens de menu, 710, 722
 acesso padrão, 325, 326
 actionPerformed(), 625, 627, 628, 630, 635, 641, 642, 645, 663, 671, 708, 755, 773
 add(), 618-620, 628
 Container, 702, 703
 JMenu, 703
 JMenuBar, 702
 addActionListener(), 623, 628, 633, 645, 773
 addSeparator(), 703
 adicionando/removendo itens de menu dinamicamente, 713-719
 alternativa aos iteradores com o uso de for-each, 946
 ambiente multitarefa baseado em processos, 429
 ambientes de desenvolvimento integrado (IDEs), 17
 anotações (metadados), 489
 criando, 489
 internas, 490
 aparência adaptável, 610
 append(), 825
 applets, 11-12, 539-553
 applet de banner simples, 546-550
 applet Swing simples, 780
 aspecto-chave da arquitetura, 544
 baseados em AWT ou Swing, 539
 criando applets Swing, 779
 esboço de applet completo, 543
 executando, 541
 inicialização e encerramento, 544
 passando parâmetros para, 551
 recursos adicionais, 553
 rolando texto em applet Swing, 782-787
 solicitando atualização, 545
 usando a janela de status, 640

appletviewer, 541
APPROVE_OPTION, JFileChooser, 752, 753, 755
 argumentos, 123, 129, 130
 em quantidade variável (varargs), 241-247
 implícitos, 146
 passando para métodos, 209-212
 argumentos curingas, 508-510
 limitados, 510-512
 argumentos de linha de comando, 184-186
 ArithmeticException, 346, 347, 351, 353, 363
 arquitetura de modelo separável, 611
 arquitetura modelo-delegação, 611, 696
 arquitetura modelo-exibição-controlador (MVC), 611
 arquivos .class, 119, 323
 arquivos .java, 755
 arquivos de acesso aleatório, 397-399
 ArrayIndexOutOfBoundsException, 343, 346, 351, 363
 arrays, 21, 152-178
 arrays multidimensionais irregulares, 159-160
 atribuindo referências de array, 163-165
 bidimensionais, 158-160
 classificando, 156-158
 dados de JTable, 693, 694
 de strings, 181
 de três ou mais dimensões, 161
 definição, 152
 demonstração de classe de pilha simples, 167-173
 encapsulamento como membro privado da classe, 204-208
 genéricos, 533
 inicializando arrays multidimensionais, 161-163
 iterando por arrays multidimensionais, 176
 método genérico para encontrar o valor mínimo de, 855-857
 pesquisando, usando o laço for melhorado, 177
 sintaxe alternativa de declaração, 163
 unidimensionais, 153-156
 usando laço for de estilo for-each com, 172-176
 usando o membro length, 164-167
 árvores, 914
 aspas, escape de, 51
 AssertionError, 555
 assinantes, 599-600
 assinatura, 219
 ativando/desativando recursos da GUI
 controles, 657, 658
 ícone em item de menu, 712
 itens de menu, 704
 atribuições
 autoboxing em, 500
 bitwise abreviadas, 193
 conversão de tipo em, 64
 declarando referência a classe genérica e criando instância, 503
 operador new em, 142
 variáveis de referência e, 121
autoboxing/unboxing, 482-488
 autoboxing e métodos, 485
 em atribuições, 500
 encapsuladores de tipo e o boxing, 482-484, 837
 fundamentos do autoboxing, 484
 ocorrendo em expressões, 486-488
 retrinando o uso de encapsuladores de tipo, 488
AWT. Consulte Abstract Windows Toolkit

B

benefícios da reutilização de código fornecidos pela herança, 584-585
 biblioteca de APIs Java, 331
 binarySearch(), 958-959, 961-962
 bits, 6-7
 classe de exibição de bits de uso geral, 193-196
 bloco try/catch de tratamento de exceções, 343, 384
 blocos de código, 32
 blocos estáticos, 233
 classes aninhadas em, 239
 escopo de variáveis em, 55

- blocos finally, 357-359, 385-387, 389, 390, 392
 blocos try/catch/finally, 357-359
 bloqueios, 1020-1023
 classe ReentrantLock, 1020, 1023
 classe ReentrantReadWriteLock, 1023
 interface Lock, 1020
 interface ReadWriteLock, 1023
 Shared.count, 1021
borda
 área de desenho e, 789
 definindo para um painel, 793
BorderLayout, 615, 618, 620, 651
botões, 653-658
 botões de ação em Swing, usando JButton, 625-628, 640, 641, 655-658
 botões de alternância, usando JToggleButton, 658-660
 botões de rádio, usando JRadioButton, 662-664
 itens de menu como, 701
 tratando eventos de ação em, 654
 tratando eventos de item em, 654
 botões de alternância, JToggleButton, 658-660
 botões de rádio
 controles JRadioButton, 662-664
 JRadioButtonMenuItem, 701, 719-722
boxing, encapsuladores de tipo e, 482-483-484
buffer duplo, 550
busca binária, 959, 961-962
busca sequencial, 913, 961-962
bytecode, 12-13
bytes, 6-7
- C**
- C/C++**
 Java e, 10-11
 nomes e escopo de variáveis, 56
 caixa de diálogo ERROR_MESSAGE, 730, 735, 739, 741
 caixa de diálogo INFORMATION_MESSAGE, 730, 735, 739, 741
 caixa de diálogo OK_CANCEL_OPTION, 735, 741
 caixa de diálogo PLAIN_MESSAGE, 730, 735, 739, 741
 caixa de diálogo QUESTION_MESSAGE, 730, 735, 739, 741
 caixa de diálogo WARNING_MESSAGE, 730, 735, 739, 741
 caixa de diálogo YES_NO_CANCEL_OPTION, 735, 741
 caixa de diálogo YES_NO_OPTION, 735, 741
 caixas de combinação, criando com JComboBox, 686-689
 caixas de diálogo, 725-765
 criando caixa de diálogo não modal, 750
 filtro de arquivos, usando com JFileChooser, 755-761
 JDialog, 746-750
 JOptionPane, 726
 opções adicionais suportadas por JFileChooser, 761
 seleção de arquivos com JFileChooser, 751-755
 showConfirmDialog(), 727, 732-736
 showInputDialog(), 736-741
 showMessageDialog(), 728-731
 showOptionDialog(), 741-745
 suporte de classes de Swing, 726
 caixas de diálogo de confirmação, 726, 732-736
 caixas de diálogo de entrada, 726, 736-741
 caixas de diálogo de mensagem, 726, 728-731
 caixas de diálogo de opções, 726, 741-745
 caixas de diálogo modais, 726, 728, 746
 caixas de diálogo não modais, 727-746
 caixas de seleção
 controles JCheckBox, 660-662
 JCheckBoxMenuItem, 701, 719-721
 campos de texto, JTextField, 633-638, 665-668
 CANCEL_OPTION, JFileChooser, 752
 caracteres, 46
 strings versus, 52
 caracteres curingas em expressões regulares, 1053-1055
 ceil(), 839
 chamada por referência, 210, 211
 chamada por valor, 210, 211
 changeBorderSize(), 794
 charValue, 834
class CountDownLatch, 995-999
classe AbstractButton, 625, 653, 657
 extensão por itens de menu, 701, 704
 setDisplayedMnemonicIndex(), 709
 setIcon(), 712
 setMnemonic(), 709
classe AbstractMap, 951
classe ActionEvent, 625, 627, 628, 635, 654, 663, 665
classe Applet, 540, 543, 779
 métodos adicionais, 553
classe ArrayDeque, 939-941
classe ArrayList, 927-930
classe Arrays, 962
classe AWTEvent, 622-624
classe Boolean, 414, 483, 837
classe Border, 793
classe BorderFactory, 793
classe BufferedReader, 401-403, 407, 420
classe Byte, 414, 483, 830-834
classe Calendar, 629, 631, 863, 868-872
classe Character, 414, 483, 834-837
classe Class, 499, 851
classe Color, 793
classe Component, 543, 545, 613, 702, 746
 método add(), 618
 método paint(), 787
classe Console, 399
classe Container, 613, 746
 add(), 702, 703
 getContentPane(), 619
 remove(), 702
classe CyclicBarrier, 1000-1002
classe DataInputStream, 393-395
classe DataOutputStream, 393-395
classe Date, 863, 867
classe DefaultMutableTreeNode, 690
classe Dialog, 746
classe Dictionary, 963

- classe Dimension, 686
classe Double, 414, 482-483, 488, 829-830
classe Enum, 474-477, 853
classe Error, 342, 359, 362
classe EventObject, 621-624, 654
classe Exception, 342, 350, 362 criando subclasses, 364-366
classe Exchanger, 1002-1004
classe File, 408, 753 método listFiles(), 412 método toPath(), 413 métodos para obtenção das propriedades de File, 406, 409 métodos utilitários, 412-414 obtendo uma listagem de diretório, 410
classe FileChooser, 756, 757
classe FileFilter, 412, 756, 757
classe FileInputStream, 383, 384, 390, 391, 394
classe FileOutputStream, 383, 387, 390, 393
classe FileReader, 405, 406
classe FileWriter, 405
classe Float, 414, 482-483, 829-830
classe FlowLayout, 615, 627
classe Formatter, 863, 873-891 construtores, 873 especificador de precisão, 881 especificadores %n e %%, 880 especificadores de formato, 876 especificando largura de campo mínima, 880 fechando um Formatter, 887 flag #, 884 flag de vírgula, 884 formatação e o método printf(), 890 formatando data e hora, 878-880 formatando números, 877 formatando para regiões diferentes, 886 formatando strings e caracteres, 877 formatando tabela de cidades e populações, 888-890 justificando a saída, 883 métodos, 875 opção de maiúscula, 884-885 usando flags de formato, 882 usando índice de argumento, 884-885
- classe Graphics, 540, 788
classe GregorianCalendar, 863, 868-872
classe hashMap, 952
classe HashSet, 934-936
classe Hashtable, 963
classe HttpURLConnection, 982-984
classe ImageIcon, 650-653, 712
classe InetAddress, 969-971
classe InheritableLocalThread, 854
classe InputEvent, 710
classe InputStream, 379-381, 393, 401 método read(), 381 métodos, 380
classe InputStreamReader, 401, 406
classe Insets, 789
classe Integer, 414, 482-484, 488, 830-834
classe ItemEvent, 654, 655, 660, 687
classe java.util.EventObject, 621
classe KeyEvent, 709, 710
classe KeyStroke, 710
classe LinkedHashMap, 955
classe LinkedHashSet, 938
classe LinkedList, 930-933
classe List, 778
classe Locale, 823, 864-867
classe Long, 414, 482-483, 830-834
classe Matcher, 1049, 1050 método replaceAll(), 1055
classe Math, 335, 838-841 constantes E e PI, 840 método abs(), 218, 508 método sqrt(), 45, 46 métodos estáticos, 233
classe MouseEvent, 722
classe MouseMotionAdapter, 624
classe MouseMotionListener, 624
classe Number, 482-483, 504, 829 métodos, 483 tipos limitados e, 505
classe Object, 294, 851-853 métodos, 295, 852 referências de tipo Object, 497 wait(), notify() e notifyAll(), 451
classe Object, 907, 908
classe Observable, 863, 901-904 métodos, 902
classe Observer, 78, 901-904
classe OutputStream, 380 métodos, 381
classe OutputStreamWriter, 405
classe Pattern, 1049
classe Phaser, 1005-1010 demonstrando método onAdvance(), 1009-1012
classe PrintStream, 379, 382, 850, 890, 891
classe PrintWriter, 404, 890, 891
classe PriorityQueue, 941
classe Process, 842-844
classe ProcessBuilder, 842-844
classe Properties, 964
classe Random, 793, 863, 900
classe Reader, 378, 399, 405, 406 métodos definidos por, 400
classe RecursiveAction, 1026, 1029
classe RecursiveTask<V>, 1026, 1034-1036
classe ReentrantLock, 1020
classe ReentrantReadWriteLock, 1023
classe Runtime, 844-846
classe RuntimeException, 342, 359, 360, 362
classe Scanner, 422, 863, 892-899 construtores, 892 exemplos, 894-899 métodos hasNext, 894 métodos next, 894 outros recursos, 899
classe Segment, 857-858
classe Semaphore, 994-997
classe Short, 414, 482-483, 830-834
classe Socket, 971-975
classe Stack, 963
classe StringBuffer, 182, 800, 825, 857-858
classe StringBuilder, 800, 825, 857-858
classe SwingUtilities, 767, 772
classe SwingWorker, 777-779
classe Thread, 429, 430, 854, 1012 estendendo, 435-437, 440 getName() e setName(), 433 método currentThread(), 461 método getPriority(), 444 método isAlive(), 440, 441 método join(), 441, 443 método notify(), 451-456

- método `notifyAll()`, 451
 método `setPriority()`, 444, 450
 método `sleep()`, 432
 método `start()`, 431, 547
 método `wait()`, 451-456
 classe `ThreadGroup`, 854
 classe `ThreadLocal`, 854
 classe `Throwable`, 342, 353, 355-357, 362, 364, 392, 534
 classe `Timer (java.util)`, 863, 905-907
 classe `Timer`, 773-777
 usando em programa de rolagem de texto em applet, 782-787
 classe `TimerTask`, 863, 905-907
 classe `TreeMap`, 953-955
 classe `TreePath`, 690
 classe `TreeSet`, 936-938
 classe `URL`, 975-977
 classe `URLConnection`, 977-982
 classe `Vector`, 963
 classe `Window`, 746
 classe `Writer`, 378, 399, 400, 405
 métodos definidos por, 401
 classes, 15-16, 116-121. *Consulte também* herança
 abstratas, 287-291
 adicionando construtor a, 141
 adicionando método a, 123-125
 adicionando método parametrizado a, 130-132
 aninhadas e internas, 237-241
 coesão de classes e padrão Expert, 571-573
 construtores, 138-142
 declaração de herança de uma superclasse, 256
 definição, 20, 118-120
 diagramas de classes UML, 581
 elegantes, 571
 evitando duplicação em, 573-575
 finais, 292
 forma geral, 117
 implementando interfaces, 300-304
 implementando interfaces genéricas, 520-522
 implementando várias interfaces, 306-308
 interface completa para, 575
 internas, 38
 movendo para outro pacote, 332-334
- nomeando, 564
 projetando visando mudanças, 576-579
 classes abstratas, 287-292, 304
 classes adaptadoras, 622-624
 classes aninhadas e internas, 237-241
 classes de exceção, genéricas, impossibilidade de criar, 534
 classes encapsuladoras de tipos primitivos, 212, 414-416, 829-838
 e *autoboxing/unboxing*, 482-488, 837
 classes encapsuladoras numéricas, 482-483, 829-838
 convertendo strings numéricos, 414-416
 Number como superclasse, 504
 tipos limitados e, 505
 classes internas, 237-241
 classes internas anônimas, 240, 619
 usando para tratar eventos, 645
 classificando
 método de classificação de bolha, 156-158
 método de classificação rápida, 235-237
 cláusula `else`, instruções `if`, 77, 81
 cláusula `implements`, 300, 522
 cláusula `throws`, 359, 362, 363
 cláusulas `catch`, 345, 346, 363, 364
 capturando exceções da subclasse, 350
 cláusula `catch` em blocos `try/catch`, 343, 344
 em blocos `try` aninhados, 351-353
 exceções relançadas, 354
 instrução `try-with-resources`, 390
 multi-catch, 363
 usando várias, 349
`clone()`, 295, 851
`CloneNotSupportedException`, 851
`close()`, 384, 385, 388-392
 código-fonte, 7-8
 código-objeto, 7-8
 coerções, 65-67, 70
 de referências `Object`, 497
 instâncias de uma classe genérica, 513
- coleta de lixo, 142-148
 demonstração de, 144-148
 método `finalize()`, 143
 Collections Framework, 862, 911-966
 acessando uma coleção via um iterador, 942-946
 algoritmos de coleção, 958-962
 alternativa aos iteradores com o uso de `for-each`, 946
 árvores, 914
 classe `ArrayDeque`, 939-941
 classe `ArrayList`, 927-930
 classe `Arrays`, 962
 classe `Collections`, 958-959
 classe `HashMap`, 952
 classe `HashSet`, 934-936
 classe `LinkedHashMap`, 955
 classe `LinkedList`, 930-933
 classe `PriorityQueue`, 941
 classe `TreeMap`, 953-955
 classe `TreeSet`, 936-938
 classes de coleção, 926
 classes e interfaces legadas, 963
 comparadores, 956-959
 interface `Collection`, 918-920
 interface `Deque`, 924-926
 interface `List`, 920, 921
 interface `Map`, 947
 interface `Map.Entry`, 949
 interface `NavigableMap`, 948, 950-951
 interface `NavigableSet`, 922, 923
 interface `Queue`, 922, 924
 interface `Set`, 920
 interface `SortedMap`, 947
 interface `SortedSet`, 920-922
 interfaces de coleção, 917
 interfaces de mapa, 946
 listas encadeadas, 913
 mapas, 946
 pilhas e filas, 912
 selecionando uma estrutura de dados, 915
 tabelas hash, 915
 visão geral das estruturas de dados, 912
 visão geral do Collections Framework, 916
 visões de coleção versus arrays obtidos a partir de uma coleção, 933

- comando de ação, 625, 628, 635, 640, 654
 definindo para JButton e JTextField, 638
 definindo para JTextField, 665
 obtendo/definindo para JButton, 655
 obtendo/definindo para JRadioButton, 663
 para menus, 708
- comentários, 20, 568. *Consulte também javadoc*
 comentários de linha única, 20
 comentários de várias linhas, 20
 comparadores, 956-959
compareTo(), 179-180, 416, 474-477, 815-817, 830-831
compareToIgnoreCase(), 815-817
 compatibilidade de tipos de dados, 64
 compilador just-in-time (JIT) para bytecode, 13-14
 compiladores, 7-8
 compilador JIT para bytecode, 13-14
 javac, 17
 verificando tipos de código Java, 42
 compilando programa com javac, 19
 componentes de hardware dos computadores, 4
 componentes leves, 610
 componentes pesados, 610
 comportamento, semelhante, 587-589
 concatenando strings, 181, 803
 condição de corrida, 457
 conjunto de caracteres ASCII, 46
 constantes, 49
 em interfaces, 314
 enumeração, 468
 finais, 293
 finais estáticas, 337
 interface WindowConstants, 618
 constantes de enumeração, 468, 470
 constantes de enumeração autotipadas, 468
 construtores, 138-142
 adicionando a uma classe, 140
 chamando construtores sobre-carregados com this(), 556
 e herança, 259-261
- encapsuladores de tipos numéricos, 483
 genéricos, 304, 515
 para enumerações, 472, 473
 parametrizados, 139
 sobrecarregando, 219-321
 usando super() para chamar construtores da superclasse, 261-265
- contêineres, 613
 painéis de contêiner de nível superior, 613
 contexto gráfico, 540, 546, 788
 controlador, 611
 controles, 649. *Consulte também controles de Swing*
 e menus como principais recursos da GUI, 700
 controles de árvore, usando JTree, 689-693
 controles de botão de ação. *Consulte botões*
 controles de lista, criando com JList, 682-686
 controles de Swing, 649-699
 árvores, usando JTree, 689-693
 botões, usando JButton, 655-658
 botões de alternância, usando JToggleButton, 658-660
 botões de rádio, usando JRadioButton, 662-664
 botões e eventos, 653-655
 caixa de combinação, criando com JComboBox, 686-689
 caixas de seleção, usando JCheckBox, 660-662
 campos de texto, usando JTextField, 665-668
 classes de componentes, 649
 lista, criando com JList, 682-686
 modelos usados por, 696
 outros tipos oferecidos por Swing, 696
 rolando componentes com JScrollPane, 675-677
 rolando um JPanel usando JScrollPane, 677-682
 rótulos, usando JLabel e ImageIcon, 650-653
 tabelas, usando JTable, 693-695
 usando em utilitário de comparação de arquivos, 668-675
- convenções de nomenclatura, 563
 conversão de galões para litros, exemplo de programa, 27
 melhorada, 35
 conversões, tipo de dado
 em atribuições, 64
 em expressões, 69-71
 usando uma coerção, 65-67
 conversões ampliadoras, 65
 conversões de tipo automáticas, 64
 conversões redutoras, 66
copy(), JTextField, 665
copySign(), 841
 CPU (Unidade Central de Processamento), 4
currentTimeMillis(), 847-849
cut(), JTextField, 665, 668

D

- dados binários, lendo e gravando, 392-395
 datagramas, 984-989
 classe DatagramPacket, 985, 989
 classe DatagramSocket, 985, 989
 I/O de arquivo, 377
 arquivos de acesso aleatório, 397-399
 fechando arquivos automaticamente, 389
 gravando em um arquivo, 387-389
 lendo/gravando arquivos usando fluxos de bytes, 383-387
 lendo/gravando dados binários, 392-395
 usando fluxos de caracteres, 405-407
 I/O de console, 21, 377
 entrada usando fluxos de caracteres, 400-403
 gravando saídas no console, 382
 lendo entradas no console, 381
 saída usando fluxos de caracteres, 404
 System.out, 21
 I/O de console baseado em texto, 377
 deadlock, 456

DEFAULT_OPTION, caixa de diálogo, 741
 delegação, 587-588, 591-593
 delegação de UI, 611
`delete()`, 825
 despacho dinâmico de métodos, 281-283
`destroy()`, 543, 544, 779, 784, 842, 844
 diagramas UML, 581-583
 dicas de ferramentas
 adicionando a controles, 713
 adicionando a itens de menu, 712
 diferenciação entre maiúsculas e minúsculas em Java, 22, 37
`digit()`, 836
 diretórios
 dando suporte a uma hierarquia de pacotes, 323
 obtendo a listagem de diretórios, 410
`dispose()`, 746, 751
 dispositivos de entrada/saída (I/O), 5-6
 documentação. *Consulte também javadoc*
 externa, 568-571
 interna, 567
`doInBackground()`, 777, 778
`done()`, 778
`doubleValue()`, 483, 484, 504
`drawLine()`, 787, 788
`drawOval()`, 788
`drawRect()`, 787, 788
`drawString()`, 540, 545, 788
 duplicação, evitando, 573-575

E

encapsulamento, 15-16, 578
 endereços
 Internet, 968
 memória do computador, 5-6
`endsWith()`, 757, 814
 entrada de console em buffer de linha, 77
 entrada do teclado, 76, 422
 enumeração TimeUnit, 1018-1020
 enumerações, 82, 467-483
 como tipos de classe, 471
 construtores, métodos e variáveis de instância, 472-474
 em semáforo controlado por computador, 477-483

herança da superclasse Enum, 474-477
 métodos `values()` e `valueOf()`, 471
 variáveis finais e, 474
`equals()`, 178-179, 181, 295, 416, 812, 830-831
 versus `==`, 813
`equalsIgnoreCase()`, 812
`erasure`, 500, 531
 erros de ambiguidade de, 531
 erro de divisão por zero, 61, 347
 ERROR_OPTION, JFileChooser, 752
 erros
 erros comuns em programas, 342
 erros de tempo de execução provenientes de tipos brutos, 528
 leitura em arquivo, 384
 tratando erros de sintaxe, 22
 erros de ambiguidade com genéricos, 531
 escopo das variáveis, 54-56
 escopo global, 54
 escopo local, 54
 escopos aninhados, 54
 espaço de nome, 321
 estendendo uma classe, 266-269
 estendendo uma interface, 316
 estruturas de dados, 911-916.
Consulte também Collections Framework
 árvores, 914
 listas encadeadas, 913
 pilhas e filas, 912
 selecionando uma estrutura de dados, 915
 tabelas hash, 915
 visão geral das estruturas de dados, 912
 eventos, 621
 classes de eventos e interfaces de ouvintes, 622-624
 controles de árvore, JTree, 689
 controles de botão em Swing, 625-628, 653-655
 controles de lista, JList, 682
 fontes de eventos, 621
 programas Swing, 618
 eventos de pressionamento de tecla, ouvinte de, 622-624
 exceções encadeadas, 362
 exceções não verificadas, 361, 363
 exceções suprimidas, 392
 exceções verificadas, 361, 363
 Exclusive OR com operador de atribuição (`^=`), 64, 193
`execute()`, 777
 executores, 1012-1015
 ForkJoinPool, 1013
 interface Executor, 1013
 interface ExecutorService, 1013
 ScheduledExecutorService, 1013
 ScheduledThreadPoolExecutor, 1013
 ThreadPoolExecutor, 1013
 exibição, 611
`exit()`, 708
`exitValue()`, 842
`exp()`, 841
 expressões, 69
 conversões de tipo em, 69-71
 espacamento e parênteses, 71
 expressões booleanas, 28, 30
 expressões regulares, 1049-1056
 caractere curinga e quantificadores, 1053-1055
 classe Matcher, 1050
 classe Pattern, 1049
 classe String e, 1056
 classes de caracteres, 1055
 demonstrando a comparação de padrões, 1051-1053
 sintaxe, 1050
 usando `replaceAll()` de Matcher, 1055

F

fatorial de um número, método recursivo retornando, 227-229
 ferramentas de linha de comando (JDK), 17
 filas, 912
 classe PriorityQueue, 941
 fila pura, implementada com, o uso da delegação, 591-593
`FileNotFoundException`, 384, 387, 406, 848
`fillOval()`, 788
`fillRect()`, 788
 filtro de arquivos, usando com JFileChooser, 755-761
`finalize()`, 143, 295
`findInLine()`, 899
`findWithinHorizon()`, 899

- floor(), 839
 fluxos, 377
 classes de fluxos de bytes, 377
 classes de fluxos de caracteres, 378
 fluxos de bytes e de caracteres, 377
 fluxos predefinidos, 379
 fluxos de bytes, 377, 380
 classes de fluxos de bytes, 377
 lendo e gravando arquivos com, 383-387
 fluxos de caracteres, 377, 400
 classes de fluxos de caracteres, 378
 usando entrada do console, 394
 usando I/O de arquivo, 405-407
 forDigit(), 836, 837
 formatação de data e hora, 878-880
 FORTRAN, 14-15
 Framework Fork/Join, 450, 1024-1037
 classe ForkJoinTask<V>, 1025
 classe RecursiveAction, 1026, 1029
 classes principais, 1025
 estratégia de dividir e conquistar, 1028
 executando tarefa assincronamente, 1036
 exemplo usando RecursiveTask<V>, 1034-1036
 ForkJoinPool, 1013, 1027
 impacto do nível de paralelismo, 1031-1034
 primeiro exemplo simples, 1029-1031
 RecursiveTask<V>, 1026
 freeMemory(), 845
- G**
- genéricos, 295, 496-538
 classe com dois parâmetros de tipo, 501-503
 classe SwingWorker, 777-779
 coerções de instâncias de classe genérica, 513
 construtores, 515
 criando pilha genérica simples, 522-526
 curingas limitados, 510-512
- erasure de informações de tipo, 531
 erros de ambiguidade, 531
 exemplo simples, 497-501
 forma geral de uma classe, 503
 hierarquias de classes, 516-519
 inferência de tipo com operador losango, 529
 JComboBox, 686-689
 JList, 682-686
 métodos, 513-515
 restrições a, 532-534
 segurança de tipos com, 497
 suporte de outras linguagens, 503
 tipos brutos para código legado, 526-529
 tipos diferindo de acordo com os argumentos de tipo, 501
 tipos limitados, 504-507
 trabalhando apenas com objetos, 501
 usando argumentos curingas, 507-510
 gerando componentes, 787-794
 calculando a área de desenho, 789
 contexto gráfico, 788
 exemplo, PaintPanel, 789-794
 solicitando geração do componente, 789
 gerenciadores de leiaute, 614, 620, 627, 651
 e a definição de tamanho de componentes, 686
 gerenciamento automático de recursos, 389
 get(), 778
 getActionCommand(), 623, 628, 654, 655, 663
 getCause(), 362
 getChars(), 809-810
 getClass(), 295, 499
 getContentPane(), 619
 getDescription(), 756
 getFreeSpace(), 412
 getGraphics(), 546
 getHeight(), 789
 getInsets(), 789
 getInstance(), 631
 getItem(), 654, 658, 660
 getKeyStroke(), 710
 getLastPathComponent(), 690
 getMenuComponentCount(), 703
 getMenuComponents(), 703
- getMenuCount(), 702
 getName(), 433, 436, 499, 753
 getParameter(), 551
 getPath(), 690
 getRuntime(), 844
 getSelectedFile(), 753
 getSelectedFiles(), 762
 getSelectedIndex(), 683
 getSelectedItem, 687
 getSelectedText(), 665, 668
 getSelectedValue(), 683
 getWidth(), 789
 GridLayout, 678, 686
 guiInit(), 784
- H**
- hashCode(), 295, 851
 HeadlessException, 728, 730, 735, 736
 herança, 253-297, 581-593
 acessando membros da superclasse com super, 265
 acesso a membros e, 256-259
 chamando construtores da superclasse usando super(), 261-265
 classe Object, superclasse de todas as outras classes, 294
 classes abstratas, 287-291
 construtores e, 259-261
 criando hierarquia de classes com vários níveis, 269-272
 custos de, 598-599
 enumerações e, 474
 estendendo uma classe, 266-269
 execução de construtores em hierarquia de classes, 272
 impedir usando a palavra-chave final, 292-294
 importância dos métodos sobrepostos, 283
 interfaces, 316
 referências da superclasse e objetos da subclasse, 273-278
 sobreposição de métodos, 278-281
 sobreposição de métodos, aplicando, 283-287
 suporte dos métodos sobrepostos ao polimorfismo, 281-283
 hierarquia de classes, criando, 269-278
 HTML, usando em rótulos de Swing, 652

- HTTP (Hypertext Transfer Protocol), 968
hypot(), 839
-
- I**
- identificadores, 37
 IDEs. Consulte ambientes de desenvolvimento integrado
 imagens
 adicionando a itens de menu, 712, 721
 formatos de arquivo de imagem, 650
 passando ícones para showOptionDialog(), 742
 independência da plataforma, 10-11
 inferência de tipo com operador losango, 529
 inicialização de variáveis, 53
 inicialização dinâmica, 53
init(), 543, 544, 779, 780, 784
initCause(), 362
 I/O (input/output ou entrada/saída), 376-427
 arquivos de acesso aleatório, 397-399
 caracteres de entrada do teclado, 76
 classe Console, 399
 classe File, 408-410, 412-414
 classe Scanner, 422
 classes de fluxos de bytes, 377
 classes de fluxos de caracteres, 378
 criando sistema de ajuda baseado em disco, 416-422
 fechando arquivo automaticamente, 389-392
 FilenameFilter, 411
 fluxos, 387
 fluxos de bytes e fluxos de caracteres, 377
 fluxos predefinidos, 379
 I/O de arquivo usando fluxos de caracteres, 405-407
java.io.IOException, 77
 lendo/gravando arquivos usando fluxos de bytes, 383-389
 lendo/gravando dados binários, 392-395
 redirecionando fluxos de I/O padrão, 849-851
- saída do console, System.out, 21
 usando encapsuladores de tipo, 414-416
 usando fluxos de bytes, 380-383
 usando fluxos de caracteres, 400-405
 utilitário de comparação de arquivos, 395-397
InputMismatchException, 422-423
insert(), 703, 824
 instâncias, 15-16
 instrução continue, 106-108
 instrução de especificação de recursos, 390
 instrução goto, break como forma de, 102-106
 instrução package, 322, 331
 instrução padrão, 82, 83, 86, 99
 dando ao membro da anotação um valor padrão, 490
 instrução throw, 353, 354
 instrução try-with-resources, 363, 389-392
 argumentos de tipo passados para parâmetros de tipo, 500, 501
 classe genérica com dois parâmetros de tipo, 502
 instruções
 reco de, 34
 terminando com ponto e vírgula (;), 33
 instruções aninhadas
 instrução if, 79
 instrução switch, 84
 instruções break, 82-84, 100-106, 108
 saindo de laços com, 100-102
 usando como forma de goto, 102-106
 instruções case, 82-84
 constantes enumeráveis em, 469
 instruções de controle, 28-32, 75-115
 escada is-else-if, 80
 instrução break, 100-106
 instrução continue, 106-108
 instrução if, 28-30, 77-79
 instrução switch, 81-85
 laço do-while, 95-97
 laço for, 30-32, 88-93
 laço while, 93-95
 laços aninhados, 111
 instruções de máquina ou código de máquina, 5-6
 instruções if, 28-30, 35, 77-81
 anhadas, 79
 cláusula else, 77
 escada if-else-if, 80, 87
 switch baseada em string versus if/else, 184
 instruções import, 330, 540
 importação estática, 335-337
 instruções return, 125-128
 instruções switch, 81-87, 98, 99, 108
 anhadas, 84
 enumeração usada para controlar, 469, 478
 objeto Integer controlando, 487
 usando strings para controlar, 183
 variável response controlando, em caixa de diálogo de confirmação, 734
 inteiros, 26
 tipos de dados, 43-45, 50
 interface ActionListener, 622-624, 627, 639, 654, 663, 708, 758, 773
 classe interna anônima implementando, 645
 método actionPerformed(), 625
 interface Annotation, 489
 interface Appendable, 857
 interface AutoCloseable, 390, 859
 interface CharSequence, 800, 824, 857-858, 1049
 interface Cloneable, 851
 interface Closeable, 390
 interface Comparable, 800, 830-831, 855, 956
 interface Comparator, 956
 interface DataInput, 397
 interface DataOutput, 393, 397
 interface Enumerator, 963
 interface FileNameFilter, 411, 412
 interface Future, 1015-1018
 interface ItemListener, 654, 658, 660
 interface Iterable, 857-858
 interface List, 920, 921
 interface membro, 317
 interface MouseListener, 722
 interface MutableTreeNode, 690
 interface NavigableMap, 948, 950-951

interface NavigableSet, 922, 923
interface Path, 413
interface Queue, 922, 924
interface Readable, 857-858
interface ReadWriteLock, 1023
interface Runnable, 429-432,
440, 477, 577, 619, 767, 771,
780, 854
interface Serializable, 800
interface Set, 920
interface SortedMap, 947
interface SortedSet, 920-922
interface TreeNode, 690
interface WindowConstants, 618
interfaces, 298-320
 anhinadas, 317
 classe implementando várias
 interfaces, 306-308
 constantes em, 314
 criando, 299, 308-314
 declarando, 299
 do pacote java.lang, 854-859
 estendendo, 316
 genéricas, 519-522
 implementando, 300-304
 nomeando, 564
 usando referências de interface,
 304-306
interfaces aninhadas, 317
interfaces Callable e Future,
1015-1018
interfaces gráficas de usuário
(GUIs), 609
 controles e menus como recur-
 sos principais, 700
Internet Protocol (IP), 968
 IPv4 e IPv6, 968, 969
interpretador, 8-9
interpretando, 8-9
InterruptedException, 432, 778
intValue(), 483, 484, 504
invariante de classe, 566
invokeAndWait(), 619, 767, 773,
780
invokeLater(), 619, 645, 767,
768, 771-773, 780
IOException, 77, 359, 360, 381,
386, 393, 402
isAlive(), 440, 441
isDigit(), 834
isDirectory(), 410, 753
isEventDispatchThread(), 772
isFile(), 753
isInfinite(), 830-831
isLetterorDigit(), 834
isNaN(), 830-831

isSelected(), 658-660, 663
isSpaceChar(), 834, 836
isWhitespace(), 836
iteradores, usando, 942-946

J

janela de status, applets, 550
janelas nativas ou pares, 610
JApplet, 613, 779
Java
 applets, 11-12
 bytecode, 13-14
 C++ e, 10-11
 contribuição para a Internet, 11-
 12
 evolução de, 13-14
 origens de, 9-10
 portabilidade, 12-13
 segurança, 11-12
java.awt.event.InputEvent, 710
java.awt.event.KeyEvent, 709
Java Development Kit (JDK),
13-14, 17
 baixando e instalando, 17
 versão atual (JDK 7), 13-14
Java Runtime Environment (JRE),
13-14
Java SE 7, 13-14
javac, 17
javadoc, 570, 1041-1047
 exemplo usando comentários de
 documentação, 1046
 forma geral de comentários de
 documentação, 1045
 tags, 1041
 tags autônomas, 1042-1045
javax.swing.Timer, 773-777
 JButton, 625-628, 638, 655-658
 classe interna anônima como
 ouvinte de ação, 645
 desativando um botão, 657
JCheckBox, 660-662
JCheckBoxMenuItem, 701, 719-
722
JComboBox, 686-689
JComponent, 613, 650, 653, 686,
752, 787
 setToolTipText(), 712
JDialog, 613, 727, 746-750
 classes herdadas do AWT, 746
 criando uma caixa de diálogo
 modal, 746-750
 criando uma caixa de diálogo
 não modal, 750

JFileChooser
 outras opções suportadas por,
 761
 selecionando arquivos com,
 751-755
 usando filtro de arquivos com
 JFileChooser, 755-761
JFrame, 613, 615
 criando, 617
 getContentPane(), 619
 setJMenuBar(), 702, 704
JLabel, 615, 618, 627, 633, 639,
767
 método setText(), 628
 rolando texto em, 782, 785
 usando com ImageIcon, 650-
 653
JList, 682-686
JMenu, 701, 703
 adicionando mnemônicos, 709
 criando e adicionando JMenuI-
 tems, 704
JMenuBar, 701, 702, 704
JMenuItem, 701, 704
 adicionando aceleradores, 710
 adicionando ícones e dicas de
 ferramentas, 712
 adicionando mnemônicos, 709
 JOptionPane, 726
 métodos show, tipos de, 727
 showConfirmDialog(), 732-
 736
 showInputDialog(), 736-741
 showMessageDialog(), 727-
 731
 showOutputDialog(), 741-745
 tipos de caixas de diálogo su-
 portadas, 726
 JPanel, 613
 rolando, usando JScrollPane,
 677-682
JPopupMenu, 722
JRadioButton, 662-664
JRadioButtonMenuItem, 701,
719-722
JRE. Consulte Java Runtime
Environment
JRootPane, 613
JScrollPane, 675-682, 690, 693
JTable, 693-695
JTextField, 633-639, 665-668
JToggleButton, 658-660
JTree, 689-693
JVM. Consulte Máquina Virtual
Java
JWindow, 613

L

laço do-while, 95-98
 instrução continue em, 106
 laço for melhorado, 93
 usando com arrays, 172-178
 laço infinito, 91
 laços, 30, 98. *Consulte também* instruções de controle
 aninhados, 111
 do-while, 95-97
 for, 30-32, 88-93
 instrução continue em laços
 while e do-while, 106
 saindo com instrução break,
 100-102
 while, 93-95
 laços aninhados, 111
 laços for, 30-32, 35, 88-93, 98
 com corpo vazio, 91
 declarando variáveis de con-
 trole de laço em instrução for,
 92
 deixando partes da definição
 vazias, 90
 laço infinito, 91
 melhorados, 93, 172-178
 variações de, 89
 laços for de estilo for-each, 93,
 172-178
 laços while, 93-95, 98
 instrução continue em, 106
 Lei de Demeter, 579-581
 linguagem simbólica, 14-15
 linguagens de alto nível, 8-9
 linguagens de baixo nível, 8-9
 linguagens de programação, 8-9
 linguagens fortemente tipadas,
 42
 list(), 410-412
 listas encadeadas, 913
 listas separadas por vírgulas, 30
 ListSelectionEvent, 682, 683
 ListSelectionListener, 682, 683
 ListSelectionModel, 683
 literais, 49
 escapes de caracteres, 51
 hexadecimais, octais e binários,
 50
 string, 51
 literais binários, 50
 literais de string, 51, 803
 literais hexadecimais, 50
 literais inteiros, 50
 literais octais, 50
 log(), 839
 log10(), 839

M

mapas, 946
 classe HashMap, 952
 classe LinkedHashMap, 955
 classe TreeMap, 953-955
 classes de mapa, 951
 interface Map, 947
 interface Map.Entry, 949
 interface NavigableMap, 948,
 950-951
 interfaces de mapa, 946
 mapeamento de caracteres, 802
 máquina virtual, 7-8
 Máquina Virtual Java (JVM),
 12-13
 tratamento de exceções, 346
 matches(), 1056
 membros, classe, 15-16
 membros, interface, 315
 membros de classes
 controlando o acesso a, 202-
 208
 estáticos, 229
 públicos ou privados, 21
 superclasse, acessando com
 super, 265
 memória, 5-6, 845
 mensagens de erro, recuperando,
 355
 menu File
 adicionando aceleradores e
 mnemônicos, 711
 criando, 708
 menus, 700-724
 adicionando imagens e dicas de
 ferramentas a itens de menu,
 712
 adicionando mnemônicos e
 aceleradores a itens, 709-711
 adicionando/removendo itens
 de menu dinamicamente, 713-
 719
 classe JMenu, 703
 classe JMenuBar, 702
 classe JMenuItem, 704
 classes do sistema de menu de
 Swing, 700
 criando menu principal, 704-
 709
 itens de menu como botões,
 701
 menus popup ativados por clique
 no botão direito do mouse, 722
 usando JRadioButtonMenuItem
 e JCheckBoxMenuItem, 719-
 722
 menus popup ativados por clique
 no botão direito do mouse, 722
 metadados. *Consulte* anotações
 método addKeyListener(), 622-
 624
 método addMouseMotionListe-
 ner(), 622-624
 método charAt(), 178-179, 809-
 810, 857-858
 método currentThread(), 461
 método floatValue(), 483
 método getPriority(), 444
 método getSource(), 654, 663
 método getStateChange(), 655
 método getSupressed(), 392
 método getText(), 633, 635, 665,
 668
 método getTimeInMillis(), 631
 método indexOf(), 179-180, 817,
 819
 método itemStateChanged(), 654,
 658, 660
 método join(), 441, 443
 método lastIndexOf(), 179-180,
 817
 método length(), 178-179, 809,
 811, 857-858
 método listFiles(), 412
 método main(), 20, 77, 98, 123
 applets e, 541
 argumentos de linha de coman-
 do em array String, 184
 assegurando que main() termi-
 ne após thread, 433
 capturando exceções em, 345,
 355
 classe aninhada em escopo de
 bloco, 240
 cláusula throws em, 359
 constantes finais em, 293
 declarado público, 21
 objeto Thread em, 432
 referência de interface em, 305
 método mouseDragged(), 624
 método notify(), 295, 451-456,
 478, 479
 método notifyAll(), 295, 451
 método onAdvance(), 1009-1012
 método parseDouble(), 415,
 830-831
 método print(), 25, 382, 404
 método printf(), 383, 399, 890
 método println(), 21, 25, 382, 404
 exceções, 355
 exibindo linha em branco, 26
 objeto String como argumento,
 178-179

método seek(), 397
 método setCharAt(), 182, 825
 método sqrt(), 45, 46, 126, 335
 método toPath(), 413
 método val(), 490
 método value(), 490
 métodos, 20, 123-138
 abstratos, 288-291, 298
 acessando, 132
 acesso por operador ponto, 118
 adicionando a uma classe, 123-125
 adicionando método parametrizado a uma classe, 130-132
 assinatura, 219
autoboxing e, 485
 coesos, 564-566
 construtor, 138-142
 de enumerações, 472, 473
 elegantes, 565
 estáticos, 232
 final, impedindo a sobreposição, 292
 genéricos, 513-515, 530
 interface, 301
 nomeando, 563
 parâmetros de, 21, 128-130
 passando objetos para, 208-212
 recursivos, 225-229
 retornando de, 125
 retornando objetos, 212-214
 retornando um valor, 126-128
 sincronizados, 445-448
 sobrecarregando, 214-219
 sobrecarregando métodos variáveis, 244-247
 sobrepondo, 278-281, 283-287
 sobrepondo método genérico em hierarquia de classes genéricas, 518
 sobrepostos, importância de, 283
 suporte dos métodos sobrepostos ao polimorfismo, 281-283
 varargs, 241-244
 métodos auxiliares, 227
 métodos format(), 383
 métodos hasNext (Scanner), 422, 893, 894, 899
 métodos next (Scanner), 422, 893, 894
 métodos printStackTrace(), 355, 356
 mnemônicos, adicionando a itens de menu, 709, 722
 modelo de delegação de eventos, 621

modelos usados por controles de Swing, 683, 696
 modificador de acesso private, 202-208, 222, 256-259, 325
 modificador de acesso protected, 203, 325, 326
 acesso de pacote e, 328-330
 modificador de acesso public, 202-208, 256-259, 301, 314, 325
 modificador static, 229-234, 314
 blocos de código, 233
 classes aninhadas, 237-238, 241
 importação estática, 335-337
 método genérico, 513
 métodos, 232
 modificador abstract e, 288
 restrições a membros estáticos genéricos, 533
 variáveis, 230
 variáveis membros finais, 294, 337
 modificador transient, 554
 modificador volatile, 554
 modificadores de acesso, 203-208
 herança e, 256-259
 pacotes e acesso a membros, 326
 monitor, 445
 mouseMoved(), 624

N

\n (nova linha), 52
 nextInt(), 794
 nome de domínio, 968
 notifyObservers(), 902, 903
 NumberFormatException, 483, 832
 números, formatando, 877

O

objetos, 15-16, 117
 bem-formados, 566
 conteúdo da variável de instância, 119
 criando, 118, 121
 passando para métodos, 208-212
 referência this a, 146-149
 retornados por métodos, 212-214
 objetos de sincronização, 994
 ocultação de informações, 578

opção –classpath, 323
 operador AND, 60
 & (bitwise), 186-188
 & (lógico booleano), 59, 67
 && (AND de curto-circuito), 59, 61, 67
 operador &= (atribuição abreviada), 64
 operador AND condicional. Consulte operadores lógicos de curto-circuito
 operador de adição (+), 24, 57, 67
 operador de adição e atribuição (+=), 64
 operador de decremento (--), 32, 57, 58, 67
 operador de divisão (/), 24, 57, 67
 operador de divisão e atribuição (/=), 64
 operador de incremento (++) , 57, 58, 67
 operador de módulo (%), 57, 67
 operador de módulo e atribuição (%=), 64
 operador de multiplicação (*), 24, 57
 operador de multiplicação e atribuição (*=), 64
 operador de subtração (-), 24, 57
 operador de subtração e atribuição (-=), 64
 operador diferente de (!=), 29, 59, 60
 operador Exclusive OR (XOR)
 ^ (XOR bitwise), 186, 187, 189
 ^ (XOR lógico), 59, 60
 operador igual a (==), 29, 813
 operador instanceof, 67, 554
 operador losango (<>), 530
 operador maior ou igual a (>=), 29, 59, 60, 67
 operador maior que (>), 29, 59, 60, 67
 operador menor ou igual a (<=), 29, 59, 60, 67
 operador menor que (<), 29, 59, 60, 67
 operador new, 121, 142, 153, 156, 354, 530
 operador NOT
 ~ (unário bitwise), 186, 187, 190
 ! (unário lógico booleano), 59, 60
 operador OR
 operador | (OR bitwise), 186-188
 operador | (OR lógico), 59, 60, 67

operador `||` (OR de curto-circuito), 59-61, 67
 operador `|=` (atribuição abreviada), 64
 operador `^` (XOR (exclusive OR)), 59, 60
 operador `^=` (atribuição abreviada), 64
 operador OR condicional. *Consulte* operadores lógicos de curto-círcito
 operador ponto, 118, 124, 125, 132, 230
 operador XOR (exclusive OR)
 `^` (OR bitweise), 186, 187, 189
 `^` (OR lógico), 59, 60
 operadores, 8-9, 57-69
 aritméticos, 24, 57-59
 bitwise, 186-196
 de atribuição (abreviada) compostos, 63
 de atribuição, 63
 lógicos, 59-63
 operador ternário (`:?`), 67, 197
 operadores lógicos de curto-círcito, 61
 precedência de, 48, 67, 71
 relacionais, 29, 48, 59-61
 operadores aritméticos, 24, 57-59, 67
 operadores bitwise, 186-196
 operadores de atribuição, 63, 67
 operadores de atribuição abreviada, 63
 operadores de atribuição compostos, 64
 operadores de deslocamento, 191-193
 operadores lógicos, 59-63
 curto-círcito, 61
 exibindo tabela-verdade de, 68
 operando de tipo Boolean, 60
 precedência de, 67
 operadores lógicos de atribuição abreviada, 64
 operadores lógicos de curto-círcito, 59, 61
 operadores relacionais, 29, 59-61
 precedência de, 67
 tipo de saída booleano, 48
 ordinal(), 474-477
 ouvinte de movimento do mouse, 622-624
 ouvintes de eventos, 622-624
 classes adaptadoras para, 622-624

P

pacote java, 331
 pacote java.applet, 332, 540
 pacote java.awt, 332, 627
 pacote java.awt.event, 621-624, 627, 722
 resumo de classes de eventos, 623
 pacote java.io, 332, 360, 390, 797
 pacote java.lang, 332, 360, 379, 390, 797-827, 828-861
 pacote java.lang.annotation, 489
 pacote java.net, 332, 967-990
 pacote java.nio, 414
 pacote java.util, 332, 629, 797, 862-911
 pacote java.util.concurrent, 450, 911, 992
 pacote java.util.concurrent.atomic, 993, 1023
 pacote java.util.concurrent.locks, 993, 1020-1023
 pacote javax.swing, 617, 627
 pacote javax.swing.event, 621-624, 682
 resumo de classes de eventos, 623
 pacote javax.swing.table, 693
 pacote javax.swing.tree, 689
 pacote NIO, 407
 pacotes, 321-340
 acesso a membro, 325-328
 definição, 322
 encontrando, 323
 importação estática, 335-337
 importando, 330
 importando pacotes Java padrão, 331
 movendo classe para outro pacote, 332-334
 padrão Adapter, 594-597
 padrão Expert, 572
 padrão Observer, 597-602
 padrões de projeto, 593-602
 padrão Adapter, 594-597
 padrão Observer, 597-602
 painéis, 613
 painéis de contêiner de nível superior, 613
 painel de conteúdo, 613, 614
 painel de vidro, 613, 614
 painel em camadas, 613, 614
 paint(), 540, 543-545, 547, 548, 787, 789
 paintBorder(), 787, 789
 paintChildren(), 787, 789
 paintComponent(), 787-789, 793
 palavra-chave assert, 555
 palavra-chave catch, 342
 palavra-chave class, 20, 117
 palavra-chave enum, 468
 palavra-chave extends, 253, 254
 curingas limitados, 512
 tipo limitado para argumentos de tipo, 505, 506
 palavra-chave final, 292-294, 314, 337
 enumerações e variáveis finais, 474
 parâmetros multi-catch, 363
 recurso de relançamento, 364
 recursos de try-with-resources, 391
 palavra-chave finally, 342
 palavra-chave int, 24
 palavra-chave interface, 298, 299
 palavra-chave native, 555
 palavra-chave private, 21
 palavra-chave protected, 143
 palavra-chave public, 20
 palavra-chave strictfp, 554
 palavra-chave throw, 342
 palavra-chave throws, 342
 palavra-chave try, 342
 animando blocos try, 351-353
 blocos try sem cláusula catch, 344
 palavras-chave, 8-9, 36, 553-557
 parâmetro args, 21
 parâmetros, 123, 128-130
 adicionando a construtores, 139
 adicionando método parametrizado a uma classe, 130-132
 de métodos sobrecarregados, conversões automáticas de tipos, 216-219
 em quantidade variável (varargs) e normais, 243
 finais, 294
 passando para applets, 551
 valores de tipo primitivo passados para, 210
 parâmetros de tipo, 497, 499
 argumentos curingas, 507-510
 argumentos de tipo passados para, 500
 classe genérica com dois, 501-503
 curingas limitados, 510-512

erasure e, 531
especificando tipo de retorno de método, 499
impossibilidade de instanciar, 532
limitados, 504-507
para construtores genéricos, 515
para interfaces genéricas, 521
para métodos genéricos, 513-515
para objetos genéricos, 499
parênteses
 após nomes de métodos, 123
 em expressões, 71
pares, 610
parseBoolean(), 837
parseByte(), 832
parseFloat(), 830-831
parseInt(), 415, 832
parseLong(), 832
parseShort(), 832
paste(), JTextField, 665, 666, 668
pilhas, 912
polimorfismo, 16, 590
 na sobrecarga de métodos, 218
 referências de interface e, 305
 suporte de métodos sobrepostos, 281-283
pontuação, 8-9
portabilidade, 10-13
pós-condições, 569
pow(), 335, 839, 841
precedênciá, operador, 48, 67, 71
pré-condições, 569
primeiro a entrar, primeiro a sair (FIFO), 912
Princípio Aberto-Fechado, 577
princípio de menor perplexidade, 586
princípio DRY (don't repeat yourself), 573
prioridades de threads, 450
process(), 778
programa do cronômetro
 cronômetro simples, 629-633
 usando Timer, 774-777
 versão melhorada, 768-772
programa java (iniciador de aplicativo), 17
 chamada do método main(), 21
 executando um programa, 19
programação com várias threads.
Consulte threads
programação estruturada, 14-15

programação na Internet, Java e, 10-11
programação orientada a objetos (OOP), 14-17
 documentação interna, 567
 encapsulamento, 15-16
 herança, 16
 polimorfismo, 16
programação paralela. *Consulte Framework Fork/Join*
programas, 7-8
 análise linha a linha de, 19-22
 compilando, 19
 exemplo de programa, convertendo galões em litros, 27
 inserindo, 18
 primeiro programa simples, 18
 segundo programa simples, 23
projeto orientado a objetos, 559-606
 aprendendo sobre princípios e padrões de projeto, 562
 classes elegantes, 571
 coesão de classes e padrão Expert, 571-573
 coesão de métodos, 564-566
 convenções de nomenclatura, 563
 documentação externa, 569-571
 documentação interna, 567
 evitando duplicação, 573-575
 herança versus delegação, 581-593
 interface completa para classes, 575
 Lei de Demeter, 579-581
 métodos elegantes, 563
 objetos bem-formados, 568
 padrões de design, 593-602
 projetando visando mudanças, 576-579
 propriedades de um software elegante, 561
 software elegante e sua importância, 560
propriedade length, arrays, 164-167
publicadores e assinantes, 599-600
publish(), 778

R

RAM (Random Access Memory), 5-6
random(), 840
rastreamento de pilha, 346
read(), 381, 384, 389, 397, 402
readInt(), 397
readLine(), 399, 403, 407, 415
readPassword(), 399
recuo de código, 34
recursão, 225-229, 235-237
rede, 967-990
 baixando arquivo da Internet, 980-982
 classe DatagramPacket, 985
 classe DatagramSocket, 985
 classe HttpURLConnection, 982-984
 classe InetAddress, 969-971
 classe Socket, 971-975
 classe URL, 975-977
 classeURLConnection, 977-980
 classe e interfaces, 969
 datagramas, 984-989
 HTTP (Hypertext Transfer Protocol), 968
 Internet Protocol (IP), 968
 Sistema de Nome de Domínio (DNS), 968
 soquetes, 967
 Transmission Control Protocol (TCP), 968
 User Datagram Protocol (UDP), 968
referências da superclasse e objetos da subclasse, 273-278
regionMatches(), 814
regras de promoção de tipos, 69
relacionamento é-um, 584-588
relançando exceções, 354, 364
remove(), 702, 703
removeActionListener(), 623, 633
removendo ouvintes de eventos, 622-624
repaint(), 545, 547, 789, 794
replace(), 820
replaceAll(), 958-959, 1055
reverse(), 958-959
rolando
 componentes com JScrollPane, 675-677
 rolando texto em um applet, 782-787
 rolando um JPanel usando JScrollPane, 677-682

- rotate(), 958-959
 rotateLeft(), 833
 rotateRight(), 833
 chamado por thread de despatcho de evento em Swing, 619
 run(), 430-433, 436, 477, 547, 767, 771
 suspenso, retomando e interrompendo threads, 457-460
 rótulos. *Consulte também JLabel*
 instrução break com, 102
 instrução continue com, 106
 rótulo exibindo várias linhas de texto, 652
- S**
- ScheduledExecutorService, 1013
 ScheduledThreadPoolExecutor, 1013
 segurança, applets e, 11-12
 segurança de tipos
 assegurada por genéricos, 500
 referências de objeto e, 497
 seleção de arquivos com JFileChooser, 751-755
 sequência de entrada, 1049
 sequências de escape, 51
 sequências de escape de caracteres, 51
 servlets, 13-14
 setActionCommand(), 635, 638, 655, 663, 665
 setBorder(), 793
 setChanged(), 902
 setDefaultCloseOperation(), 617
 setDisabled(), 657
 setDisabledIcon(), 712
 setDisplayedMnemonicIndex(), 709
 setEnabled(), 657, 704
 setFileFilter(), 756, 758
 setFileHidingEnabled(), 762
 setFileSelectionMode(), 761
 setIcon(), 652, 712
 setJMenuBar(), 702, 704
 setMaximumSize(), 686
 setMinimumSize(), 686
 setMnemonic(), 709
 setMultiSelectionEnabled(), 761
 setName(), 433
 setPreferredSize(), 683, 686
 setPressedIcon(), 657
 setPriority(), 444, 450
 setRepeats(), 774
 setRolloverIcon(), 657
 setSelectedIndex(), 685
 setSize(), 617
 setText(), 628, 631, 665
 setToolTipText(), 712
 setVisible(), 618, 746, 750
 Shared.count, 1021
 showConfirmDialog(), 727, 732-736
 showDialog(), 752
 showInputDialog(), 727, 736-741
 showMessageDialog(), 727-731
 showOpenDialog(), 752, 755
 showOptionDialog(), 727, 741-745
 showSaveDialog(), 752, 755
 showStatus(), 550
 showType(), 499
 sincronização, 429, 444
 Sistema de Nome de Domínio (DNS), 968
 sistema de números binários, 6-7
 sistemas multiprocessadores e *multicore*, 429
 sistemas operacionais, 6-7
 multitarefa e o tempo de CPU para as threads, 444
 sistemas *single-core*, 429
 skip(), 899
 sleep(), 432, 433
 sobrecregendo construtores, 219-224
 sobrecregendo métodos, 214-219
 métodos varargs, 244-247
 sobrepondo métodos, 278-281, 283-287
 impedindo com final, 292
 importância de, 283
 método genérico em hierarquia de classes genéricas, 420
 suporte dos métodos sobrepostos ao polimorfismo, 281-283
 soquetes, 967
 soquetes TCP/IP, 971
 sort(), 959
 start(), 431, 432, 543, 544, 547, 773, 779, 842
 startsWith(), 814
 stop(), 543, 544, 547, 773, 779, 784
 strings, 51, 178-186, 799-827
 alterando a caixa dos caracteres, de 822
 arrays de, 181
 caracteres versus, 52
 classe String, 178, 799, 800, 857-858
 comparando, 812-817
 concatenando, 803
 construindo, 149
 construtores, 800-802
 imutabilidade de, 182, 800
 método charAt(), 809-810
 método length(), 809
 método toString(), 295
 métodos matches() de String, 1056
 objetos String, 21, 799
 obtendo string modificado, 819-822
 operando com, 178-181
 outros métodos de String, 824
 pesquisando usando indexOf() e lastIndexOf(), 817
 sobrepondo toString(), 804
 toCharArray(), 811
 usando para controlar instrução switch, 183
 vazias, 800, 805
 subclasse, 253, 291. *Consulte também herança*
 capturando exceções da subclasse, 350
 referências da superclasse e objetos da subclasse, 273-278
 tipos limitados, 505
 submenus, 701
 subSequence(), 857-858
 substituído, 490
 substring(), 183, 819
 Sun Microsystems, 9-10, 13-14
 super
 acessando membros da superclasse, 265
 acessando versão da superclasse de método sobreposto, 279-281, 290
 chamando construtor de Thread, 436
 chamando paintComponent(), 788, 793
 super(), 261-265
 usando para chamar construtores da superclasse, 261-265
 superclasse, 253. *Consulte também herança*
 criando e estendendo, 254-256
 tipos limitados, 515
 Swing, 540, 609-648
 applets baseados em, 539, 779-787
 componentes e contêineres, 612-614

criando cronômetro simples, 629-633
 criando máquina de codificação simples, 638-645
 eventos e tratamento de eventos, 621-624
 gerando componentes em, 787-794
JTextField, 633-638
 origens e filosofia de projeto, 610-612
 primeiro programa simples, 615-619
 usando botão de ação, 625-628
 usando classes internas anônimas para tratar eventos, 645

T

tabelas, criando com **JTable**, 693-695
 tabelas hash, 915
tag APPLET, HTML, 541
 atributo PARAM, 551
 tamanho de controle **JList**, definindo, 683
TCP (Transmission Control Protocol), 968
TCP/IP, 968
 tecla ALT, acelerador, 710
 tecla CTRL, máscara para, tecla SHIFT, acelerador, 710
 tempo de vida das variáveis, 55
 texto, rolando em um applet, 782-787
 this (palavra-chave), 146-149
 this(), 556
 thread de despacho de evento em programas Swing, 618, 628, 767, 772
 thread principal, 461
ThreadPoolExecutor, 1013
 threads de segundo plano, gerenciando com **SwingWorker**, 777-779
 tipo booleano, 47, 60
 conversão em outros tipos, 64
 para resultados de condição de laço, 90
 resultados de expressões condicionais controlando instruções if, 78

tipo byte, 43, 44, 45, 50, 82
 aplicação de operadores bitwise, 186
 conversão em outros tipos, 65, 69
 tipo char, 43, 45, 46, 50, 82
 aplicação de operadores bitwise, 186
 conversões de e para outros tipos, 65, 69, 70
 entrada do teclado, 76
 operações aritméticas com, 57
 tipo de retorno void, 123, 125-126, 227
 tipo double, 25, 26, 45, 50, 198
 conversões de e para outros tipos, 65, 66, 69
 tipo flutuante, 25, 26, 45
 conversão de e para outros tipos, 64, 69
 tipo int, 24, 25, 43, 44, 45, 82
 aplicação de operadores bitwise a, 186
 conversão em atribuições, 64
 literais inteiros como, 50
 promoção de outros tipos a, 69
 tipo long, 43, 44, 45, 50
 aplicação de operadores bitwise a, 186
 conversão de e para outros tipos, 65, 66, 69
 tipo short, 43-45, 50, 82
 conversões de e para outros tipos, 66, 69
 operadores bitwise e, 186
 tipos. *Consulte* tipos de dados
 tipos brutos, 526-529
 tipos de dados, 26, 42-57
 argumentos e parâmetros de métodos, 129
 arrays, 21, 152-178
 atribuições de variáveis de referência, 121
 conversão de tipos de parâmetros de métodos sobrecarregados, 216-218
 conversão em atribuições, 64
 conversão em expressões, 69-71
 conversão usando uma coerção, 65-67
 encapsuladores de tipos para a conversão de strings numéricos, 414-416
 especificando para variáveis, 24
 importância de, 42
 literais, 49
 operadores bitwise aplicados a, primitivos, 43-49
 referências da superclasse e objetos da subclasse, 273-278
 strings, 51, 178-186
 tipo de retorno de métodos, 123
 tipos de ponto flutuante, float e double, 25, 45
 tipos de retorno, 123
 tipos limitados, 504-507
 argumentos curingas, 510-512
 tipos primitivos, 43-49
 atribuição de variáveis de referência, 121
 booleanos, 47
 byte, short, int e long, 43-45
 char, 46
 classes encapsuladoras para, 212, 414-416, 482-488, 829-838
 float e double, 45
 lendo/gravando valores binários de, 392-395, 397-399
 métodos para ler/gravar, 397
 passando para métodos, 210, 212
 variáveis de, operador new e, 142
toBinaryString(), 833
toDegrees(), 839
toHexString(), 833
toLowerCase(), 822
toOctalString(), 833
toRadians(), 839
toString(), 295, 355, 365, 404, 483
 adicionando à classe **GenSimpleStack** (exemplo), 806-809
 classe Boolean, 837
 definido por Byte, Short, Integer ou Long, 833
 sobrepondo, 804
totalMemory(), 844
toUpperCase(), 822
 tratamento de eventos
 em programas Swing, 621-624
 usando classes internas anônimas, 645
 tratamento de exceções, 341-375
 adicionando exceções a classes de pilha, 367-371
 blocos try aninhados, 351-353

- classe `Throwable`, 355-357
consequências de exceção não capturada, 346
criando subclasses de `Exception`, 364-366
exceções internas Java, 360-362
exemplo simples, 343-345
fornecendo respostas elegantes a erros, 347
hierarquia de classes de exceção, 342
lançando exceções, 353
palavras-chave no tratamento de exceções, 342
recursos novos do JDK 7, 363
relançando exceções, 354
tratador de exceções, 341
usando, versus criando classes de exceção personalizadas, 366
usando blocos `finally`, 357-359
usando `throws`, 359
usando `try` e `catch`, 343
usando várias cláusulas `catch`, 349-351
`TreeSelectionEvent`, 689
`TreeSelectionListener`, 689, 691
`trim()`, 821
- U**
- último a entrar, primeiro a sair (`LIFO`), 912
`Unicode`, 46, 47
`update()`, 902, 903
`updateTime()`, 772, 776
`useDelimiter()`, 899
User Datagram Protocol (UDP), 968
uso de threads, 428-466, 547, 766-779
 classe `SwingWorker`, 777-779
 classe `Thread` e interface `Runnable`, 429
 classes relacionadas a threads e a interface `Runnable`, 853
 comunicação usando `notify()`, `wait()` e `notifyAll()`, 451-456
 configurações de prioridade, 443
 criando uma thread, 430-435, 440
criando várias threads, 438-440
deadlock e condições de corrida, 456
despacho de evento, programas Swing, 618
determinando quando a thread termina, 440-443
em semáforo controlado por computador, 477-483
estendendo `Thread`, 435-437
fundamentos do uso de várias threads, 428
instrução `synchronized`, 448-450
sincronização de threads, 444
suspendendo, retomando e interrompendo, 457-460
usando a thread principal, 461
usando métodos sincronizados, 445-448
usando o multithreading de maneira eficaz, 460
usando `Timer`, 773-777
uso de várias threads em Swing, 766-772
versão melhorada do cronômetro, 768-772
utilitário de comparação de arquivos
 baseado em Swing, 668-675
 baseado no console, 395-397
utilitários de concorrência, 450, 991-1040
 abordagem Java tradicional versus, 1037
 bloqueios, 1020-1023
 `Callable` e `Future`, 1015-1018
 classe `CountDownLatch`, 995-999
 classe `CyclicBarrier`, 1000-1002
 classe `Exchanger`, 1002-1004
 classe `Phaser`, 1005-1010
 classe `Semaphore`, 994-997
 coleções de concorrência, 1020
 enumeração `TimeUnit`, 1018, 1020
 objetos de sincronização, 994
 operações atômicas, 1023
 pacote `java.util.concurrent`, 992
 pacote `java.util.concurrent.atomic`, 993
 pacote `java.util.concurrent.locks`, 993, 1020
programação paralela via Framework Fork/Join, 1024-1037
usando um executor, 1012, 1015
- V**
- valor nulo para variáveis de instância, 140
valores de ponto flutuante, 26
 `strictfp`, 554
 tipos `double` e `float`, 25, 45
valores de propriedades, obtendo, 849
valores true/false (tipo booleano), 47
`valueChanged()`, 682, 689, 691
`valueOf()`, 471, 472, 824, 853
`values()`, 471, 472
varargs (argumentos em quantidade variável), 241-247
 ambiguidade em métodos sobrecarregados, 246
 sobrecarregando métodos variargs, 244-246
variáveis, 23-25, 52-56
 controle laço, 30, 31
 declarando, 24
 escopo e tempo de vida de, 54-56
 especificando tipo, 24, 25
 estáticas, 230-232
 finais, 294
 inicialização dinâmica, 53
 inicializando, 53
 interface, 314
 nomeando, 37, 38, 564
variáveis de fluxo de `in`, `out` e `err`, 379
variáveis de instância, 15-16, 117, 118, 233
 acessando, 118, 132
 cópias contidas em objetos, 119
 definidas por enumerações, 472, 473
 envolvidas em invariantes de classe, 567
 tornando privadas, 259, 578
uso direto de, em métodos, 125
valores iniciais definidos por construtores, 138
variáveis estáticas versus, 230-232

variáveis de referência
atribuição e, 121
atribuindo referências de array,
163-165
referências da superclasse e
objetos da subclasse, 273-278
referências de interface, 304-
306
variáveis membro, 15-16
finais, 293, 294
variável count, 31
variável de ambiente CLASSPA-
TH, 323

verificação de tipo por genéricos,
500
visibilidade de uma janela, 618,
746, 750

write(), 382, 386, 397
writeDouble(), 397

X

XOR (exclusive OR) com opera-
dor de atribuição (^=), 64, 193

wait(), 295, 451-456, 479
waitFor(), 842, 843
World Wide Web, surgimento da,
10-11