JAVA A REFERÊNCIA COMPLETA

COBERTURA ABRANGENTE DA LINGUAGEM JAVA

Apresentação

Quando alguém escreve um livro torna-se o “dono” da língua, atribuindo sentido às palavras. Por esse motivo um livro é propriedade intelectual, não em relação a informação contida nele, mas ao trabalho realizado por alguém.

Mas a língua é um bem coletivo, os indivíduos revezam nesse papel, ora um toma a palavra e produz um texto, ora outro a toma e os demais precisam buscar um significado no que foi produzido.

Um livro não é produzido para ser um bem, um mero objeto de valor, para ser guardado, ele é feito para impactar o leitor a fim de compartilhar a visão de mundo do autor, a fim de que o leitor faça uso desse conhecimento para mudar sua própria realidade (seu pequeno grande mundo).

Esse e-book é uma tradução do livro “Java: The Complete Reference – Eleventh Edition” de Herbert Schildt.

Muito se diz em computação sobre propriedade intelectual, assim como em outras áreas, o autor quer ser reconhecido como o dono daquilo que foi produzido.

Como você verá, a linguagem Java começou a ser liberada como código aberto, em 2006. Embora toda criação tenha um dono legal, ela é feita para as pessoas que irão se beneficiar dela. “Não existe nenhuma academia no meio do deserto” – Scarpelly – isso quer dizer que: ninguém constrói músculos para se admirar com o próprio corpo. As pessoas fazem as coisas para mudar o mundo a sua volta, é isso que as motiva: a forma como vêm o mundo, e a forma como o mundo as vê.

Esse e-book foi traduzido (embora sem tanto conhecimento sobre a língua inglesa, com o auxílio de aplicativos de tradução e fazendo uso da interpretação de textos) a partir de uma versão online facilmente encontrada em formato pdf, o objetivo foi absorver o conhecimento sobre a linguagem Java que poderia ser adquirido por pesquisas, um processo mais lento e cansativo se forem usadas outras fontes de imediato. O mesmo será disponibilizado como fonte de estudo. Afinal, há duas coisas que se perde se forem guardadas: o conhecimento, e o afeto.

Sumário

Prefacio

Java é uma das linguagens de computador mais importantes e amplamente usadas no mundo. Além disso, ela manteve essa distinção por muitos anos. Ao contrário de algumas outras linguagens de computador cuja influência diminuiu com o passar do tempo, Java permaneceu forte. Java saltou para a frente da programação da Internet com seu primeiro lançamento. Cada versão subsequente solidificou essa posição. Hoje, ela continua sendo a primeira e a melhor escolha para desenvolvimento de aplicações baseadas na web. É também uma poderosa linguagem de programação para uso geral, adequada para uma ampla variedade de finalidades. Simplificando: Grande parte do mundo moderno roda em código Java. Java é realmente muito importante.

O motivo para o sucesso de Java é a agilidade. Desde o lançamento original da JDK 1.0, Java foi continuamente adaptada para mudanças no ambiente de programação e no modo como programadores programam. Mais importante, ela não apenas seguiu as tendências, mas ajudou a cria-las. A capacidade de acomodar a rápida taxa de mudanças no mundo da computação é uma parte crucial do motivo pelo qual tem sido e continua sendo bem sucedida.

Desde que foi lançada Java continuou evoluindo, e como resultado a versão Java SE 11 (JDK 11) tem uma quantidade substancial de novos materiais, atualizações e mudanças. A discussão de dois recursos principais que foram adicionados à Java desde a última versão é de interesse especial. O primeiro é a inferência do tipo de variável local porque simplifica alguns tipos de declarações de variáveis locais. Para dar suporte à inferência de tipo de variável local, o *nome* reservado **var,** em caso sensitivo, foi adicionado para a linguagem. O segundo novo recurso de Java é a reformulação do número da versão para refletir o que se espera ser um ciclo de lançamento rápido, que iniciou com a JDK 10. Como explicado no Capítulo 1, é esperado que os lançamentos de recursos Java ocorram a cada seis meses. Isso é importante porque agora é possível que novos recursos sejam adicionados mais rápido que no passado.

Há dois novos recursos adicionados à Java introduzidos em uma versão anterior que ainda têm um forte impacto nos programadores Java. O primeiro é o módulo, que permite especificar as relações e dependências do código que compreendem o aplicativo. A adição de módulos pela versão JDK 9 representa uma das mais profundas alterações já feitas para a linguagem Java. Por exemplo, ele resulta na adição de 10 palavras-chave em caso

sensitivo. Módulos também impactaram a biblioteca da API Java porque seus pacotes agora estão organizados em módulos. Além disso, para dar suporte aos módulos novas ferramentas foram adicionadas, ferramentas já existentes foram atualizadas, e um novo formato de arquivos foi definido. Devido a sua importância, todo o Capítulo 16 é dedicado aos módulos. O segundo recurso adicionado recentemente é o JShell. JShell é uma ferramenta que oferece um ambiente interativo no qual é fácil experimentar trechos de código sem precisar escrever um programa inteiro. Tanto iniciantes quanto profissionais experientes acharão isso bastante útil. Uma Introdução ao JShell é encontrada no Apêndice B.

O Que Há Dentro

Este livro é um guia abrangente para a linguagem Java, descrevendo sua sintaxe, palavras-chave e princípios fundamentais de programação. Partes significativas da biblioteca da API Java também são examinadas. O livro está dividido em quatro partes, cada uma focada em um aspecto diferente do ambiente de programação Java.

A Parte I apresenta um tutorial detalhado da linguagem Java. Começa com o básico, incluindo coisas como tipos de dados, operadores, instruções de controle e classes. Em seguida passa para herança, pacotes, interfaces, manipulação de exceções e multiprocessamento. Depois, descreve anotações, enumerações, caixa automática, genéricos e expressões lambda. A I/O também é introduzida. O capítulo final da Parte I aborda os módulos.

A Parte II examina os aspectos principais da biblioteca de API Java. Tópicos incluem strings, I/O, rede, Collections Framework, AWT, manipulação de eventos, geração de imagens, simultaneidade (incluindo Fork/Java Framework), expressões regulares e a biblioteca stream.

A Parte III oferece três capítulos que apresentam Swing.

A Parte IV contém dois capítulos que mostram exemplos de Java em ação. O primeiro discute Java Beans. O segundo apresenta uma introdução a servlets.

PARTE I A Linguagem Java

CAPÍTULO I A História e a Evolução de Java

Para entender Java completamente, é preciso entender as razões por trás de sua criação, as forças que a moldaram, e o legado que ela herdou. Como as bem-sucedidas linguagens de computador que vieram antes, Java é uma mistura dos melhores elementos de sua rica herança combinada com os conceitos inovadores exigidos por sua missão única. Enquanto os demais capítulos deste livro descrevem os aspectos práticos de Java – incluindo sua sintaxe, principais bibliotecas, e aplicações – este capítulo explica como e porque a linguagem Java surgiu, e como ela evoluiu ao longo dos anos.

Embora Java tenha se tornado inseparavelmente ligada ao ambiente online da Internet, é importante lembrar que Java é primeiramente e principalmente uma linguagem de programação. A Inovação e o desenvolvimento da linguagem de computador ocorrem por duas razões fundamentais:

- Para se adaptar às mudanças de ambiente e aos usos;
- Para implementar refinamentos e melhorias na arte da programação.

O desenvolvimento da linguagem Java foi impulsionado por ambos os elementos quase na mesma medida.

A Linhagem de Java

Java está relacionado à C++, que é um descendente direto de C. Grande parte do caráter de Java foi herdado dessas duas linguagens. De C, Java deriva a sintaxe. Muitos recursos de orientação a objetos de Java foram influenciados por C++. De fato, várias características definidoras de Java vêm de – ou são respostas para – suas predecessoras. Além disso, a criação de Java estava profundamente enraizada no processo de refinamento e adaptação que ocorreu nas linguagens de programação de computadores nas últimas décadas. Por esses motivos, esta seção revisa a sequência de eventos e forças que levaram à Java. Cada inovação no design na linguagem foi motivada pela necessidade de resolver um problema fundamental que as linguagens anteriores não puderam resolver. Java não é uma exceção.

O Nascimento da Programação Moderna: C

A linguagem C chocou o mundo dos computadores. Seu impacto não deve ser subestimado, porque mudou fundamentalmente a maneira como a programação era abordada e pensada. A criação de C foi um resultado direto da necessidade de uma linguagem estruturada, eficiente e de alto nível que pudesse substituir a montagem de código ao criar

sistemas de programas. Quando uma linguagem de computador é projetada, geralmente são feitas trocas, como as seguintes:

- Facilidade de uso versus poder;
- Segurança versus eficiência;
- Rigidez versus extensibilidade.

Antes de C, os programadores geralmente tinham de escolher entre linguagens que otimizavam um conjunto de características ou outro. Por exemplo, embora FORTRAN pudesse ser usada para escrever programas bastante eficientes para aplicações científicas, não era muito boa para o código de sistema. E embora BASIC fosse fácil de aprender, não era muito poderosa, e sua falta de estrutura tornava sua utilidade questionável para programas grandes. A linguagem Assembly pode ser usada para produzir programas altamente eficientes, mas não é fácil de aprender ou de usar com eficácia. Além disso, a depuração (procura por erros) do código da Assembly pode ser muito difícil.

Outro problema de composição foi que linguagens de computador antigas como BASIC, COBOL, e FORTRAN não foram projetadas em torno de princípios estruturados. Em vez disso, elas confiavam no GOTO como principal meio de controle do programa. Como resultado, programas escritos usando essas linguagens tendiam a produzir “código espaguete” – uma massa de saltos emaranhados e ramificações condicionadas que tornam um programa praticamente impossível de entender. Embora linguagens como Pascal sejam estruturadas, elas não foram projetadas para a eficiência, e falharam em incluir certos recursos necessários para torna-los aplicáveis a uma ampla variedade de programas. (Especialmente, considerando os dialetos padrão de Pascal disponíveis na época, não era prático considerar o uso da Pascal para código em nível de sistema).

Portanto, pouco antes da invenção de C, nenhuma língua havia reconciliado os atributos conflitantes que haviam perseguido os esforços anteriores. No entanto, a necessidade de uma linguagem assim era urgente. No início da década de 1970, a revolução dos computadores estava começando a acontecer, e a demanda por software ultrapassava rapidamente a capacidade de produção. Um grande esforço foi gasto em círculos acadêmicos, na tentativa de criar uma linguagem de computador melhor. Mas, e talvez a mais importante, uma força secundária estava começando a ser sentida. O hardware do computador estava finalmente se tornando comum o suficiente para atingir uma massa crítica. Os computadores não eram mais mantidos atrás de portas trancadas. Pois estavam sendo alcançados. Pela primeira vez, programadores estavam obtendo praticamente acesso ilimitado às suas máquinas. Isso permitiu a liberdade de experimentar. Também permitiu que

os programadores começassem a criar suas próprias ferramentas. Na véspera da criação de C, o palco estava montado para um salto quântico em linguagens de computador.

Inventada e implementada pela primeira vez por Dennis Ritchie em um DEC PDP-11 executando o sistema operacional UNIX, a linguagem C foi o resultado de um processo de desenvolvimento iniciado com uma linguagem mais antiga chamada BCPL, desenvolvida por Martin Richards. BCPL foi influenciada por uma linguagem mais antiga chamada B, inventada por Ken Thompson, que levou ao desenvolvimento de C na década de 1970. Por muitos anos, o padrão de fato fornecido para C foi o fornecido com o sistema operacional UNIX e descrito em *The C Programming Language* por Brian Kernighan e Dennis Ritchie. C foi padronizada formalmente em dezembro de 1989, quando foi adotado o padrão de ANSI (American National Standards Institute).

A criação de C é considerada por muitos por ter marcado o início da era moderna das linguagens de computador. Ela sintetizou com sucesso os atributos conflitantes que haviam incomodado tanto as linguagens anteriores. O resultado foi uma linguagem estruturada poderosa, eficiente e relativamente fácil de aprender. Isso também inclui um outro aspecto quase intangível: era a linguagem de *programador*. Antes da invenção de C, linguagens de computador eram geralmente designadas como exercícios acadêmicos ou por comitês burocráticos. C é diferente. Ela foi projetada, implementada e desenvolvida por programadores reais, que trabalham de verdade, refletindo a maneira como abordam o trabalho de programação. Seus recursos foram aprimorados, testados, refletidos e repensados pelas pessoas que realmente usam a linguagem. O resultado foi uma linguagem que programadores gostam de usar. De fato, C atraiu rapidamente muitos seguidores que tinham um zelo quase religioso por ela. Como tal, encontrou ampla e rápida aceitação na comunidade de programadores. Tal legado foi herdado por Java.

C++: O Próximo Passo

Durante o final da década de 1970 e início da década de 1980, C se tornou a linguagem de programação de computadores dominante, e ainda é amplamente usada. Mesmo com a linguagem C sendo útil e bem-sucedida, a crescente *complexidade* dos programas levou à necessidade de maneiras melhores de gerencia-la. C++ (lê-se “C mais mai” ou “C plus plus”) é uma resposta a essa necessidade. Para entender melhor por que o gerenciamento da complexidade do programa foi fundamental para a criação da C++, considere o seguinte.

As abordagens de programação mudaram drasticamente desde a inovação do computador. Por exemplo, quando computadores foram inventados, a programação era feita alterando manualmente as instruções da máquina binária usando o painel frontal. Desde

que os programas tivessem apenas algumas centenas de instruções, essa abordagem funcionava. Como os programas cresceram, a linguagem Assembly foi inventada para que um programador pudesse lidar com programas maiores, e cada vez mais complexos usando representações simbólicas das instruções de máquina. À medida que os programas continuavam a crescer, foram introduzidas linguagens de alto nível que deram ao programador mais ferramentas para lidar com a complexidade.

A primeira linguagem difundida foi FORTRAN. Embora FORTRAN tenha sido um primeiro passo impressionante, na época não era uma linguagem que incentivava programas claros e fáceis de entender. A década de 1960 deu origem para a *programação estruturada*. Esse é o método de programação defendido por linguagens como C. O uso de linguagens estruturadas permitiu aos programadores escrever, pela primeira vez, programas complexos com bastante facilidade. No entanto, mesmo com métodos de programação estruturados, uma vez que um projeto atinge um certo tamanho, sua complexidade excede o que um programador pode gerenciar. No início dos anos 80, muitos projetos estavam ultrapassando seus limites de abordagem estruturada. Para resolver este problema, uma nova maneira de programar foi inventada, chamada *programação orientada a objetos* (OOP). Programação orientada a objetos é discutida em detalhes mais adiante; mas aqui está uma breve definição: OOP é uma metodologia que ajuda a organizar programas complexos por meio do uso de herança, encapsulamento e polimorfismo (duas ou mais classes derivadas de uma mesma superclasse que invocam métodos que têm a mesma identificação).

Como análise final, embora C seja uma das melhores linguagens de programação do mundo, há um limite para sua capacidade. Uma vez que o tamanho de um programa excede certo ponto, ele se torna tão complexo que é difícil de entender totalmente. Embora o tamanho exato em que isso ocorra seja diferente, dependendo da natureza do programa e do programador, sempre há um limite no qual um programa se torna incontrolável. C++ adicionou recursos que permitiram que esse limite fosse quebrado, permitindo que os programadores compreendessem e gerenciassem programas maiores.

C++ foi inventado por Bjarne Stroustrup em 1979, enquanto trabalhava nos Laboratórios Bell em Murray Hill, Nova Jersey. Stroustrup inicialmente chamou a nova linguagem de “C com Classes”. No entanto, em 1983, o nome foi alterado para C++. C++ estende C adicionando recursos orientados a objetos. Como C++ é construída na linguagem C, ela inclui todos os recursos, atributos e benefícios de C. Esta é a razão crucial para o sucesso de C++ como uma linguagem.

A invenção de C++ não foi uma tentativa de criar uma linguagem de programação completamente nova. Em vez disso, foi um aprimoramento para uma já altamente bem sucedida.

O Cenário Está Pronto Para Java

No final dos anos 1980 e início dos anos 1990, a programação orientada a objetos usando C++ entrou em vigor. De fato, por um breve momento pareceu que os programadores tinham finalmente encontrado a linguagem perfeita. Porque C++ combinou os elementos de alta eficiência e estilística de C com o paradigma da orientação a objetos, C++ era uma linguagem que poderia ser usada para criar uma ampla gama de programas. No entanto, assim como no passado, surgiram forças que, mais uma vez, impulsionaram a evolução da linguagem de computador. Dentro de alguns anos, a World Wide Web e a Internet atingiram uma massa crítica. Este evento precipitaria outra revolução na programação.

A Criação de Java

Java foi concebida por James Gosling, Patrick Naughton, Chris Warth, Ed Frank e Mike Sheridan na Sun Microsystems, Inc. em 1991. Demorou 18 meses para desenvolver a primeira versão mundial. Essa linguagem foi inicialmente chamada de “Oak”, mas foi renomeada para “Java” em 1995. Entre a implementação inicial do Oak no outono de 1992 e o anuncio público da Java na primavera de 1995, muito mais pessoas contribuíram para o design e a evolução da linguagem. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin e Tim Lindholm foram os principais contribuintes para o amadurecimento do protótipo original.

Surpreendentemente, o impulso original para Java não era a Internet! Em vez disso, a principal motivação foi a necessidade de uma linguagem independente de plataforma (arquitetura neutra) que pode ser usada para criar softwares a serem incorporados em vários dispositivos eletrônicos de consumo, como fornos micro-ondas e controles remotos. Como você provavelmente pode adivinhar, muitos tipos diferentes de CPUs são usados como controladores. O problema com C e C++ (e com a maioria das outras linguagens) é que elas são projetadas para serem compiladas para um destino específico. Embora seja possível compilar um programa em C++ para praticamente qualquer tipo de CPU, para isso é necessário um compilador C++ completamente direcionado para tais CPUs. O problema é que os compiladores são caros e demorados para criar. Uma solução mais fácil – e mais barata – era necessária. Na tentativa de encontrar uma solução, Gosling e outros começaram a trabalhar em uma linguagem portátil independentemente da plataforma que pudesse

ser usada para produzir o código que seria executado em uma variedade de CPUs em ambientes diferentes. Esse esforço levou à criação de Java.

Na época em que os detalhes da linguagem Java estavam sendo elaborados, um segundo, e mais importante, fator estava surgindo e teria um papel crucial no futuro de Java.

Essa segunda força foi, é claro, a World Wide Web. Se a web não tivesse se formado quase na mesma época em que a Java estava sendo implementada, Java poderia ter permanecido uma linguagem útil, porém obscura para a programação de eletrônicos de consumo. No entanto, com o surgimento da WWW, Java foi impulsionada para a frente como modelo de linguagens de computador, porque a web também exigia programas portáteis.

A maioria dos programadores aprende, no início de suas carreiras, que os programas portáteis são tão esquivos quanto desejáveis. Embora a busca por uma maneira de criar programas portáteis eficientes seja quase tão antiga quanto a disciplina da programação em si, ela ficou atrás de outros problemas mais urgentes. Além disso, porque (naquela época) grande parte do mundo dos computadores se dividiu nos três campos concorrentes da Intel, Macintosh, e UNIX, a maioria dos programadores permaneceu dentro de seus limites fortificados e a necessidade de códigos portáteis foi reduzida. No entanto, com o advento da Internet e da web, o antigo problema da portabilidade voltou com força total. Afinal a Internet consiste de um universo diversificado e distribuído, preenchido com vários tipos de computadores, sistemas operacionais e CPUs. Mesmo que muitos tipos de plataformas estejam conectados à Internet, os usuários gostam que todos possam executar o mesmo programa. O que antes era um problema irritante, mas de baixa prioridade, tornou-se uma necessidade de alto nível.

Em 1993, tornou-se óbvio para os membros da equipe de design de Java que os problemas de portabilidade frequentemente encontrados ao criar códigos para controladores incorporados também eram encontrados ao criar códigos para a Internet. De fato, o mesmo problema que a linguagem Java foi projetada para resolver em pequena escala também pode ser aplicado à Internet em larga escala. Essa percepção fez com que o foco da linguagem passasse dos eletrônicos de consumo para a programação da Internet. Portanto, enquanto o desejo por uma linguagem de programação neutra em arquitetura forneceu a faísca inicial, a Internet acabou levando Java ao sucesso em larga escala.

Como mencionado anteriormente, Java deriva muito em caráter de C e C++. Os designers de Java sabiam que o uso da sintaxe familiar de C e a propagação dos recursos orientados a objetos de C++ tornariam sua linguagem atraente para as legiões de programadores experientes em C/C++. Além das semelhanças superficiais. Java compartilha

alguns dos outros atributos que ajudaram C e C++ a alcançarem o sucesso. Primeiramente, Java foi projetada, testada e refinada por programadores reais que trabalham de verdade. É uma linguagem baseada nas necessidades e experiências das pessoas que a criaram. Em segundo lugar, Java é coesa e logicamente consistente. Em terceiro lugar, Exceto por essas restrições impostas pelo ambiente da Internet, Java oferece ao programador total controle. Sendo assim, os programas refletem fidedignamente a capacidade do programador.

Devido às semelhanças entre Java e C++, é tentador pensar em Java simplesmente como a "versão para a Internet de C++". No entanto, fazer isso seria um grande erro. Java possui diferenças práticas e filosofias diferentes. Embora seja verdade que Java foi influenciada por C++, ela não é uma versão aprimorada de C++. Por exemplo, Java não é compatível com C++. Obviamente as semelhanças entre ambas é significativa e programadores de C++ se sentem à vontade programando em Java. Outro ponto: Java não foi projetada para substituir C++. Ela foi projetada para resolver um certo conjunto de problemas. C++ foi projetada para resolver outro grupo diferente de problemas. Ambas coexistirão por muitos anos.

Como mencionado no início deste capítulo, linguagens de computador evoluem por duas razões: para se adaptar às mudanças no ambiente e para implementar avanços na arte da programação. A mudança ambiental que levou à Java foi a necessidade de programas independentes de plataforma destinados a distribuição na Internet. No entanto, Java também incorpora mudanças na maneira como as pessoas abordam a criação de programas. Por exemplo, Java herdou e refinou o paradigma da orientação a objetos usada por C++, adicionando suporte integrado para multiprocessamento, e forneceu uma biblioteca que simplificou o acesso à Internet. Em análise final, porém, não foram os recursos individuais de Java que a tornaram tão notável. Pelo contrário, foi a linguagem como um todo. Java foi a resposta perfeita para as demandas do universo computacional recém-emergente e altamente distribuído.

Java era para programar para a Internet, e C era para programar em sistema: uma força revolucionária que mudou o mundo.

A conexão com C#

O alcance e o poder de Java continuam sendo sentidos no mundo do desenvolvimento da linguagem de computador. Muitos de seus recursos, construções e conceitos inovadores tornam-se parte da linha de base de qualquer nova linguagem. O sucesso de Java é simplesmente muito importante para ser ignorado.

Talvez o mais importante exemplo da influência de Java seja C# (lê-se “C sharp”). Criado pela Microsoft para dar suporte a .NET Framework, C# é intimamente relacionada à Java. Por exemplo, ambas compartilham a mesma sintaxe geral, suportam programação distribuída, e utilizam o mesmo modelo de objeto. Há, obviamente, diferenças entre Java e C#, mas a aparência dessas duas linguagens é muito similar. Essa “troca” de Java para C# é o testemunho mais forte de que Java redefiniu a maneira como a linguagem de computador é pensada e usada.

Como Java Impactou a Internet

A Internet ajudou a catapultar Java para a frente da programação, e Java, por sua vez, teve um profundo efeito na internet. Além de simplificar a programação da web em geral, Java inovou cirando um novo tipo de programa em rede chamado applet, que mudou a maneira como o mundo online pensava sobre o conteúdo.

A linguagem Java também abordou alguns dos problemas mais difíceis associados à internet: a vulnerabilidade e a segurança. Um olhar mais atento é dado a cada um deles a seguir.

Java Applets

No momento da criação de Java, um dos mais interessantes recursos foi o applet. *Applet* é um tipo especial de programa Java projetado para ser transmitido para a Internet e ser executado automaticamente dentro de um navegador web compatível com Java. Se o usuário clicar em um link que contenha um applet, esse será baixado e executado pelo navegador. Applets são destinados a ser pequenos programas. Eles são tipicamente usados para mostrar dados fornecidos pelo servidor, manipular a entrada do usuário, ou fornecer funções simples, tal como uma calculadora de empréstimos, que executa localmente, e não no servidor. Em essência, applets permitem que algumas funcionalidades sejam movidas do servidor para o cliente.

A criação do applet foi importante porque, na época, ela expandiu o universo de objetos que podia se mover livremente no espaço cibernético. Em geral, há duas categorias muito amplas de objetos que são transmitidos entre o servidor e o cliente: informações passivas e programas ativos e dinâmicos. Por exemplo, um e-mail é um agrupamento de dados passivos. Mesmo quando um programa é baixado, o código do programa ainda é apenas dados passivos até que seja executado. Por outro lado, o applet é um programa dinâmico autoexecutável. Esse programa é ativado no computador do cliente, mas é iniciado pelo servidor.

Nos primeiros dias de Java, os aplets eram uma parte crucial da programação. Eles ilustravam o poder e os benefícios de Java, adicionaram uma dimensão interessante para

as páginas web, e permitiram que os programadores explorassem toda a extensão que era possível com Java. Embora seja provável que ainda existam applets em uso hoje, com o passar do tempo eles se tornaram menos importantes. Por razões que serão explicadas, a partir da versão JDK 9 começou o suporte a applets, sendo o mesmo removido pela versão JDK 11.

Segurança

Por mais desejáveis que sejam os programas dinâmicos em rede, eles também podem apresentar sérios problemas nas áreas da segurança e portabilidade. Obviamente, um programa que é baixado e executado no computador do cliente deve ser impedido de causar danos. Ele também deve ser capaz de executar em uma variedade de diferentes ambientes e sistemas operacionais. Java resolveu esses problemas de maneira eficaz e elegante.

A cada programa “normal” que é baixado, o usuário assume um risco, porque o código baixado pode conter um vírus, Cavalo de Troia, ou outro código nocivo. No centro do problema está o fato de que um código malicioso pode causar danos porque obteve acesso não autorizado aos recursos do sistema. Por exemplo, um vírus pode coletar informações privadas, tal como números de cartão de crédito, saldos da conta bancária, e senhas, pesquisando por conteúdo do sistema no computador local. Para que Java permitisse que os programas fossem baixados e executados com segurança no computador do cliente, era necessário impedi-los de iniciar um ataque desse tipo. Java alcançou essa proteção, permitindo ao usuário confinar uma aplicação ao ambiente de execução Java e impedindo-o de acessar outras partes do computador.

A capacidade de baixar programas com um grau de confiança de que nenhum dano será causado pode ter sido o aspecto mais inovador de Java.

Portabilidade

A portabilidade é o aspecto mais importante da Internet porque existem muitos tipos diferentes de computadores e sistemas operacionais conectados a ela. Se um programa Java fosse executado virtualmente em qualquer computador conectado à Internet, seria necessário que houvesse algum modo de permitir que esse programa fosse executado em sistemas diferentes. Em outras palavras, seria necessário um mecanismo que permitisse que uma mesma aplicação fosse baixada e executada por uma ampla variedade de CPUs, sistemas operacionais, e navegadores. Não é nada prático ter diferentes versões do aplicativo para diferentes computadores. O mesmo código de computador deveria funcionar em todos os tipos de computadores. Portanto, seria necessário algum meio de gerar um

código executável portátil. Em Java, um mesmo mecanismo que ajuda a garantir a segurança também ajuda a criar portabilidade.

A Mágica de Java: Bytecode

A chave que permitiu a Java resolver ambos os problemas: segurança e portabilidade, é que a saída de um compilador Java não é um código executável. Pelo contrário, é um bytecode. *Bytecode* é um conjunto de instruções altamente otimizado e projetado para ser executado pela *Máquina Virtual Java (JVM)*, que faz parte do Ambiente de Execução Java (JRE). Em essência, a JVM original foi projetada como um *interprete de bytecode*. Isso pode ser uma surpresa, já que muitas linguagens são projetadas para serem compiladas e executadas devido a problemas de desempenho. No entanto, o fato de um programa Java ser executado pela JVM ajuda a resolver os principais problemas associados aos programas baseados na web. Aqui está o porquê.

A conversão de um programa Java em bytecode facilita muito a execução de um programa em uma ampla variedade de ambientes porque apenas a JVM precisa ser implementada para cada plataforma. Uma vez que uma JRE existe para um determinado sistema, qualquer programa Java pode ser executado nele. Lembre-se, embora os detalhes da JVM sejam diferentes de plataforma para plataforma, todas entendem o mesmo bytecode Java. Se um programa Java fosse compilado para um código nativo, então existiriam diferentes versões do mesmo programa para cada tipo de CPU conectada à Internet. Essa, claramente, não é uma solução viável. Portanto, a execução do bytecode pela JVM é a maneira mais fácil de criar programas realmente portáteis.

O fato de um programa Java ser executado pela JVM também ajuda a torna-lo seguro. Porque a JVM está no controle, ela gerencia a execução do programa. Assim, é possível que a JVM crie um ambiente de execução restrito, chamado *sandbox*, o qual contém o programa, impedindo o acesso irrestrito à máquina. A segurança é também aprimorada por certas restrições que existem na linguagem Java.

Na verdade, quando um programa é compilado para um formato intermediário e então interpretado para uma máquina virtual, ele é executado mais devagar do que se fosse compilado em código executável. No entanto, com Java, a diferença entre ambos os métodos não é tão grande. Porque o bytecode foi altamente otimizado, o uso do bytecode permite à JVM a execução de programas mais rápido do que o esperado.

Embora Java tenha sido projetada como uma linguagem interpretada, não há nada sobre Java que impeça a compilação imediata do bytecode em código nativo para aumentar o desempenho. Por esse motivo, a tecnologia HotSpot foi introduzida pouco depois do lançamento inicial de Java. HotSpot fornece uma compilação Just-In-Time (JIT) – na hora

certa. Como um compilador JIT faz parte de JVM, partes selecionadas de bytecode são compiladas em código executável em tempo real, parte por parte, com base na demanda. É importante entender que um programa Java não é compilado no código executável todo de uma só vez. Em vez disso, nem todas as sequências de bytecode são compiladas – somente aquelas que se beneficiarão da compilação. O código restante é simplesmente interpretado. No entanto, a abordagem JIT ainda gera um aumento significativo no desempenho. Mesmo quando a compilação dinâmica é aplicada ao bytecode, os recursos de portabilidade e de segurança ainda se aplicam, porque a JVM ainda é responsável pelo ambiente de execução.

Outro ponto: a partir da JDK 9, alguns ambientes Java também dão suporte a um compilador *antecipado* que pode ser usado para compilar bytecode em código nativo *antes* da execução pela JVM, e não em tempo real. A compilação antecipada é um recurso especializado e não substitui a abordagem tradicional de Java. Devido à natureza altamente especializada da compilação antecipada, ela não será mais discutida nesse livro.

Indo Além dos Applets

Quando a linguagem Java foi criada, a Internet era uma inovação emocionante; os navegadores estavam passando por desenvolvimento e refinamento muito rápidos; a forma moderna dos smartphones ainda não havia sido inventada; e ainda faltavam alguns anos para o uso quase onipotente dos computadores. Java também mudou e, com ela, a forma como era usada. Talvez nada ilustre a evolução continua de Java melhor que o applet.

Nos primeiros anos de Java, applets eram uma parte crucial da programação em Java. Eles não apenas adicionavam entusiasmo para as páginas web, mas também eram uma parte altamente visível de Java, o que aumentou seu carisma. No entanto, applets dependiam de um plug-in Java de navegador. Portanto, para que um applet funcionasse, o navegador deveria dar suporte. Recentemente, o suporte dos navegadores para esse plug-in Java foi diminuindo. Simplificando, sem o suporte dos navegadores, applets não são viáveis. Por esse motivo, a partir da JDK 9, a eliminação parcial dos applets foi iniciada, com o suporte aos applets caindo. Na linguagem Java, *descontinuado* significa que ainda está disponível, mas sinalizado como obsoleto. Portanto, um recurso obsoleto não deve ser usado para novos códigos. A eliminação gradual foi completada com a JDK 11 porque o suporte para applets foi removido.

Poucos anos após a criação de Java uma alternativa aos applets foi adicionada ao Java. Chamada Web Start, permitiu que uma aplicação fosse baixada dinamicamente da página web. Era um mecanismo de implantação especialmente útil para aplicativos Java que não eram apropriados para applets. A diferença entre applet e um aplicativo Web Start

é que um aplicativo Web Start é executado sozinho, não em um navegador. Assim, parece muito como um aplicativo “normal”. No entanto, requer um JRE independente que suporte Web Start esteja disponível no sistema host. A partir da versão JDK 11, o suporte para Web Start foi removido.

Visto que nem applet nem Web Start são suportados pelas versões modernas de Java, a solução é usar a ferramenta **jlink** adicionada pela versão JDK 9. Ela pode criar uma imagem em tempo de execução completa que inclui todo o suporte necessário para os programas, incluindo a JRE. Embora uma discussão detalhada das estratégias de implementação esteja fora do escopo deste livro, é algo que se deve prestar muita atenção ao avançar.

Um Cronograma de Lançamento Mais Rápido

Outa mudança importante ocorreu recentemente em Java, mas não envolve mudanças feitas na linguagem, ou no ambiente de execução. Em vez disso, está relacionada à maneira como os lançamentos de Java são planejados. No passado, lançamentos importantes eram tipicamente separados por dois anos ou mais. Em vez disso, após o lançamento da JDK 9, o tempo entre os principais lançamentos foi reduzido. Hoje, prevê-se que um lançamento importante ocorra dentro de um cronograma estrito, com o tempo esperado entre esses lançamentos sendo de apenas seis meses.

Lançamentos a cada seis meses, agora chamados de *lançamento de recursos*, incluirão os recursos prontos no momento do lançamento. Essa *cadência de lançamento* aumentada agora permite que novos recursos e aprimoramentos estejam disponíveis para os programadores Java em tempo hábil. Além disso, permite que Java responda rapidamente à demanda de um ambiente de programação em constante mudança. Simplificando, o cronograma de lançamento rápido promete ser um desenvolvimento muito positivo para os programadores Java.

Atualmente, os lançamentos de recursos estão programados para março e setembro de cada ano. Como resultado, JDK 10 foi lançada em março de 2018, seis meses após o lançamento da versão JDK 9. E o lançamento da versão JDK 11 foi em setembro de 2018.

Servlets: Java no Lado do Servidor

O Código do lado do cliente é apenas metade da equação cliente/servidor. Pouco tempo após o lançamento de Java, ficou óbvio que Java também seria útil do lado do servidor. O resultado foi o *servlet*. Um servlet é um programa pequeno que é executado no servidor.

Servlets são usados para criar conteúdo gerado dinamicamente que é veiculado no cliente. Por exemplo, uma loja online pode usar um servlet para procurar o preço de um item em um banco de dados. As informações de preço são usadas para gerar dinamicamente uma página web que é enviada ao navegador. Embora o conteúdo gerado dinamicamente estivesse disponível por meio de mecanismos como CGI (Common Gateway Interface) – Interface de Gateway Comum -, o servlet oferecia várias vantagens, incluindo desempenho aprimorado.

Como servlets são compilados em bytecode e executados pela JVM, eles são altamente portáteis. Portanto, um mesmo servlet pode usar uma variedade de ambientes de servidor diferentes. Os únicos requisitos são: o servidor que dê suporte a JVM e um contêiner de servlet. Hoje, o código do lado do servidor em geral constitui um uso importante de Java.

As Palavras-Chave de Java

Nenhuma discussão sobre a história de Java é completa sem uma olhada nas palavras-chave. Embora as forças fundamentais que exigiram a invenção de Java sejam portabilidade e segurança, outros fatores também tiveram um papel importante na moldagem da forma final da linguagem. As principais considerações foram resumidas pela equipe Java na seguinte lista de palavras chave:

- Simple – simples;
- Secure – segura;
- Portable – portátil;
- Object-oriented – orientada a objeto;
- Robust – robusta;
- Multithreaded – multiprocessamento;
- Architecture-neutral – arquitetura neutra;
- Interpreted – interpretada;
- High performance – alta performance;
- Distributed – distribuída;
- Dynamic – dinâmica.

Duas das principais palavras chave já foram discutidas: segurança e portabilidade.

Simples

Java foi projetada para facilitar o aprendizado e o uso efetivo do programador profissional. Programadores têm pouca dificuldade de dominar a linguagem Java, conhecendo os conceitos básicos da programação orientada a objetos o aprendizado se torna ainda

mais fácil. Como Java herda a sintaxe de C/C++ e muito dos recursos orientados a objetos de C++, a maioria dos programadores têm poucos problemas em aprender Java.

Orientação a Objetos

Embora influenciada por seus antecessores, Java não é projetada para ser compatível com qualquer outra linguagem. Isso permitiu à equipe Java a liberdade para projetar com uma folha em branco. Um resultado disso foi a abordagem limpa, utilizável, e pragmática para objetos. Emprestando liberalmente de muitos ambientes de software de objeto inspiradores das últimas décadas, Java consegue encontrar um equilíbrio entre o paradigma do purista “tudo é um objeto” e o modelo do pragmatista “fique fora do meu caminho”. O modelo de objeto em Java é simples e fácil de estender, enquanto tipos primitivos, como números inteiros, são mantidos como objetos não de alto desempenho.

Robusta

O ambiente multiplataforma da web exige demandas extraordinárias em um programa, porque o programa deve ser executado de maneira confiável em vários sistemas. Assim a capacidade de criar programas robustos recebeu alta prioridade na projeção de Java. Para ganhar confiabilidade, Java restringe o programador a algumas áreas importantes para força-lo a encontrar seus erros no início do desenvolvimento do programa. Ao mesmo tempo, Java evita que o programador se preocupe com muitas das causas mais comuns de erros de programação. Como Java é uma linguagem estritamente tipificada, ela verifica o código no momento da compilação. No entanto, ela também verifica o código em tempo de execução. Muitos erros difíceis de rastrear que geralmente aparecem em situações de tempo de execução difíceis de reproduzir são simplesmente impossíveis de criar em Java. Saber que o que foi escrito se comportará de maneira previsível sob diversas condições é um recurso essencial de Java.

Para entender melhor como a linguagem Java é robusta, considere duas das principais razões para a falha de um programa: os erros de gerenciamento de memória e as condições excepcionais manipuladas incorretamente (erros em tempo de execução). O gerenciamento de memória pode ser difícil e tedioso nos ambientes de programação tradicionais. Por exemplo, em C/C++, o programador deve alocar e liberar manualmente a memória dinâmica para o sistema. Às vezes, isso leva a problemas, porque os programadores esquecem de liberar memória alocada anteriormente ou, pior ainda, tentam liberar alguma memória que outra parte do código ainda esteja usando. Java praticamente elimina esses problemas, gerenciando a alocação de memória e desalocando memória (de fato, desalocando memória de modo totalmente automático, porque Java fornece coleta de lixo para objetos não utilizados). Condições excepcionais em ambientes tradicionais geralmente

surgem em situações como “divisão por zero” ou “arquivo não encontrado”, e devem ser gerenciadas de modo difícil e são construções difíceis para ler e/ou entender. Java ajuda nesta área fornecendo manipulação de exceção orientada a objetos. Em um programa Java bem escrito, todos os erros em tempo de execução podem – e devem – ser gerenciados pelo programa.

Multiprocessamento

A linguagem Java foi projetada para atender aos requisitos do mundo real da criação de programas interativos e que funcionam em rede. Para isso, Java dá suporte à programação de multiprocessamento, a qual permite a codificação de programas que fazem muitas coisas simultaneamente. O sistema de tempo de execução Java é fornecido com uma solução sofisticada para sincronização de múltiplos processos, que permite a construção de sistemas interativos sem problemas. A abordagem fácil de usar de Java para multiprocessamento permite que se preveja o comportamento específico do programa, não no subsistema multitarefa.

Arquitetura Neutra

Um problema central na projeção de Java era a longevidade e a portabilidade do código. No momento da criação de Java, um dos problemas principais enfrentados pelos programadores era que não existia garantia de que se você escrevesse um programa hoje, ele poderia ser executado corretamente amanhã – mesmo que fosse na mesma máquina. Atualizações nos sistemas operacionais, atualizações no processador, e alterações nos principais recursos do sistema podiam ser combinados para causar um mal funcionamento do programa. Os designers de Java tomaram várias decisões difíceis na linguagem Java e na JVM na tentativa de alterar essa situação. O objetivo era “escrever uma vez; executar em qualquer lugar, a qualquer hora, para sempre”. Em grande parte, esse objetivo foi alcançado.

Interpretada e de Alto Desempenho

Como descrito anteriormente, Java permite a criação de programas de plataforma cruzada ao compilar em uma representação intermediária chamada bytecode Java. Esse código pode ser executado em qualquer sistema que implemente a JVM. A maioria das tentativas anteriores de soluções de plataforma cruzada o fez à custa de desempenho. Conforme explicado anteriormente, o bytecode de Java foi cuidadosamente projetado para que fosse fácil converter diretamente em código de máquina nativo para obter um desempenho muito alto usando um compilador JIT (Just-In-Time). Os sistemas em tempo de execução que forneciam esses recursos não perdem nenhum dos benefícios do código independente de plataforma.

Distribuída

Java foi projetada para o ambiente distribuído da Internet porque lida com protocolos TCP/IP. De fato, acessar um recurso usando uma URL não é muito diferente de acessar um arquivo por se tratar de um endereçamento. Java também suporta RMI (Remote Method Invocation) – Método Remoto de Invocação. Esse recurso possibilita que um programa invoque métodos em uma rede.

Dinâmica

Os programas Java carregam com eles quantidades de informações do tipo tempo de execução que são usadas para verificar e resolver acessos a objetos em tempo de execução. Isso torna possível vincular dinamicamente o código de maneira segura e conveniente. Isso é crucial para a robustez do ambiente Java, no qual pequenos fragmentos de bytecode podem ser atualizados dinamicamente em um sistema em execução.

A Evolução de Java

O lançamento inicial de Java não foi nada menos que revolucionário, mas não marcou o fim da era de sua rápida inovação. Diferentemente da maioria dos outros sistemas de software que geralmente se estabelecem em um padrão de pequenas melhorias incrementais, Java continua a evoluir em um ritmo explosivo. Logo após o lançamento da versão 1.0 de Java, os designers de Java já haviam criado a versão 1.1. Os recursos adicionados foram mais significativos e substanciais do que se pode imaginar. Muitos elementos foram adicionados para a biblioteca, foi redefinida a maneira como os eventos eram manipulados, e foram reconfigurados muitos recursos da biblioteca 1.0. Também foram reprovados muitos recursos originalmente definidos. Assim, Java 1.1 adicionou e subtraiu dos atributos de sua especificação original.

O próximo grande lançamento de Java foi Java 2, onde o “2” indica “segunda geração”. A criação de Java 2 foi um evento divisor de águas, marcando o início da “idade moderna” de Java. A primeira versão de Java 2 carregava o número da versão 1.2. Pode parecer estranho que a versão 2 de Java tenha usado o número de versão 1.2. O motivo é que ele se referia originalmente ao número da versão interna das bibliotecas Java, mas depois foi generalizado para se referir a todos os lançamentos. Com Java 2, a Sun reembalou o produto Java como J2SE (Java 2 Plataform Standard Edition) – Segunda Edição da Plataforma Padrão Java –, e os números da versão começaram a ser aplicados a esse produto.

O lançamento de Java 2 adicionou suporte a vários novos recursos, como Swing e Collections Framework, e aprimorou a JVM e várias ferramentas de programação. Algumas

substituições foram feitas. A classe **Thread** foi a mais afetada, na qual os métodos **suspend()**, **resume()**, e **stop()** foram desprezados.

A versão J2SE 1.3 foi a primeira grande atualização para a Java 2 original. Na maior parte, adicionou e reforçou o ambiente de desenvolvimento para funcionalidades ainda existentes. Em geral os programas escritos para a versão 1.2 e para a versão 1.3 são compatíveis com o código fonte. Embora a versão 1.3 contivesse um conjunto menor de alterações que as três principais versões anteriores, ela era importante.

O lançamento da versão J2SE 1.4 aprimorou ainda mais a linguagem Java. Esta versão continha várias atualizações, aprimoramentos, e adições importantes. Por exemplo, ela adicionou novas palavras-chave como ***assert*** (definições), *chained exceptions* (exceções encadeadas), e *channel-based I/O subsystem* (subsistema de I/O baseado em canal). Ela também fez alterações para Collections Framework e nas classes de rede. Além disso, inúmeras pequenas alterações foram feitas em todo o processo. Apesar do número significante de novos recursos, a versão 1.4 mantinha quase 100% de compatibilidade com o código fonte das versões anteriores.

O próximo lançamento de Java foi J2SE 5, e foi revolucionário. Diferentemente da maioria das atualizações anteriores, que ofereciam melhorias importantes, mas lentas, J2SE 5 expandiu fundamentalmente o escopo, o poder, e o alcance da linguagem. Para entender a magnitude das mudanças que J2SE 5 fez ao Java, considere a seguinte lista de seus principais recursos:

- Generics – genéricos;
- Annotations – anotações;
- Autoboxing and auto-unboxing – caixa e desempacotamento automático;
- Enumerations – enumerações;
- Enhanced, for-each style for loop – alterações, para cada estilo do loop **for**;
- Variable-length arguments (varargs) – argumentos de tamanho variável;
- Static import – importação estática;
- Formatted I/O – entrada e saída formatada;
- Concurrency utilities – utilitários de simultaneidade.

Esta não é uma lista de pequenos ajustes ou atualizações incrementais. Cada item na lista representa uma adição significativa para a linguagem Java. Alguns, como genéricos, enhanced **for**, e varargs, introduzem uma nova sintaxe de elementos. Outros, como autoboxing e auto-unboxing, alteraram a semântica da linguagem. Annotations adicionou uma dimensão totalmente nova à programação. Em todos os casos, o impacto dessas adições foi além de seus efeitos diretos. Eles mudaram o próprio caráter de Java.

A importância desses novos recursos é refletida no uso do número “5”. O próximo número de versão seria 1.5. No entanto, os novos recursos foram tão significantes que 1.5 não expressaria a magnitude dessa mudança. Em vez disso, a Sun decidiu aumentar o número da versão para 5 como forma de enfatizar que um grande evento estava ocorrendo. Assim, foi nomeado J2SE 5, e o kit do desenvolvedor foi chamado de JDK 5. No entanto para manter a consistência, a Sun decidiu usar 1.5 como número da versão interna, também conhecido como número de *versão do desenvolvedor*. O “5” no J2SE 5 é chamado de número da *versão do produto*.

O próximo lançamento de Java foi chamado de Java SE 6. A Sun novamente decidiu por alterar o nome da plataforma Java. Primeiro, observe que o número “2” foi descartado. Portanto, a plataforma agora chamava-se Java SE, e o nome oficial do produto era Java Platform, Standard Edition 6. O Kit de Desenvolvimento Java era chamado de JDK 6. Assim como no J2SE 5, o 6 no Java SE 6 é o número da versão do produto. O da versão interna do desenvolvedor era 1.6.

Java SE 6 construiu na base de J2SE 5, adicionando melhorias incrementais. A Java SE 6 não adicionou nenhum recurso importante à linhagem Java propriamente dita, mas aprimorou as bibliotecas API, adicionou vários novos pacotes, e ofereceu melhorias no tempo de execução. Ela também passou por várias atualizações durante seu longo ciclo de vida, com várias atualizações adicionadas ao longo do caminho. Em geral, Java SE 6 serviu para solidificar ainda mais os avanços feitos pela J2SE 5.

Java SE 7 foi o próximo lançamento de Java, com o Kit de Desenvolvimento Java chamado de JDK 7, e o número da versão interna de 1.7. Java SE 7 foi o primeiro grande lançamento de Java desde que a Sun Microsystems foi adquirida pela Oracle. Java SE 7 continha muitos recursos novos, incluindo adições significativas à linguagem e à biblioteca API. Também foram incluídas atualizações para o sistema de execução em tempo real de Java que dava suporte à algumas linguagens, mas são as adições de bibliotecas e linguagens que mais interessam aos programadores Java.

Os novos recursos da linguagem foram desenvolvidos como parte do *Project Coin*. O objetivo desse projeto era identificar uma série de pequenas mudanças na linguagem Java que seriam incorporadas a JDK 7. Embora esses recursos sejam referidos como “pequenos”, os efeitos dessas alterações foram bastante grandes em termos de código que eles impactam. De fato, para muitos programadores, essas mudanças podem muito bem ter sido os novos recursos mais importantes na Java SE 7. Aqui há uma lista dos recursos de linguagem adicionados pela JDK 7:

- Uma **String** agora pode controlar uma instrução **switch**;

- Inteiros binários literais;
- Underlines em numéricos literais;
- Uma instrução **try** expandida, chamada ***try****-with-resources*, que suporta o gerenciamento automático de recursos (Por exemplo, os fluxos podem ser fechados automaticamente quando não são mais necessários);
- Inferência de tipo (através do operador *Diamond*) ao construir uma instancia genérica;
- Tratamento aprimorado de exceções, no qual duas ou mais exceções podem ser capturadas por uma única **catch** (multi-catch) e melhor verificação de tipo para exceções que são mostradas novamente;
- Embora não seja uma alteração de sintaxe, os avisos do compilador associados a alguns tipos de métodos varargs foram aprimorados, e tem-se mais controle sobre os avisos.

Mesmo que os recursos do Project Coin fossem considerados pequenas alterações na linguagem, seus benefícios eram muito maiores do que o qualificador “pequeno” sugeriria. Em particular, a instrução ***try****-with-resources* afetou profundamente a maneira como o código baseado em fluxo era gravado. Além disso, a capacidade de usar uma **String** para controlar uma instrução **switch** foi uma melhoria muito desejada, que simplificou a codificação em muitas situações.

Java SE 7 fez várias adições à biblioteca API de Java. Dois dos mais importantes foram: os aprimoramentos no NIO Framework e a adição de Fork/Join Framework. NIO (que originalmente significava *Nova I/O*) foi adicionada à Java na versão 1.4. No entanto, as alterações adicionadas pela Java SE 7 expandiram fundamentalmente seus recursos. As mudanças foram tão significativas, que o termo *NIO.2* é frequentemente usado.

A Fork/Join Framework fornece suporte importante para *programação paralela*. Programação paralela é o nome comumente dado às técnicas que fazem uso efetivo de computadores que contêm mais de um processador, incluindo sistemas multicore. A vantagem que os ambientes multicore oferecem é a perspectiva de aumentar significativamente o desempenho do programa. A Fork/Join Framework abordou a programação paralela por:

- Simplificar a criação e o uso de tarefas que podem ser executadas simultaneamente;
- Utilizar automaticamente vários processadores.

Portanto, usando Fork/Join Framework, pode-se criar facilmente aplicações escaláveis que tiram vantagem automaticamente dos processadores disponíveis no ambiente de execução. Obviamente, nem todos os algoritmos se prestam à paralelização, mas para

aqueles que o fazem, uma melhoria significativa na velocidade de execução pode ser obtida.

A próxima versão de Java foi Java SE 8, com o Kit do Desenvolvedor chamado de JDK 8. Ela teve o número de versão interna de 1.8. A JDK 8 foi uma atualização significativa para a linguagem Java devido à inclusão de um novo recurso de linguagem de longo alcance: a *expressão lambda*. O impacto das expressões lambda foi e continuará sendo profundo, mudando a maneira como as soluções de programação são conceituadas e como o código Java é escrito. Como explicado em detalhes no Capítulo 15, expressões lambda adicionaram recursos de programação funcionais à Java. No processo, expressões lambda podem simplificar e reduzir a quantidade de código fonte necessária para criar certas construções, como alguns tipos de classes anônimas. A adição de expressões lambda também fez com que um novo operador (->) e um novo elemento de sintaxe fossem adicionados para a linguagem.

A inclusão de expresses lambda também tiveram um efeito abrangente nas bibliotecas de Java, com novos recursos sendo adicionados para tirar proveito deles. Um dos mais importantes foi a nova API stream, que é empacotada na **java.util.stream**. A API stream suporta operações de pipeline em dados e é otimizada para expressões lambda. Outro novo pacote foi o **java.util.function**. Ele define várias *interfaces funcionais*, que fornecem suporte adicional para expressões lambda. Outros novos recursos relacionados a lambda podem ser encontrados em todas as bibliotecas API da linguagem.

Outro recurso inspirado em lambda afeta a **interface**. A partir da JDK 8 se tornou possível definir uma implementação padrão para um método específico por uma interface. Se nenhuma implementação padrão para um método for criada, o padrão definido pela interface é usado. Esse recurso permite que as interfaces evoluam normalmente ao longo do tempo, porque um novo método pode ser adicionado para uma interface sem quebrar o código existente. Ele também pode simplificar a implementação de uma interface quando os padrões forem apropriados. Outros novos recursos na JDK 8 incluem uma nova API de tempo e data, tipos de anotações, e a capacidade de usar processamento paralelo ao classificar um vetor, entre outros.

A próxima versão de Java foi a Java SE 9. O Kit de Desenvolvedor foi chamado de JDK 9. Com o lançamento da JDK 9, o número de versão interna também é 9. JDK 9 representa o maior lançamento, incorporando aprimoramentos significantes para a linguagem Java e suas bibliotecas. Como as versões JDK 5 e JDK 8, JDK9 afetou a linguagem Java e as bibliotecas API de maneiras fundamentais.

O primeiro novo recurso da JDK 9 foi *módulos*, que permitem a especificação de relações e dependências de código que compreende um aplicativo. Módulos também adicionam outra dimensão para recursos de controle de acesso em Java. A inclusão de módulos fez com que um novo elemento de sintaxe e várias palavras chave fossem adicionadas para Java. Além disso, uma ferramenta chamada **jlink** foi adicionada à JDK, que permite ao programador criar uma imagem em tempo de execução da aplicação que contêm apenas os módulos necessários. Um novo tipo de arquivo, chamado JMOD, foi criado. Módulos também afetam profundamente a biblioteca API porque, começando com a JDK 9, os pacotes da biblioteca agora estão organizados em módulos. Embora os módulos constituam uma grande nova melhoria de Java, eles são conceitualmente simples e diretos. Além disso, como o código legado "pré-modulo" é totalmente suportado, módulos podem ser integrados ao processo de desenvolvimento em sua linha do tempo. Não há necessidade de alterar imediatamente nenhum código pré-existente para manipular os módulos. Em resumo, os módulos adicionaram funcionalidades substanciais sem alterar a essência de Java.

Além de módulos, a JDK incluiu muitos outros recursos. Um em particular foi JShell, que é uma ferramenta que suporta a experimentação e o aprendizado interativos de programas (Uma introdução a JShell é encontrada no Apêndice B). Outra atualização interessante é o suporte aos métodos de interface privada. Sua inclusão aprimorou ainda mais o suporte de JDK 8 para métodos padrão em interfaces. JDK 9 adicionou um recurso de pesquisa à ferramenta **javadoc** e uma nova tag chamada **@index** para suportá-la. Como nas versões anteriores, a JDK 9 continha um número de aprimoramentos nas bibliotecas API de Java.

Como regra geral, em qualquer versão de Java, são os novos recursos que recebem mais atenção. No entanto, havia um aspecto de alto perfil de Java que foi preferido pela JDK 9: applets. A partir da JDK 9, os applets não eram mais recomendados para novos projetos. Conforme explicado anteriormente neste capítulo, devido ao declínio do suporte dos navegadores aos applets (e por outros fatores), a JDK 9 reprovou toda a API applet.

A próxima versão de Java foi a Java SE 10 (JDK 10). Como explicado anteriormente, a partir da JDK 10, espera-se que os lançamentos ocorram em um cronograma estrito baseado em tempo, com o tempo entre os principais lançamentos previstos para apenas seis meses. Como resultado, a JDK 10 foi lançada em março de 2018, que foi lançada seis meses depois de JDK 9. O novo recurso principal da linguagem adicionado pela JDK 10 foi o suporte para a *inferência de tipo de variável local*. Com a inferência de tipo de variável local, agora é possível permitir que o tipo de uma variável local seja inferido do tipo de seu

inicializador, em vez de ser especificado explicitamente. Para suportar esse novo recurso, o identificador sensível ao contexto **var** foi adicionado à Java como um nome de tipo reservado. A inferência de tipo pode simplificar o código eliminando a necessidade de especificar redundantemente o tipo de uma variável quando ela puder ser deduzida do inicializador. Também pode simplificar as declarações nos casos em que o tipo é difícil de discernir ou não pode ser especificado explicitamente. A inferência de tipo de variável local tornou-se uma parte comum do ambiente de programação contemporâneo. Sua inclusão em Java ajuda a mantê-la atualizada com as tendências em evolução no design da linguagem. Juntamente com várias outras mudanças, a JDK 10 também redefiniu a sequência de versões de Java, alterando o significado dos números de versão para que eles se alinhem melhor com o novo cronograma de lançamento baseado em tempo.

A versão Java SE 11 (JDK 11) foi lançada em setembro de 2018, seis meses após a JDK 10. O novo recurso principal da linguagem em JDK 11 é o suporte ao uso de **var** em expressões lambda. Juntamente com vários ajustes e atualizações na API em geral, JDK 11 adicionou uma nova API de rede, que será de interesse para uma ampla gama de desenvolvedores. Chamada de HTTP Client API, é empacotada em java.net.http, e fornece suporte aprimorado e atualizado à rede para clientes HTTP. Além disso, outro modo de execução foi adicionado ao iniciador Java que permite executar diretamente programas simples de arquivo único. JDK 11 também removeu alguns recursos. Talvez a mais interessante por causa de seu significado histórico foi a remoção do suporte aos applets. Lembre-se que applets foram desprezados pela primeira vez pela JDK 9. Com a versão JDK 11, o suporte aos applets foi removido. O suporte para outra tecnologia relacionada à implementação chamada Java Web Start também foi removido da JDK 11. Como o ambiente de execução continuou a evoluir, ambos estavam rapidamente perdendo a relevância. Outra mudança importante na JDK 11 é que JavaFX não está mais incluído na JDK. Em vez disso, essa estrutura GUI se tornou um projeto de código aberto separado. Como esses recursos não fazem mais parte da JDK, eles não serão discutidos neste livro.

Outro ponto sobre a evolução de Java: A partir de 2006, o processo do código aberto Java começou. Hoje, as implementações de código aberto da JDK estão disponíveis. O código aberto contribui ainda mais para a natureza dinâmica do desenvolvimento Java. Em análise final, o legado de inovação de Java está seguro, Java continua sendo a linguagem vibrante e ágil que o mundo da programação espera.

O material deste livro foi atualizado para a JDK 11. Muitos recursos novos, atualizados, e adições de Java são descritos por toda parte. Como a discussão anterior destacou, no entanto, o histórico da programação Java é marcado por mudanças dinâmicas.

A Cultura da Inovação

Desde o início, Java está no centro de uma cultura da inovação. Seu lançamento original redefiniu a programação para a Internet. A JVM e o bytecode mudaram a maneira como a segurança e a portabilidade são pensadas. O código portátil deu vida à Web. O JCP (Java Community Process) – Processo Comunitário Java -, redefiniu a maneira como as novas ideias são assimiladas para a linguagem.

CAPÍTULO II Uma Visão Geral de Java

Como em todas as outras linguagens de computador, os elementos de Java não existem em isolamento. Em vez disso, eles trabalham juntos para formar a linguagem como um todo. No entanto, essa interrelação pode dificultar a descrição de um aspecto de Java sem envolver vários outros. Frequentemente uma discussão de um recurso implica conhecimento prévio de outro. Por esse motivo, esse capítulo apresenta uma visão geral rápida de muitos recursos chave de Java. O material descrito aqui dá uma base para escrever e entender programas simples. A maioria dos tópicos discutidos serão examinados em mais detalhes nos demais capítulos da Parte I.

Programação Orientada a Objeto

A Programação Orientada a Objetos (OOP) está no centro de Java. De fato, todos os programas são pelo menos em certa medida orientados a objetos. OOP é tão essencial para Java que é melhor entender seus princípios básicos antes de começar a escrever programas simples em Java. Portanto, este capítulo começa com uma discussão dos aspectos teóricos de OOP.

Dois Paradigmas

Todos os programas de computador consistem de dois elementos: código e dados. Além disso, um programa pode ser conceitualmente organizado em torno do código ou dos dados. Ou seja, alguns programas são escritos em torno do que “está acontecendo” e outros são escritos em torno de quem “está sendo afetado”. Esses são os dois paradigmas que governam a construção de um programa. A primeira maneira é chamada de *modelo orientado ao processo*. Essa abordagem caracteriza um programa como uma série de etapas lineares (que é o código). O modelo orientado ao processo pode ser pensado como *código atuando nos dados*. Linguagens metodológicas como C empregam esse modelo para um sucesso considerável. No entanto, como mencionado no Capítulo 1, problemas

com essa abordagem aparecem à medida que os programas se tornam maiores e mais complexos.

Para gerenciar a complexidade crescente a segunda abordagem, chamada de *programação orientada a objetos*, foi concebida. A programação orientada a objetos organiza um programa em torno dos dados (que são os objetos) e um conjunto de interfaces bem definidas para esses dados. Um programa orientado a objetos pode ser caracterizado como *dados que controlam o acesso ao código*. Mudando a entidade que controla os dados, pode-se obter vários benefícios organizacionais.

Abstração

Um elemento essencial da programação orientada a objetos é a *abstração*. Os seres humanos gerenciam a complexidade através da abstração. Por exemplo, as pessoas não pensam nos carros como o conjunto das dezenas de milhares de peças individuais. Elas pensam neles como objetos bem definidos com seus próprios comportamentos únicos. Essa abstração permite que as pessoas usem um carro para dirigir ao supermercado sem ficarem sobrecarregadas pela complexidade das peças individuais. Elas podem ignorar os detalhes de como o motor, a transmissão e os sistemas de freio funcionam. Assim, elas são livres para usar o objeto como um todo.

Uma maneira poderosa de gerenciar abstrações é através do uso de classificações hierárquicas. Isso permite que a semântica seja disposta em camadas de sistemas complexos, dividindo-as em partes mais gerenciáveis. De fora, o carro é um único objeto. Uma vez lá dentro, o carro consiste de vários subsistemas: direção, freios, sistema de som, cinto de segurança, aquecimento, e assim por diante. Por sua vez, cada um desses subsistemas é composto por unidades mais especializadas. Por exemplo, o sistema de som pode consistir de um rádio, CD player, e/ou MP3 player. O ponto é que a complexidade dos sistemas é gerenciada através do sistema de abstrações hierárquicas.

Abstrações hierárquicas de sistemas complexos podem também ser aplicadas para programas de computador. Os dados de um programa tradicional orientado ao processo podem ser transformados pela abstração em objetos componentes. Uma sequência de etapas do processo pode se tornar uma coleção de mensagens entre esses objetos. Assim, cada um desses objetos descreve seu próprio comportamento único. Pode-se tratar esses objetos como entidades concretas que respondem a mensagens solicitando que elas *façam alguma coisa*. Essa é a essência da programação orientada a objetos.

Conceitos de orientação a objetos formam o coração de Java, assim como formam a base da compreensão humana. A programação orientada a objetos é um paradigma poderoso e natural para criar programas que sobrevivem às mudanças inevitáveis que

acompanham o ciclo de vida de qualquer grande projeto de software, incluindo concepção, crescimento e envelhecimento. Por exemplo, depois de ter objetos bem definidos e interfaces limpas e confiáveis para esses objetos, pode-se decompô-los em partes de um sistema antigo sem medo.

Os Três princípios da OOP

Todas as linguagens de programação orientadas a objetos fornecem mecanismos que ajudam na implementação de modelos orientados a objetos. São eles: encapsulamento, herança e polimorfismo.

- Encapsulamento

O *encapsulamento* é o mecanismo que une o código e os dados que ele manipula e mantém ambos protegidos contra inferências externas e de uso indevido. Uma maneira de pensar sobre encapsulamento é como um invólucro protetivo que previne o código e os dados de serem acessados arbitrariamente por outro código definido fora do invólucro. O acesso ao código e aos dados dentro do invólucro é totalmente controlado por uma interface bem definida. Para relacionar isso ao mundo real, considere a transmissão automática de um carro. Ela contém centenas de bits de informação sobre o motor, como o quanto se acelera, a inclinação da superfície, e a posição da alavanca de câmbio. O usuário tem apenas um método de afetar esse complexo encapsulamento: movendo a alavanca de câmbio. O usuário não pode afetar a transmissão usando o pisca alerta ou os limpadores de vidro, por exemplo. Assim, o câmbio de mudança de marchas é uma interface bem definida (de fato, única). Além disso, o que ocorre dentro da transmissão não afeta os objetos de fora dela. Por exemplo, a mudança de marchas não liga as luzes! Como a transmissão automática é encapsulada, dezenas de fabricantes de carros podem implementar uma da maneira que desejarem. No entanto, do ponto de vista do motorista, todas funcionam do mesmo jeito. Essa mesma ideia pode ser aplicada para a programação. O poder do código encapsulado é que todos sabem como acessá-lo e podem usá-lo independentemente dos detalhes da implementação – e sem medo de efeitos colaterais inesperados.

Em Java, a base do encapsulamento é a classe. Embora a classe seja examinada em detalhes mais adiante neste livro, a breve discussão a seguir será útil agora. Uma classe define a estrutura e o comportamento (dos dados e do código) que serão moldados por um conjunto de objetos. Cada objeto de uma determinada classe contém a estrutura e o comportamento definidos pela classe como se fossem estampadas por um molde na forma da classe. Por esse motivo, os objetos são as vezes referidos como *instâncias de uma classe*. Assim, uma classe é uma construção lógica: um objeto tem realidade física.

Ao criar uma classe, o código e os dados que constituem essa classe são especificados. Coletivamente, esses elementos são chamados de *membros* da classe.

Especificamente, os dados definidos pela classe são referidos como *variáveis de membro* ou *variáveis de instância*. Os códigos que operam com esses dados são referidos como *métodos membro* ou apenas *métodos* (para programadores familiarizados com C/C++, ajuda saber que o que um programador Java chama de *método*, um programador C/C++ chama de *função*). Em programas escritos corretamente em Java, os métodos definem como as variáveis membro pode ser usadas. Isso significa que o comportamento e a interface da classe são definidos por métodos que operam em seus dados de importância.

Como o objetivo de uma classe é encapsular a complexidade, há mecanismos para esconder a complexidade da implementação dentro da classe. Cada método ou variável em uma classe é marcado como privada ou pública. A interface *public* de uma classe representa tudo o que os usuários externos de uma classe precisam ou podem saber. Os métodos e os dados *private* podem ser acessados apenas pelo código que é membro da classe. Portanto, qualquer outro código que não seja membro da classe não pode acessar um método ou variável privada. Como os membros privados da classe podem ser acessados apenas por outras partes do programa através de métodos públicos da classe, pode-se garantir que nenhuma ação imprópria ocorra. Obviamente, isso significa que uma interface pública deve ser cuidadosamente projetada para não expor muito dos trabalhos internos da classe. É como se as variáveis de instância pública, e os métodos públicos fossem alocados na parte mais externa da classe, e as variáveis de instância privada e os métodos privados fossem alocados na parte mais interna da classe.

- Herança

Herança é o processo pelo qual um objeto adquire as propriedades de outro. Isso é importante porque suporta o conceito de classificação hierárquica. Como mencionado anteriormente, a maior parte do conhecimento é gerenciável por classificações hierárquicas (de cima para baixo). Por exemplo, um Golden Retriever é classificado como cachorro, que por sua vez é parte da classe *mamíferos*, pertencente à classe de *animais* maiores. Sem o uso de hierarquias, cada objeto precisaria definir todas as suas características explicitamente. No entanto, ao usar hierarquia, um objeto precisa apenas definir as qualidades que o tornam único dentro de sua classe. Ele pode herdar seus atributos de seu pai. Portanto, é o mecanismo de herança que torna possível que um objeto seja uma instância especifica de um caso mais geral.

A maioria das pessoas vêm o mundo mais naturalmente como composto de objetos que são relacionados entre si de maneira hierárquica, tal como animais, mamíferos e

cachorros. Para descrever animais de modo abstrato, deve-se dizer que eles têm alguns atributos, tal como tamanho, inteligência e tipo de sistema de esqueleto. Animais também têm certos aspectos comportamentais: eles comem, respiram e dormem. Essa descrição de atributos e comportamentos é a definição de classe para animais.

Para descrever uma classe de animais mais específica, como mamíferos, ela deve ter atributos mais específicos, como o tipo de dentes e glândulas mamárias. Isso é conhecido como uma *subclasse* de animais, onde os animais são chamados de *superclasse* dos mamíferos.

Como os mamíferos são animais simplesmente especificados com mais precisão, eles *herdam* todos os atributos dos animais. Uma subclasse profundamente herdada herda todos os atributos de cada antecessor da *hierarquia* de classes.

A herança interage bem com o encapsulamento. Se uma determinada classe encapsular alguns atributos, qualquer subclasse terá os mesmos atributos, *mais* os que adicionar como parte de sua especialização. Esse é um conceito chave que permite aos programas orientados a objetos aumentar a complexidade linearmente e não geometricamente. Uma nova subclasse herda todos os atributos dos seus antecessores. Não possui interações imprevisíveis com a maioria do código restante no sistema.

- Polimorfismo

Polimorfismo (do Grego, que significa “muitas formas”) é um recurso que permite que a interface seja usada por ações de uma classe geral. A ação específica é determinada pela natureza da situação. Considerando uma pilha (uma lista de último a entrar, primeiro a sair). Deve-se ter um programa que exija três tipos de pilhas. Uma pilha é usada para valores inteiros, uma para valores de ponto-flutuante, e outra para caracteres. O algoritmo que implementa cada pilha é o mesmo, independentemente do tipo de dados. Em linguagens não orientadas a objetos, deve-se criar três conjuntos diferentes de rotinas de pilha. Com cada conjunto usando nomes diferentes. No entanto, devido ao polimorfismo, em Java, pode-se especificar um conjunto geral de rotinas que compartilham o mesmo nome.

De maneira mais geral, o conceito de polimorfismo é frequentemente expressado pela frase “uma interface, múltiplos métodos”. Isso significa que é possível projetar uma interface genérica para um grupo de atividades relacionadas. Isso ajuda a reduzir a complexidade, permitindo que a mesma interface use uma *ação especifica da classe geral*. O trabalho do compilador é selecionar uma *ação específica* (ou seja, método) que se aplica a cada situação. O programador não precisa fazer essa seleção manualmente. Precisa apenas lembrar e utilizar a interface geral.

Estendendo a analogia do cão, o olfato de um cachorro é polimórfico. Se o cachorro farejar um gato, ele latirá e correrá atrás dele. Se o cachorro farejar comida, ele salivará e correrá para a tigela. O mesmo olfato está funcionando em ambas as situações. A diferença está no cheiro que o cachorro está sentindo, ou seja, “o tipo de dado que está sendo operado pelo nariz do cachorro!”. Esse mesmo conceito geral pode ser implementado em Java pois se aplica a métodos dentro de um programa Java.

- Polimorfismo, Encapsulamento, e Herança Trabalham Juntos

Quando aplicados adequadamente, polimorfismo, encapsulamento e herança combinam-se para produzir um ambiente de programação que suporta o desenvolvimento de programas mais robustos e escalonáveis do que o modelo orientado ao processo. Uma hierarquia de classes bem projetada é a base para reutilizar o código no qual se investiu tempo e esforço para desenvolver e testar. O encapsulamento permite a migração de implementações ao longo do tempo sem quebrar o código que depende da interface pública de suas classes. O polimorfismo permite a criação de um código limpo, sensível, legível e flexível.

Dos dois exemplos do mundo real, o automóvel ilustra mais completamente o poder do design orientado a objetos. É divertido pensar em cães do ponto de vista da herança, mas carros são mais parecidos com programas.

Todos os motoristas dependem da herança para conduzir diferentes tipos (subclasses) de veículos. Independentemente de ser um ônibus escolar, um Mercedes sedan, um Porsche, ou uma minivan da família, motoristas podem encontrar e operar o volante, os freios e o acelerador. Depois de um pouco de esforço, a maioria das pessoas pode lidar com a diferença entre uma troca de marchas de um câmbio manual e um automático, porque eles basicamente têm uma superclasse comum, a transmissão.

As pessoas interagem com os recursos encapsulados em carros o tempo todo. Os pedais de freio e acelerador escondem uma incrível variedade de complexidade com uma interface tão simples que pode ser operada com os pés! A implementação do motor, o estilo dos freios e o tamanho dos pneus não afetam a maneira como se interage com a definição de classe dos pedais.

O atributo final, o polimorfismo, reflete-se claramente na capacidade dos fabricantes de automóveis de oferecer uma ampla gama de opções basicamente do mesmo veículo. Por exemplo, pode-se obter um sistema de freios antibloqueio ou tradicionais, direção hidráulica ou mecânica, e motores de 4, 6 ou 8 cilindros. De qualquer forma, o pedal do freio ainda é pressionado para parar, o volante ainda é girado para mudar de direção e o

acelerador ainda é pressionado para se mover. A mesma interface pode ser usada para controlar um número de diferentes implementações.

É através da aplicação de encapsulamento, herança e polimorfismo que as partes individuais são transformadas no objeto conhecido como carro. O mesmo se aplica aos programas de computador. Pela aplicação de princípios orientados a objetos, as várias partes de um programa complexo podem ser reunidas para formar um todo coeso, robusto e sustentável.

Como mencionado no início desta seção, todos programas Java são orientados a objetos. Ou, mais precisamente, todo programa Java envolve encapsulamento, herança e polimorfismo. Embora os pequenos exemplos de programas mostrados no restante deste capítulo e nos próximos capítulos pareçam não exibir todos esses recursos, eles ainda estão presentes. Muitos dos recursos fornecidos pela linguagem Java fazem parte de suas bibliotecas de classes internas, que fazem uso extensivo de encapsulamento, herança e polimorfismo.

Um Primeiro Exemplo Simples

Agora que foi discutido o básico da orientação a objetos de Java, pode-se olhar para alguns programas Java reais. Um pequeno programa de exemplo pode ser compilado e executado aqui. Isso envolve um pouco mais de trabalho do que se pode imaginar.

```
/*
        Este é um simples programa Java.
        Este arquivo é nomeado de “Exemplo.java”.
*/
class Exemplo
        {
                // O programa começa com uma chamada para “main()”.
                public static void main(String args[])
                        {
                                System.out.println(“Este é um simples programa Java”);
                        }
        }
```

Nota: As descrições a seguir usam o padrão Java SE Development Kit (JDK), disponibilizado pela Oracle (A versão em código aberto também está disponível). Para usar um ambiente de desenvolvimento integrado (IDE) é preciso seguir um procedimento diferente para compilar e executar programas Java. Nesse caso, é necessário consultar a documentação do IDE para obter mais detalhes.

Entrando no Programa

Para a maioria das linguagens de computador, o nome do arquivo que contém o código fonte para um programa é irrelevante. No entanto, este não é o caso de Java. A primeira coisa que se deve aprender sobre Java é que o nome dado ao arquivo de origem é muito importante. Neste exemplo, o nome do arquivo de origem deve ser **Exemplo.java**.

Em Java, um arquivo de origem é oficialmente chamado de *unidade de compilação*. É um arquivo de texto que contém (entre outras coisas) uma ou mais definições de classe (Por enquanto, pode-se usar arquivos de origem que contém apenas uma classe). O compilador Java requer que um arquivo de origem use a extensão de arquivo **.java**.

O nome da classe definida pelo programa é **Exemplo**. Isto não é uma coincidência. Em Java, todo o código deve residir dentro de uma classe. Por convenção, o nome da classe principal deve corresponder ao nome do arquivo que contém o programa. Deve-se ter certeza que a capitalização do nome do arquivo corresponda ao nome da classe. A razão para isso é que Java é caso sensitivo (letras maiúsculas e minúsculas se diferem). Nesse ponto, a convenção de que os nomes dos arquivos correspondam aos nomes das classes pode parecer arbitrária. No entanto, esta convenção facilita a manutenção e a organização dos programas. Além disso, em alguns casos, isso é requerido.

Compilando o Programa

Para compilar o programa **Exemplo**, deve-se executar o compilador **javac**, especificando o nome do arquivo de origem na linha de comando, como mostrado a seguir:

```
C:\>javac Example.java
```

O compilador **javac** cria um arquivo chamado **Exemplo.class** que contém uma versão bytecode do programa. Como discutido anteriormente, o bytecode de Java é uma representação intermediária do programa que contém instruções que a Máquina Virtual Java executará. Portanto, a saída do **javac** não é um código que possa ser executado diretamente.

Para realmente executar um programa, deve-se usar o inicializador de aplicação Java chamado **java**. Para tanto, passa-se o nome da classe **Exemplo** como um argumento da linha de comando, como é mostrado a seguir:

```
C:\>java Example
```

Quando o programa é executado, a saída a seguir é mostrada:

```
Este é um simples programa Java
```

Quando o código fonte Java é compilado, cada classe individual é colocada em seu próprio arquivo de saída nomeado após a classe e usando a extensão **.class**. Por isso é uma boa ideia dar aos arquivos de origem os mesmos nomes das classes contidas neles –

o nome do arquivo de origem corresponderá ao nome do arquivo **.class**. Quando se executa **java** como no exemplo mostrado, especifica-se realmente o nome da classe que se quer executar. Ela pesquisa automaticamente por um arquivo pelo nome que tenha a extensão **.class**. Se encontrar o arquivo, ela executará o código contido na classe especificada.

Nota: A partir da JDK 11, Java fornece uma maneira para executar alguns tipos de programas simples diretamente do arquivo de origem, sem invocar explicitamente **javac**. Essa técnica, que pode ser útil em algumas situações, é descrita no Apêndice C. Para os propósitos deste livro, supõe-se que será usado o processo de compilação já descrito.

Um Olhar Mais Atento ao Primeiro Programa Simples

Embora o programa **Exemplo.java** seja bastante curto, ele inclui muitos recursos chave que são comuns a todos os programas Java.

O programa começa com as seguintes linhas:

```
/*
    Este é um simples programa Java.
    Este arquivo é nomeado de "Exemplo.java".
*/
```

Esse é um *comentário*. Como em outras linguagens de programação, Java permite que uma observação seja inserida no arquivo de origem do programa. O conteúdo do comentário é ignorado pelo compilador. Um comentário descreve ou explica a operação do programa para quem está lendo o código fonte. Nesse caso, o comentário descreve o programa e relembra que o arquivo de origem deve se chamar **Exemplo.java**. É claro, nas aplicações reais, os comentários geralmente explicam como algumas partes do programa funcionam ou o que um recurso especifico faz.

Java suporta três estilos de comentários. O comentário mostrado na parte superior do programa é chamado de *comentário multilinha*. Esse tipo de comentário deve começar com "/*" e terminar com "*/". Qualquer coisa entre esses dois símbolos de comentários é ignorada pelo compilador. Como o nome sugere, um comentário multilinha pode ter várias linhas.

A próxima linha do código no programa é mostrada aqui:

```
class Exemplo {
```

Esta linha usa a palavra-chave **class** para declarar que uma nova classe está sendo definida. **Exemplo** é um *identificador* que é o nome da classe. A definição da classe, incluindo todos os membros, é feita entre as chaves de abertura ({) e de fechamento (}). Por hora, não é preciso se preocupar muito com os detalhes de uma classe, exceto para

observar que em Java, todas as atividades do programa ocorrem em uma. Esse é o motivo pelo qual todos os programas Java são orientados a objetos.

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

```
// O programa começa com uma chamada para "main()".
```

Este é o segundo tipo de comentário suportado por Java. Um comentário *de linha simples* começa com "//" e termina no final da linha. Como regra geral, programadores usam comentários multilinhas para observações mais longas e comentários de linha única para descrições breves, linha por linha. O terceiro tipo de comentário, um *comentário da documentação*, será discutido na seção "Comentários", posteriormente neste capítulo.

A próxima linha do código é mostrada aqui:

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

Esta linha inicia o método **main()**. Como sugere o comentário anterior, esta é a linha na qual o programa começará a ser executado. Como regra geral, um programa Java começa a execução chamando **main()**. O significado completo de cada parte desta linha não pode ser dado agora, porque envolve um entendimento detalhado da abordagem de encapsulamento de Java. No entanto, como a maioria dos exemplos na primeira parte deste livro usará essa linha de código, será dada uma breve olhada em cada parte agora.

A palavra-chave **public** é um *modificador de acesso*, que permite ao programador controlar a visibilidade dos membros da classe. Quando um membro da classe é precedido por **public**, esse membro pode ser acessado pelo código de fora da classe em que é declarado (o oposto de **public** é **private**, o que impede que um membro seja usado pelo código definido fora da sua classe). Nesse caso, **main()** deve ser declarado como **public**, pois deve ser chamado pelo código de fora da classe quando o programa é iniciado. A palavra-chave **static** permite que **main()** seja chamado sem precisar instanciar uma instância específica da classe. Isso é necessário pois **main()** é chamado pela JVM antes que qualquer objeto seja criado. A palavra-chave **void** simplesmente diz ao compilador que **main()** não retorna um valor. Métodos podem também retornar valores. Todos esses conceitos serão discutidos em detalhes nos capítulos seguintes.

Como afirmado, quando o método **main()** é chamado uma aplicação Java inicia. É importante entender que o compilador Java compilará classes que não contenham um método **main()**. Mas não tem como executar essas classes.

Qualquer informação que precise ser passada para o método é recebida por variáveis especificadas no conjunto de parênteses que seguem o nome do método. Essas variáveis são chamadas de *parâmetros*. Se não houver parâmetros necessários para um determinado método, ainda será necessário incluir os parâmetros vazios. Em **main()**, há

apenas um parâmetro, embora seja um complicado. **String args[]** declara um parâmetro chamado **args**, que é uma matriz de instâncias da classe **String** (*Matrizes* são coleções de objetos similares).

Objetos do tipo String armazenam cadeias de caracteres. Nesse caso, args recebe quaisquer argumentos da linha de comando presente quando o programa é executado. Este programa não utiliza essas informações, mas outros programas mostrados mais adiante neste livro o farão.

O símbolo "{", na próxima linha, sinaliza o início do corpo de **main()**. Todo o código que compreende um método ocorrerá entre a chave de abertura do método e a sua chave de fechamento.

Um outro ponto: **main()** é um simples ponto de partida para o programa. Um programa complexo terá dezenas de classes, sendo que apenas uma delas precisará ter o método **main()** para iniciar as coisas. Além disso, para alguns tipos de programas, não será necessário ter **main()**. No entanto, para a maioria dos programas mostrados neste livro, **main()** é necessário.

A próxima linha do código é mostrada aqui. Isso ocorre dentro de **main()**.

```
System.out.println("Este é um simples programa Java");
```

Essa linha gera a sequência "Este é um simples programa Java". Seguido por uma nova linha na tela. A saída é realmente realizada pelo método interno **println()**. Nesse caso, **println()** exibe a string que lhe é passada. **Println()** pode ser usado para exibir outros tipos de informação também. A linha começa com **System.out**. Embora seja muito complicado explicar em detalhes neste momento, brevemente, **System** é uma classe predefinida que fornece acesso ao sistema, e **out** é o fluxo de saída que é conectado ao console.

A saída (e a entrada) do console não é usada frequentemente na maioria das aplicações Java do mundo real. Como a maioria dos ambientes de computação modernos são de natureza gráfica, a I/O do console é usada principalmente para programas utilitários, programas de demonstração e código do lado do servidor. Mais adiante neste livro, serão mostradas outras maneiras de gerar saída usando Java. Mas por enquanto, os métodos de I/O continuaram sendo utilizados.

Note que a instrução **println()** termina com um ponto e vírgula. Muitas instruções em Java terminam com ponto e vírgula, o que o torna uma parte importante da sintaxe Java.

O primeiro símbolo "}" no programa finaliza o método **main()**, e o último "}" finaliza a definição da classe **Exemplo**.

Um Segundo Programa Curto

Talvez nenhum outro conceito seja mais fundamental para uma linguagem de programação do que o de variável. Uma variável é um local da memória nomeado ao qual o programa atribui um valor. O valor da variável pode ser alterado durante a execução do programa. O próximo programa mostra como uma variável é declarada e como lhe é atribuído um valor. O programa também ilustra alguns novos aspectos da saída do console. Como os comentários na parte superior do estado do programa mostram, esse segundo programa será chamado de **Exemplo2.Java**.

```
/*
        Aqui está outro exemplo curto.
        Esse programa é nomeado "Exemplo2.java".
*/

class Exemplo
        {
        public static void main (String args [])
                {
                        int num;
                        // esta declara uma variável chamada "num"

                        num=100;
                        // esta atribui o valor 100 à "num"

                        System.out.println("Este é o num: " + num);

                        num=num*2;

                        System.out.println("O valor de num * 2 é ");
                        System.out.println(num);
                }
        }
```

Quando executado, este programa mostrará a seguinte saída:

```
Este é o num: 100
O valor de num * 2 é 200
```

A primeira linha do programa é mostrada aqui:

```
                        int num;
```

Esta linha declara uma variável do tipo inteiro chamada **num**. Java (como a maioria das outras linguagens) requer que as variáveis sejam declaradas antes de serem usadas. A seguir está a forma geral da declaração de uma variável:

tipo nome-da-variável;

Aqui, *tipo* especifica o tipo de variável que está sendo declarado, e *nome-da-variável* é o nome que um local da memória recebe. Para declarar mais de uma variável do mesmo tipo especificado, pode-se usar uma lista separando os nomes das variáveis por vírgula. Java define muitos tipos de dados, incluindo inteiros, caracteres e ponto-flutuante. A palavra-chave **int** especifica um tipo inteiro. No programa, a linha:

num=100;

atribui a **num** o valor 100. Em Java, o operador de atribuição é um único sinal de igual.

A próxima linha da saída do código gera o valor de **num** precedido pela string “Este é num:”.

The next line of code outputs the value of num preceded by the string "This is num:".

System.out.println(“Este é o num: “ + num);

Nesta declaração, o sinal de mais faz com que o valor de **num** seja anexado à cadeia de caracteres que a precede e, em seguida, a cadeia de caracteres resultante é emitida (na verdade, **num** é primeiramente convertida de um número inteiro em seu equivalente de string e, em seguida, concatenado com a string que lhe precede. Esse processo é declarado em detalhes mais adiante neste livro). Essa abordagem pode ser generalizada. Usando o operador **+**, pode-se juntar quantos itens forem necessários com uma única instrução **println()**.

A próxima linha de código atribui a **num** o valor de **num** duas vezes. Como na maioria das outras linguagens, Java usa o operador ***** para indicar a multiplicação. Após esta linha ser executada, **num** conterá o valor 200.

Aqui estão as duas próximas linhas no programa:

*System.out.println(“O valor de num * 2 é 200”);*

System.out.println(num);

Várias coisas novas estão ocorrendo aqui. Primeiro, o método **print()** é usado para exibir a string “O valor de num * 2 é “. Esta string não é seguida por uma nova linha. Isso significa que, quando a próxima saída for gerada, ela começara na mesma linha. O método **print()** é como **println()**, exceto que ele não gera um caractere de nova linha após cada chamada. Ambos os métodos podem ser usados para gerar valores de qualquer um dos tipos internos de Java.

Duas Declarações de Controle

Embora o Capítulo 5 analise atentamente as declarações de controle, duas são brevemente apresentadas aqui para que possam ser usadas em programas de exemplo no Capítulo 3 e no Capítulo 4. Elas também ajudarão a ilustrar um aspecto importante de Java: blocos de código.

A Instrução if

A instrução **if** em Java funciona de maneira semelhante à instrução if em qualquer outra linguagem. Ela determina o fluxo de execução com base em alguma condição ser verdadeira ou falsa. Sua forma mais simples é mostrada aqui:

```
if(condição) declaração;
```

Aqui, condição é uma expressão booleana. (Uma expressão booleana é aquela que avalia como verdadeira ou falsa). Se a condição for verdadeira, a declaração será executada. Se a condição for falsa, a declaração será ignorada. Aqui está um exemplo:

```
if (num<100) System.out.println("num é menor que 100");
```

Nesse caso, se **num** contiver um valor menor que 100, a expressão condicional será verdadeira, e **println()** será executado. Se **num** contiver um valor maior ou igual a 100, então o método **println()** será ignorado.

Como discutido no Capítulo 4, Java define um complemento de operadores relacionais que podem ser usados em uma expressão condicional. Aqui estão alguns:

- < - menor que;
- > - maior que;
- == - igual a.

Aqui está um programa que ilustra a declaração if:

```
/*
        Demonstração de if.
        Este programa é nomeado "IfSimples.java".
*/
class IfSample
        {
                public static void main(String args[])
                        {
                                int x, y;

                                x=10;
                                y=20;
```

```
                    if(x<y)
                        System.out.println("x é menor que y");
                        x=x*2;
                    if(x==y)
                        System.out.println("x agora é igual a y");
                        x=x*2;
                    if(x>y)
                        System.out.println("x agora é maior que y");
                    // Isso não exibirá nada
                    If(x==y)
                        System.out.ptintln("Você não vê isto");
                }
        }
```

A saída gerada por esse programa é mostrada aqui:

x é menor que y
x agora é igual a y
x agora é maior que y

Observe uma outra coisa nesse programa. A linha:

```
int x, y;
```

declara duas variáveis **x** e **y**, usando uma lista separada por virgula.

O Loop for

As instruções **for** são uma parte importante de praticamente qualquer linguagem de programação porque elas fornecem uma maneira de executar repetidamente alguma tarefa. Como detalhado no Capítulo 5, Java fornece uma variedade poderosa de construções de loop. Talvez o loop mais versátil seja **for**. A forma mais simples do loop **for** é mostrada aqui:

```
for(inicialização; condição; iteração) declaração;
```

Na forma mais comum, a parte de *inicialização* do loop define um controle do loop variável para um valor inicial. A *condição* é uma expressão booleana que testa a variável de controle do loop. Se o resultado desse teste for verdadeiro, a *declaração* será executada e o loop **for** continuará a iterar. Se o resultado for falso, o loop termina. A expressão de *iteração* determina como a variável de controle do loop é alterada cada vez que o loop itera. Aqui está um pequeno programa que ilustra o loop **for**:

```
/*
```

```
        Demonstração do loop for.
        Este programa é nomeado "TesteFor.java".
*/
class TesteFor
        {
                public static void main(String args[])
                        {
                                int x;

                                for(x=0; x<10; x=x+1)
                                        System.out.println("Esse é x: " + x);
                        }
        }
```

Esse programa gera a seguinte saída:

```
Esse é x: 0
Esse é x: 1
Esse é x: 2
Esse é x: 3
Esse é x: 4
Esse é x: 5
Esse é x: 6
Esse é x: 7
Esse é x: 8
Esse é x: 9
```

Neste exemplo, **x** é a variável de controle do loop. Ela é inicializada com zero na parte da inicialização do **for**. No início de cada iteração (incluindo a primeira), o teste condicional **x**<10 é executado. Se o resultado desse teste for verdadeiro, a instrução **println()** é executada, o que aumentará **x** em 1. Esse processo continua até que o teste condicional seja falso.

Como ponto de interesse, em programas Java profissionalmente escritos, quase nunca se vê a parte da iteração do loop gravada como mostrado no programa anterior. Ou seja, raramente se vê declarações como esta:

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

O motivo é que 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 consecutivos). O operador de incremento aumenta seu operando em um. Com o uso do operador de incremento, a declaração anterior pode ser escrita assim:

```
x++;
```

Assim, o **for** no programa anterior geralmente será escrito assim:

```
for(x=0; x<10; x++);
```

O loop ainda é executado exatamente da mesma maneira que antes.

Java também fornece um operador de decremento, que é especificado como --. Este operador diminui seu operando em um.

Usando Blocos de Código

Java permite que duas ou mais instruções sejam agrupadas em *blocos de código*. Isso é feito encerrando as instruções entre abrir e fechar chaves. Depois que um bloco de código é criado, ele se torna uma unidade lógica que pode ser usada em qualquer lugar que uma instrução possa. Por exemplo, um bloco pode ser apontado para as declarações **if** e **for**. Considerando esta instrução **if**:

```
if(x<y)
        {
                // Começo de bloco
                x = y;
                y = 0;
        }
        // Final do bloco
```

Aqui, se **x** for menor que **y**, as duas instruções dentro do bloco serão executadas. Assim, as duas instruções dentro do bloco formam uma unidade lógica, e uma instrução não pode ser executada sem que a outra também seja executada. O ponto principal aqui é que sempre que preciso vincular logicamente duas ou mais instruções, isso pode ser feito criando um bloco.

O programa a seguir usa um boco de código como destino de um loop **for**.

```
/*
        Demonstração de um bloco de código.
        Este arquivo se chama “BlockTest.java”
*/
class BlockTest
        {
                public static void main(String arg[])
                        {
```

```
                int x, y;

                y=20;

                // o destino desse loop é um bloco
                for(x=0; x<10;x++)
                        {
                                System.out.println("Este é x: " + x);
                                System.out.println("Este é y: " + y);
                                y=y-2;
                        }
        }
}
```

A saída mostrada por esse programa é mostrada aqui:

Este é x: 0
Este é y: 20
Este é x: 1
Este é y: 18
Este é x: 2
Este é y: 16
Este é x: 3
Este é y: 14
Este é x: 4
Este é y: 12
Este é x: 5
Este é y: 10
Este é x: 6
Este é y: 8
Este é x: 7
Este é y: 6
Este é x: 8
Este é y: 4
Este é x: 9
Este é y: 2

Neste caso, o alvo do loop **for** é um bloco de código e não apenas uma única declaração. Assim, cada vez que o loop itera, as três instruções dentro do bloco são executadas.

Esse fato é evidenciado pela saída gerada pelo programa.

Como detalhado mais adiante neste livro, blocos de código têm propriedades e usos adicionais. No entanto, a principal razão de sua existência é criar unidades lógicas inseparáveis.

Questões Lexicais

Agora que foram mostrados vários programas curtos, é hora de descrever mais formalmente os elementos atômicos de Java. Os programas Java são uma coleção de espaços em branco, identificadores, literais, comentários, operadores, separadores e palavras-chave. Os operadores serão descritos no próximo capítulo. Os demais itens dessa lista serão descritos a seguir.

Espaço em branco

Java é uma linguagem de forma livre. Isso significa que você não precisa seguir nenhuma regra de endentação especial. Por exemplo, o programa **Exemplo** poderia ter sido escrito em uma linha ou de qualquer outra maneira, desde que houvesse pelo menos um caractere de espaço em branco separando tudo aquilo que não estivesse delineado por um operador ou separador. Em Java, espaço em branco inclui um espaço, tabulação, nova linha ou feed de formulário.

Identificadores

Identificadores são usados para nomear coisas, tal como classes, variáveis e métodos. Um identificador pode ser qualquer sequência descritiva de letras maiúsculas e minúsculas, números, ou os caracteres underline e cifrão. (O caractere de cifrão não é destinado ao uso geral). Eles não devem começar com um número, para que não sejam confundidos com uma literal numérica. Novamente, Java é caso-sensitivo, então **VALUE** é um identificador diferente de **Value**. Alguns exemplos de identificadores validos são: AvgTemp, count, a4, $test, this_is_ok. Identificadores inválidos se parecem com: 2count, high-temp, Not/ok.

Nota: A partir da JDK 9, o underline não pode ser usado por si só como um identificador.

Literais

Um valor constante em Java é criado usando uma representação *literal*. Por exemplo, aqui estão alguns literais: 100, 98.6, ‘X’, “This is a test”.

Da esquerda para a direita, o primeiro literal especifica um inteiro, o próximo é um valor de ponto-flutuante, o terceiro é uma constante de caractere e o último é uma string. Um literal pode ser usado em qualquer lugar em que um valor desse tipo seja permitido.

Comentários

Como mencionado, existem três tipos de comentários definidos por Java: linha única (//), multilinha (/* */) e da documentação (esse tipo de comentário é usado para produzir um arquivo HTML que documenta seu programa. O *comentário da documentação* começa com "/**" e termina com "*/". O comentário da documentação é explicado no Apêndice A.

Separadores

Em Java, há poucos caracteres que são usados como separadores. O separador mais comumente usado em Java é o ponto e vírgula. Como já mencionado, é frequentemente usado para finalizar instruções. Os separadores são mostrados na tabela a seguir:

Símbolo	Nome	Propósito
()	Parênteses	Usados para conter uma lista de parâmetros na definição e na invocação de método. Também usados para definir precedências em expressões e conter expressões em instruções de controle.
{}	Chaves	Usadas para conter valores de matrizes inicializadas automaticamente. Também usadas para definir blocos de código, classes, métodos e escopos locais.
[]	Colchetes	Usados para declarar tipos de matrizes. Também usados para diferenciar valores da matriz.
;	Ponto e vírgula	Termina instruções.
,	Vírgula	Separa identificadores consecutivos em uma declaração de variáveis. Também usada para encadear instruções de um loop **for.**
.	Ponto	Usado para separar nomes de pacotes de pacotes internos e classes. Também usado para separar uma variável ou método de uma variável de referência.
::	Dois pontos	Usados para criar um método ou construtor de referência.
...	Elipse	Indica um número variável de parâmetros.
@	Arroba	Começa uma anotação.

As Palavras-Chave de Java

Atualmente existem 61 palavras chave definidas na linguagem Java. Essas palavras-chave, combinadas com a sintaxe dos operadores e separadores, formam a base da linguagem Java. Em geral, essas palavras-chave não podem ser usadas como identificadores, o que significa que não podem ser usadas como nomes para uma variável, classe ou

método. As exceções a esta regra são as palavras-chave caso-sensitivas adicionadas pela JDK 9 para suportar módulos (Consulte o Capítulo 16 para obter detalhes). Além disso, começando com a JDK 9, um sublinhado por si só considerando uma palavra-chave para impedir seu uso como o nome de algo no programa.

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

As palavras-chave **const** e **goto** são reservadas, mas não são usadas. Nos primeiros dias de Java, várias outras palavras-chave foram reservadas para possível uso no futuro. No entanto, a especificação atual para Java define apenas as palavras-chave mostradas na tabela acima.

Além das palavras-chave, Java reserva outros quatro nomes. Três fizeram parte de Java desde o início: **true**, **false** e **null**. Esses são os valores definidos por Java. Não se pode usar essas palavras para os nomes de variáveis, classes e assim por diante. A partir da JDK 10, a palavra **var** foi adicionada como um nome de tipo reservado e sensível ao contexto. (Veja o Capítulo 3 para mais detalhes sobre **var**).

As Bibliotecas de Classes de Java

Os programas mostrados neste capítulo usam dois dos métodos internos de Java: **println()** e **print()**. Como mencionado, esses métodos estão disponíveis na classe **System.out.System** predefinida por Java que é automaticamente incluída nos seus programas. Na visão ampliada, o ambiente Java conta com várias bibliotecas de classes internas que contêm muitos métodos internos que fornecem suporte para itens como I/O, manipulação de strings, redes e gráficos. As classes padrão também fornecem suporte para uma interface gráfica do usuário (GUI). Assim, Java como um todo é uma combinação da própria linguagem Java, além de suas classes padrão. As bibliotecas de classes fornecem grande parte da funcionalidade que vêm com Java. De fato, parte de se tornar um programador

Java é aprender a usar as classes padrão de Java. Na Parte I deste livro, vários elementos das classes e métodos padrão são descritos conforme necessário. Na Parte II, várias bibliotecas de classes são descritas em detalhes.

CAPÍTULO III

Tipos de Dados, Variáveis e Matrizes

Este capítulo examina três dos elementos mais fundamentais de Java: tipos de dados, variáveis e matrizes. Como todas as linguagens de programação modernas, Java suporta muitos tipos de dados. Pode-se usar esses tipos para declarar variáveis e para criar matrizes. A abordagem de Java para esses itens é limpa, eficiente e coesa.

Java É Uma Linguagem Fortemente Tipificada

É importante afirmar desde o início que Java é uma linguagem fortemente tipificada. De fato, parte da segurança e robustez de Java vem desse fato. Primeiro, toda variável tem um tipo, toda expressão tem um tipo e todo tipo é estritamente definido. Segundo, todas as atribuições, sejam explicitas ou passadas como parâmetros em chamadas de métodos, são checadas por compatibilidade de tipo. Não há coerções ou conversões de conflitos de tipos como em algumas linguagens. O compilador Java checa todas as expressões e parâmetros para garantir que todos os tipos são compatíveis. Quaisquer incompatibilidades são erros que devem ser corrigidos antes que o compilador termine de compilar a classe.

Os Tipos Primitivos

Java define oito tipos *primitivos* de dados: **byte, short, int, long, char, float, double** e **boolean**. Os tipos primitivos são também comumente referidos como tipos simples, e ambos os termos são usados neste livro. Estes podem ser colocados em quatro grupos:

- **Inteiros** – esse grupo inclui **byte**, **short**, **int** e **long**, que são valores inteiros com sinais;
- **Pontos-flutuantes** – esse grupo inclui **float** e **double**, que representam números com precisão decimal.
- **Caracteres** – esse grupo inclui **char**, que representa símbolos em um conjunto de caracteres, como letras e números.
- **Booleanos** – esse grupo inclui **boolean**, que é um tipo especial para representação de valores verdadeiro/falso.

Pode-se usar esses tipos como estão, ou para construir matrizes ou tipos de classes. Assim, eles formam a base para todos os outros tipos de dados que se pode criar.

Os tipos primitivos representam valores únicos – não objetos complexos. Embora Java seja totalmente orientada a objetos, os tipos primitivos não são. Eles são análogos aos tipos simples encontrados na maioria das linguagens não orientadas a objetos. A razão

para isso é eficiência. Transformar tipos primitivos em objetos teria prejudicado muito o desempenho.

Os tipos primitivos são definidos para ter um intervalo e um comportamento matemático. Linguagens como C e C++ permitem que o tamanho de um inteiro varie com base nos ditames do ambiente de execução. No entanto, Java é diferente. Devido ao requisito de portabilidade de Java, todos os tipos de dados têm um intervalo estritamente definido. Por exemplo, um **int** é sempre 32 bits, independentemente da plataforma. Isso permite que programas sejam escritos com a garantia de que serão executados *sem portar* nenhuma arquitetura de máquina. Embora a especificação rigorosa do tamanho de um inteiro possa causar uma pequena perda de desempenho em alguns ambientes, isso é necessário para obter portabilidade.

Inteiros

Java define quatro tipos de inteiros: **byte**, **short**, **int** e **long**. Todos esses têm sinais, valor positivos e negativos. Java não suporta números inteiros positivos sem sinais. Muitas outras linguagens de computador suportam ambos inteiros com sinais e sinais. No entanto, os designers de Java consideram que inteiros sem sinais desnecessários. Especificamente, eles consideram que o conceito de *sem sinal* foi usado principalmente para especificar o comportamento do bit de *ordem superior* (sequência binária que carrega o maior valor numérico), o que define o *sinal* de um valor inteiro. Como discutido no Capítulo 4, Java gerencia o significado de bit de ordem superior de maneira diferente, adicionando um operador especial “deslocando sem sinais à direita”. Assim, a necessidade de um tipo inteiro sem sinal foi eliminada.

A largura de um tipo inteiro não deve ser pensada como a quantidade de armazenamento que consome, mas como o comportamento que define para variáveis e expressões desse tipo. O ambiente de tempo de execução Java é livre para usar o tamanho que desejar, desde que os tipos se comportem conforme o declarado. A largura e os intervalos desses tipos inteiros variam muito, conforme mostrado nesta tabela:

Nome	Largura	Intervalo
long	64	-9.223.372.036.854.775.808 para 9.223.372.036.854.775.807
int	32	-2.147.483.648 para 2.147.483.647
short	16	-32.768 para 32.767
byte	8	-128 para 127

byte

O menor tipo de inteiro é byte. Esse é um tipo de 8 bits com sinal que varia de -128 a 127. Variáveis do tipo byte são especialmente úteis quando se trabalha com um fluxo de dados de uma rede ou arquivo. Elas também são úteis quando se trabalha com dados

binários brutos que não podem ser diretamente compatíveis com outros tipos internos de Java.

Variáveis **byte** são declaradas pelo uso da palavra-chave **byte**. Por exemplo, duas variáveis **byte** chamadas **b** e **c** são declaradas a seguir:

```
byte b, c;
```

short

short é um tipo de 16 bits com sinal. Tem um intervalo de -32.768 a 32.767. Provavelmente é o tipo menos usado em Java. Aqui há alguns exemplos de declarações de variáveis **short**.

```
short s;
short t;
```

int

O tipo mais comumente utilizado é **int**. É um tipo de 32 bits com sinal que tem um intervalo de 2.147.483.648 a 2.147.483.647. Além de outros usos, variáveis do tipo **int** são comumente empregadas para controlar loops e para indexar matrizes. Os tipos **byte** e **short** são usados em uma expressão, eles são *promovidos* para **int** quando a expressão é validada, por isso, mesmo que não se precise de um intervalo tão grande em determinada situação usar esses tipos não será mais eficiente do que usar **int** quando um número inteiro é necessário.

long

long é um tipo de 64 bits com sinal e é útil para aquelas ocasiões em que o tipo **int** não é grande o suficiente para armazenar o valor desejado. O intervalo de um **long** é bastante grande. Isso o torna útil quando grandes números são necessários. Por exemplo, aqui está um programa que computa o número de milhas que a luz percorrerá em um número especifico de dias.

```
// Computa a distância que a luz percorrerá usando variáveis long
class Light
        {
                public static void main (String args[])
                {
                        int lightspeed;
                        long days;
                        long seconds;
                        long distance;
```

```
                    // velocidade da luz aproximada em milhas por segundo
                    lightspeed=186000;

                    // especifica o número de dias
                    days=1000;

                    // conversão para segundos
                    segundos=days*24*60*60;

                    // computa a distância
                    distance=lightspeed*seconds;

                    System.out.print("Em " + days);
                    System.out.print(" dias a luz terá percorrido ");
                    System.out.println(distance + " milhas.");
                }
        }
```

Esse programa gera a seguinte saída:

Em 1000 dias a luz terá percorrido 16070400000000 milhas.

Claramente, o resultado não poderia ter sido armazenado em uma variável int.

Pontos-Flutuantes

Números ponto-flutuantes, também chamados de números reais, são usados quando são avaliadas expressões que requerem uma precisão fracionária. Por exemplo, cálculos como raiz quadrada ou transcendentais como seno e cosseno, resultam em um valor cuja precisão requer um tipo ponto-flutuante. Java implementa o conjunto padrão (IEE-754) de tipos e operadores de ponto-flutuante. Há dois tipos de pontos-flutuantes, **float** e **double**, que representam números de precisão única e dupla, respectivamente. A largura e o intervalo são mostrados aqui:

Nome	Largura em Bits	Intervalo Aproximado
double	64	4.9e-324 a 1.8e+308
float	32	1.4e-045 a 3.4e+038

float

O tipo **float** especifica uma precisão única que usa 32 bits de armazenamento. A precisão única é mais rápida em alguns processadores e ocupa metade do espaço que uma precisão dupla, mas se tornará imprecisa quando os valores forem muito grandes ou

muito pequenos. Variáveis do tipo **float** são usadas quando é preciso um componente fracionário, mas não exige um alto grau de precisão. Por exemplo, **float** pode ser útil ao representar valores monetários.

Aqui estão alguns exemplos de declarações de variáveis do tipo **float**:

```
float hightemp, lowtemp;
```

double

A precisão dupla, como indicado pela palavra-chave **double**, usa 64 bits para armazenar um valor. A precisão dupla é realmente mais rápida do que uma precisão única em alguns processadores modernos que têm sido otimizados para cálculos matemáticos de alta velocidade. Todas as funções matemáticas transcendentais, como **sin()**, **cos()**, e **sqrt()**, retornam valores duplos. Quando é preciso manter a precisão de muitos cálculos iterativos ou manipular números de grande valor, **double** é a melhor opção.

Aqui está um pequeno programa que usa variáveis **double** para computar a área de um círculo:

```
// computa a área de um círculo.
class Area
        {
                public static void main (String args[])
                        {
                                double pi, r, a;

                                // raio do círculo
                                r=10.8;

                                // pi aproximadamente
                                pi=3.1416;

                                // computa a área
                                a=pi*r*r;

                                System.out.println("A área do círculo é" + a);
                        }
        }
```

Caracteres

Em Java, o tipo de dados usado para armazenar caracteres é **char**. Um ponto chave para entender é que Java usa *Unicode* para representar caracteres. O Unicode define um conjunto de caracteres totalmente internacional que pode representar todos os caracteres encontrados em todos os idiomas humanos. É uma unificação de dezenas de conjuntos e caracteres, como Latim, Grego, Arábico, Cirílico, Hebraico e muito mais. No momento da criação de Java, Unicode exigia 16 bits. Assim, em Java **char** é um tipo de 16 bits. O intervalo de um **char** é de 0 a 65.536. Não há caracteres negativos. O conjunto padrão de caracteres conhecido como ASCII ainda varia de 0 a 127 como sempre, e o conjunto estendido de caracteres de 8 bits, ISO-Latin-1, varia de 0 a 255. Como Java foi desenvolvido para permitir a codificação de programas para uso em todo o mundo, faz sentido usar Unicode para representar caracteres. Obviamente, o uso de Unicode é um tanto ineficiente para linguagens como Inglês, Alemão, Espanhol, ou Francês, cujos caracteres podem facilmente ser contidos em 8 bits. Mas esse é o preço que deve ser pago pela portabilidade global.

Aqui está um programa que demonstra variáveis **char**:

```
// Demonstra o tipo de dados char.
class CharDemo
	{
		public static void main(String args[])
			{
				char ch1, ch2;

				// Código para X
				ch1=88;
				ch2='Y';

				System.out.print("ch1 e ch2: ");
				System.out.println(ch1 + " " + ch2);
			}
	}
```

Esse programa mostra a seguinte saída:

```
ch1 e ch2: X Y
```

Note que ch1 recebe o valor 88, que é o valor ASCII (e Unicode) correspondente à letra X. Como mencionado, o conjunto de caracteres ASCII ocupa os primeiros 127 valores

no conjunto de caracteres Unicode. Por esse motivo, todos os “truques antigos” usados em outras linguagens também funcionarão em Java.

Embora **char** seja projetado para conter caracteres Unicode, ele também pode ser usado como um tipo inteiro no qual se pode executar operações aritméticas. Por exemplo, pode-se somar dois caracteres, ou incrementar o valor de uma variável **char**. Considere o seguinte programa:

```
// variáveis char se comportando como inteiros.
clas CharDemo2
    {
        public static void main(String args[])
            {
                char cha1;

                ch1='X';
                System.out.println("ch1 contém " + ch1);

                // incrementa ch1
                ch1++;
                System.out.println("ch1 agora é " + ch1);
            }
    }
```

As saídas geradas por esse programa são mostradas aqui:

ch1 contém X
ch1 agora é Y

No programa, **ch1** recebe primeiro o valor X. Depois, **ch1** é incrementado. Isso resulta em **ch1** contendo Y, o próximo caracteres na sequência ASCII (e Unicode).

Nota: Na especificação formal para Java, **char** é chamado de tipo integral, o que significa que está na mesma categoria geral que **int**, **short**, **long** e **byte**. No entanto, como seu principal uso é para representar caracteres Unicode, **char** é comumente considerado uma categoria própria.

Booleanos

Java tem um tipo primitivo, chamado **booleano**, para valores lógicos. Ele pode ter apenas um de dois valores possíveis, **true** ou **false**. Este é o tipo retornado por todos os operadores relacionais, como no caso de **a < b**. Booleano também é o tipo *exigido* pelas expressões condicionais que governam as instruções de controle como **if** e **for**.

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

```
// Demonstração de valores booleanos
class BoolTest
	{
		public static void main(String args[])
			{
				boolean b;

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

				// Um valor booleano pode controlar a declaração if
				if (b)
				System.out.println("Isto é executado.");

				b=false;
				if(b)
				System.out.println("Isto não é executado.");

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

A saída gerada por esse programa é mostrada aqui:

```
b é false
b é true
Isto é executado.
10 > 9 é true
```

Há três coisas interessantes a serem observadas sobre esse programa. Primeiro, como pode-se ver, quando um valor **booleano** é gerado por **println()**, "true" ou "false" é mostrado. Segundo, o valor de uma variável **booleana** é suficiente, por si só, para controlar a instrução **if**. Não há necessidade de escrever uma declaração **if** como esta:

```
if (b==true)
```

...

Terceiro, o resultado gerado de um operador relacional, como **<**, é um valor **booleano**. É por isso que a expressão **10 > 9** mostra o valor “true”. Além disso, o conjunto fora dos parênteses ao redor de **10 > 9** é necessário porque o operador **+** tem uma precedência mais alta que **>**.

Um Olhar Mais Atento Sobre Literais

Os literais foram mencionados brevemente no Capítulo 2. Agora que os tipos internos foram descritos formalmente, pode-se dar uma olhada neles.

Inteiros Literais

Inteiros são provavelmente o tipo inteiro mais comumente usados em um programa típico. Qualquer valor numérico é um inteiro literal. Como exemplos tem-se 1, 2, 3 e 42. Esses são todos valores decimais, o que significa que eles estão descrevendo um número na base 10. Duas outras bases que podem ser usadas em inteiros literais são *octal* (base oito) e *hexadecimal* (base 16). Valores Octais são denotados em Java por um zero inicial. Números decimais normais não podem ter um zero à esquerda. Assim, o valor aparentemente valido 09 produzirá um erro de compilação, pois 9 está fora do intervalo de 0 a 7 dos octais. Uma base mais comum para números usada por programadores é hexadecimal, que corresponde perfeitamente com o módulo de palavras de tamanho 8. Como 8, 16, 32, e 64. Uma constante hexadecimal é denotada por zero seguido de x (**0x** ou **0X**). O intervalo de um dígito hexadecimal é de 0 a 15, portanto, de A até F (ou de a até f) são substituídos por 10 a 15.

Inteiros literais criam um valor **int**, que em Java é um valor inteiro de 32 bits. Quando um valor literal é atribuído a uma variável **byte** ou **short**, nenhum erro é gerado se o valor literal estiver dentro do intervalo do tipo apontado. Um inteiro literal pode sempre ser atribuído para uma variável **long**. No entanto, para especificar um literal **long**, precisa-se informar explicitamente que o valor literal é do tipo **long**. Isso é feito anexando um L maiúsculo ou minúsculo ao literal. Por exemplo, 0x7fffffffffffffL ou 9223372036854775807L é o maior **long**. Um número inteiro também pode ser atribuído a um **char**, desde que esteja dentro do intervalo.

Pode-se também especificar inteiros literais a binários. Para tanto, prefixa-se o valor com **0b** ou **0B**. Por exemplo, isso especifica o valor decimal 10 usando um binário literal:

```
int x=0b1010;
```

Entre outros usos, a adição de literais binários facilita a inserção de valores usados como máscaras de bits. Nesse caso, a representação decimal (ou hexadecimal) do valor não transmite visualmente seu significado em relação ao seu uso. O literal binário o faz.

Pode-se embutir um ou mais underlines em um inteiro literal. Isso facilita a leitura de literais inteiros. Quando um literal é compilado, os underlines são descartados. Por exemplo, dado:

int x=123_456_789;

o valor dado para **x** é 123.456.789. Os underlines serão ignorados. Underlines podem apenas ser usados para separar dígitos. Eles não podem vir no começo ou no final de um literal. No entanto, é permitido que mais de um underline possa ser usado entre dois dígitos. Por exemplo, isso é valido:

Por exemplo, isso é válido:

int x=123__456__789;

O uso de um underline em um inteiro literal é especificamente útil ao codificar coisas como números de telefones, números de identificação, números de peças e assim por diante.

Eles também são úteis para fornecer agrupamentos visuais ao especificar binários literais. Por exemplo, valores binários são geralmente agrupados visualmente em unidades de quatro dígitos, como mostrado aqui:

int x=0b1101_0101_0001_1010;

Pontos-Flutuantes Literais

Números ponto-flutuantes representam valores decimais com um componente fracionário. Eles podem ser expressos em notação padrão ou notação científica. A *notação padrão* consiste de um componente chamado por um ponto decimal chamado por um componente fracionário. Por exemplo, 2.0, 3.14159, e 0.6667 representa números de ponto-flutuante em notação padrão valido. A *notação científica* usa uma notação padrão, números ponto-flutuante acrescido de um sufixo que especifica a potência de 10 pela qual o número deve ser multiplicado. O expoente é indicado por um *E* ou *e* seguido por um número decimal que pode ser positivo ou negativo. Como exemplos tem-se: 6.022E23, 314159E-05, e 2e+100.

Pontos-flutuantes literais em Java são por padrão de precisão **double**. Para especificar um **float** literal, deve-se anexar um *F* ou *f* para a constante. Pode-se também especificar explicitamente um **double** literal ao anexar um *D* ou *d*. Fazer isso, claramente, é redundante. O tipo **double** padrão consome 64 bits de memória, enquanto um **float** menor requer apenas 32 bits.

Hexadecimais pontos-flutuantes literais também são suportados, mas eles são raramente usados. Eles devem estar em uma forma semelhante à notação científica, mas um

P ou **p** é usado, em vez de um **E** ou **e**. Por exemplo, 0x12.2P2 é um ponto-flutuante literal válido.

O valor seguinte a **P**, chamado *expoente binário*, indica a potência de dois pela qual o número é multiplicado. Portanto, **0x12.2P2** representa 72,5. Pode-se embutir uma ou mais underlines em um ponto-flutuante literal. Esse recurso funciona da mesma forma de um inteiro literal, que acabou de ser descrito. Seu objetivo é facilitar a leitura de pontos-flutuantes literais grandes. Quando o literal é compilado, os underlines são descartados. Por exemplo, dado:

double num=9_423_497_862.0;

o valor dado a **num** será 9.423.497.862,0. Os underlines serão ignorados. Como no caso de inteiros literais, underlines são usados apenas para separar dígitos. Eles não podem vir no ínicio ou no fim de um literal. No entanto, é permitido que mais de um underline seja usado entre dois dígitos. Também é permitido usar underlines em porções fracionárias de um número. Por exemplo:

double num=9_423_497.1_0_9;

é permitido. Nesse caso, a parte fracionária é ,109.

Booleanos Literais

Booleanos literais são simples. Há apenas dois valores lógicos que um **booleano** possa ter, **true** e **false**. Os valores de **true** e **false** não são convertidos em representações numéricas. O **true** literal em Java não é igual a 1, nem o valor **false** literal é igual a 0. Em Java, os booleanos literais podem ser apenas atribuídos a variáveis declaradas como **boolean** ou usados em expressões com operadores booleanos.

Caracteres Literais

Caracteres em Java são índices no conjunto de caracteres Unicode. São valores de 16 bits que podem ser convertidos em números inteiros e manipulados com os operadores inteiros, como os operadores de adição e de subtração. Um caractere literal é representado dentro de pares de aspas simples. Todos os caracteres ASCII visíveis podem ser inseridos diretamente dentro das aspas, como ‘a’, ‘z’, e ‘@’. Para caracteres que são impossíveis de inserir diretamente, existem várias sequencias de escape que permitem inserir o caractere necessário, como ‘\’ para o próprio caractere de aspas simples e ‘\n’ para o caractere de nova linha. Há também um mecanismo para inserir diretamente o valor de um caractere em octal ou hexadecimal. Para notação octal, usa-se a barra invertida seguida pelo número de três dígitos. Por exemplo, ‘\141’ é a letra ‘a’ Para hexadecimal, insere-se ‘\u’ e, em seguida, exatamente quatro dígitos hexadecimais. Por exemplo, ‘\u0061’ é o ‘a’ em ISO-Latin-1

porque o byte superior é zero. ‘\ua432’ é um caractere japonês katakana. A tabela a seguir mostra as sequencias de escape de caracteres:

Sequência de escape	Descrição
\ddd	Caractere Octal (ddd)
\uxxxx	Caractere Unicode Hexadecimal (xxxx)
\’	Aspas Simples
\”	Aspas Duplas
\\	Barra Invertida
\r	Retorno ao começo da linha
\n	Quebra de linha
\f	Alimentação do formulário
\t	Tab
\b	Backspace

Strings Literais

Strings literais em Java são especificadas como na maioria das outras linguagens – colocando uma sequência de caracteres entre um par de aspas duplas. Exemplos de strings literais são:

“Hello World”

“two\nlines”

“\”Isso está entre aspas”\”

Nota: Em algumas outras linguagens as sequências de caracteres são implementadas como matrizes de caracteres. No entanto, este não é o caso em Java. Strings são realmente tipos de objetos. Como mostrado mais adiante, como Java implementa strings como objetos, Java inclui amplos recursos de manipulação de strings, poderosos e fáceis de usar.

Variáveis

As variáveis são a unidade básica da memória de um programa Java. Uma variável é definida pela combinação de um identificador, um tipo, e um inicializador opcional. Além disso, todas as variáveis têm um escopo, que define sua visibilidade e seu tempo de vida. Esses elementos são examinados a seguir.

Declarando uma Variável

Em Java, todas as variáveis devem ser declaradas antes de serem usadas. A forma básica da declaração de uma variável é mostrada aqui:

tipo identificador [= valor][,identificador[=valor]...];

Aqui, tipo é um dos tipos atômicos de Java, ou o nome de uma classe ou interface (tipos de classe e de interface são discutidos na Parte I deste livro). O identificador é o nome da variável. Pode-se inicializar uma variável especificando um sinal de igual e um valor. Sendo que a expressão de inicialização deve resultar em um valor do mesmo tipo

(ou compatível) especificado para a variável. Para declarar mais de uma variável do tipo especificado, usa-se uma lista separada por vírgula.

Aqui estão muitos exemplos de declarações de variáveis de vários tipos.

```
// declara três inteiros: a, b e c.
int a, b, c;

// declara mais três inteiros, inicializando d e f.
int d=3, e, f=5;

// inicializa z;
byte z=22;

// declara uma aproximação de pi.
double pi=3.14159;
// a variável x tem o valor 'x'.
char x='x';
```

Os identificadores não têm nada intrínseco em seus nomes que indique seu tipo. Java permite que qualquer identificador formado adequadamente tenha qualquer tipo declarado.

Inicialização Dinâmica

Embora os exemplos anteriores usem apenas constantes como inicializadores, Java permite que as variáveis sejam inicializadas dinamicamente, usando qualquer expressão válida para permitir que as variáveis sejam inicializadas dinamicamente, usando qualquer expressão válida no momento em que a variável é declarada.

Por exemplo, aqui está um pequeno programa que computa o comprimento da hipotenusa de um triângulo retângulo dando o comprimento dos dois lados opostos:

```
// Demonstra a inicialização dinâmica.
class DynInit
    {
        public static void main(String args[])
            {
                double a=3.0, b=4.0;

                // c é inicializado dinamicamente.
                double c=Math.sqtr(a*a+b*b);
```

```
                System.out.println("A hipotenusa é " + c);
            }
        }
```

Aqui, as três variáveis – a, b e c – são declaradas. As duas primeiras, **a** e **b**, são inicializadas por constantes. No entanto, **c** é inicializada dinamicamente ao comprimento da hipotenusa (Usando o teorema de Pitágoras). O programa usa outros método interno de Java, **sqrt()**, que é o membro da classe **Math**, para computar a raiz quadrada de um argumento. O ponto principal aqui é que a expressão de inicialização pode usar qualquer elemento valido no momento da inicialização, incluindo chamadas para métodos, outras variáveis ou literais.

O Escopo e o Tempo de Vida das Variáveis

Até o momento, todas as variáveis usadas foram declaradas no início do método **main()**. No entanto, Java permite que variáveis sejam declaradas em qualquer bloco. Como explicado no Capítulo 2, um bloco é iniciado com uma chave de abertura e termina com uma chave de fechamento. Um bloco define um escopo. Assim, a cada vez que um novo bloco é iniciado, cria-se um novo escopo. Um escopo determina quais objetos são visíveis para outras partes do programa. Também determina o tempo de vida desses objetos.

Não é incomum pensar em termos de duas categorias gerais de escopos: global e local. No entanto, esses escopos tradicionais não se encaixam bem no modelo estrito orientado a objetos de Java. Embora seja possível criar o que significa ser um escopo global, ele é de longe a exceção, não a regra. Em Java, os dois maiores escopos são aqueles definidos por uma classe e aqueles definidos por um método. Mesmo que essa distinção seja um tanto artificial. No entanto, como o escopo das classes possui várias propriedades únicas e atributos que não se aplicam ao escopo definido por um método, essa distinção tem algum sentido. Por causa das diferenças, uma discussão do escopo de classe (e variáveis declaradas nela) é adiada até o Capítulo 6, quando as classes serão descritas. Por enquanto, serão examinados 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 serão incluídos no escopo do método. Um escopo de método termina com a chave de fechamento. Esse bloco de código é chamado de *corpo do método*.

Como uma regra geral, variáveis chamadas dentro de um escopo não são visíveis (ou seja, acessíveis) para o código que é definido de fora desse escopo. Assim, quando uma variável é declarada em um escopo, ela é localizada e protegida de acesso e/ou

modificação não autorizada. De fato, as regras de escopo fornecem a base para o encapsulamento. Uma variável declarada em um bloco é chamada de *variável local*.

Escopos podem ser aninhados. Por exemplo, cada vez que um bloco de código é criado, um novo escopo aninhado é criado. Quando isso ocorre, o escopo externo inclui o escopo interno. Isso significa que os objetos declarados no exterior do escopo serão visíveis para o código no interior do escopo. No entanto, o contrário não é verdade. Objetos declarados dentro do escopo interno não serão visíveis fora dele.

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

```
// Demonstração do escopo de um bloco.
class Scope
	{
		public static void main(String args[])
			{
				// conhecida por todo o código dentro de main
				int x;

				x=10;
				if(x==10)
					{
						// começa um novo escopo
						// conhecida apenas neste bloco
						int y=20;

						System.out.println("x e y: " + x + " " + y);
						x=y*2;
					}
				// y=100;
				// Erro! y não é conhecida aqui

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

Como os comentários indicam, a variável **x** é declarada no início do escopo de **main()** e é acessível a todos os código subsequentes em **main()**. No bloco **if**, **y** é declarado. Como um bloco define um escopo, **u** é visível apenas para outro código nesse bloco. É por

isso que fora desse bloco, a linha **y=100** é comentada. Se for removido o símbolo de comentário, ocorrerá um erro em tempo de compilação, porque **y** não é visível fora desse bloco. No bloco **if**, **x** pode ser usada porque o código em um bloco (ou seja, um escopo aninhado) tem acesso a variáveis declaradas por um escopo anexado.

Dentro de um bloco, variáveis podem ser declaradas em qualquer ponto, mas são válidas apenas após serem declaradas. Assim, se uma variável for definida no início de um método, ela estará disponível para todo o código desse método. Por outro lado, se uma variável for declarada no final de um bloco, ela será efetivamente inútil, porque nenhum código terá acesso a ela. Por exemplo, este fragmento é invalido porque **count** não pode ser usada antes de sua declaração:

```
// Este fragmento está errado!
count=100;
//ops! Não se pode usar count antes de ser declarada!
int count;
```

Aqui está outro ponto importante a ser lembrado: variáveis são declaradas quando seu escopo é inserido, e destruídas quando seu escopo é deixado. Isso significa que uma variável não manterá seu valor depois que estiver fora do escopo. Portanto, as variáveis declaradas em um método não manterão seus valores entre as chamadas para esse método. Além disso, uma variável declarada dentro de um bloco perderá seu valor quando o bloco for deixado. Assim, o tempo de vida de uma variável é limitada ao seu escopo.

Se uma declaração de variável incluir um inicializador, então essa variável será reinicializada a cada vez que o bloco em que é declarada for inserida. Por exemplo, considere o próximo programa:

```
// Demonstração do tempo de vida de uma variável.
class LifeTime
    {
        public static void main(String args[])
            {
                int x;
                for(x=0; x<3;x++)
                    {
                        // y será inicializada com o bloco
                        int y=-1;

                        System.out.println("y é: " + y);
```

```
                    y=100;
                    System.out.println("y agora é: " + y);
                }
            }
        }
```

A saída gerada por esse programa é mostrada aqui:

```
y é: -1
y agora é: 100
y é: -1
y agora é: 100
y é: -1
y agora é: 100
```

Aqui **y** é inicializado em -1 a cada vez que o loop **for** é inserido. Mesmo que seja atribuído posteriormente o valor 100, esse valor será perdido.

Um último ponto: Embora blocos possam ser aninhados, pode-se declarar uma variável para ter o mesmo nome que em um escopo externo. Por exemplo, o programa a seguir não está correto:

```
// Este programa não será compilado
class ScopeErr
    {
        public static void main(String args[])
            {
                int bar=1;
                    {
                        // criado um novo escopo
                        /* erro em tempo de compilação – bar já havia sido declarada */
                        int bar=2;
                    }
            }
    }
```

Conversão e Fusão de Tipos

É bastante comum atribuir um valor de um tipo para uma variável de outro tipo. Se os dois tipos forem compatíveis, então Java executará a conversão automaticamente. Por exemplo, sempre é possível atribuir um valor **int** para uma variável **long**. No entanto, nem

todos os tipos são compatíveis, e portanto, nem todas as conversões de tipos são implicitamente permitidas. Por exemplo, não existe uma versão automática definida de **double** para **byte**. Felizmente, ainda é possível obter uma conversão entre tipos incompatíveis. Para fazer isso, deve-se usar uma conversão, que execute uma conversão explícita entre tipos incompatíveis.

Conversões Automáticas de Java

Quando um tipo de dados é atribuído para outro tipo de variável, ocorrerá uma *conversão automática de tipo*, se as duas condições a seguir forem atendidas:

- Os dois tipos serem compatíveis;
- O tipo destinado for maior que o tipo de origem.

Quando essas duas condições são atendidas, ocorre uma conversão crescente. Por exemplo, o tipo **int** é sempre grande o suficiente para armazenar todos os valores válidos de **byte**, portanto, nenhuma conversão explícita é necessária.

Para ampliar as conversões, os tipos numéricos, incluindo os números inteiros e os pontos-flutuantes, são compatíveis entre si. No entanto, não há conversões automáticas de tipos numéricos para **char** ou **boolean**. Além disso, **char** e **boolean** não são compatíveis um com o outro.

Como mencionado anteriormente, Java também realiza uma conversão de tipo automática quando armazenar uma constante inteira literal em variáveis do tipo **byte**, **short**, **long** ou **char**.

Fundindo Tipos Incompatíveis

Embora as conversões de tipo sejam úteis, elas não atenderão a todas as necessidades. Por exemplo, para atribuir um valor **int** para uma variável **byte** a conversão é chamada de conversão restrita (por **int** ser maior do que **byte**).

Para criar uma conversão entre dois tipos incompatíveis, deve-se usar uma conversão de tipo explicita, sua forma geral é:

tipo-destino valor;

Aqui, o *tipo-destino* especifica o tipo desejado para o qual converter o valor especificado. Por exemplo, o fragmento a seguir converte um **int** para um **byte**. Se o valor inteiro for maior do que o intervalor de **byte**, ele será reduzido usando módulo (o resto da divisão inteira) do intervalo de **byte**.

```
int a;
byte b;

b=byte a;
```

Uma diferente conversão de tipo irá ocorrer quando um valor de ponto-flutuante é atribuído a um tipo inteiro: truncamento. Inteiro não podem ter componentes fracionários. Assim, quando um valor de ponto-flutuante é atribuído a um tipo inteiro, o componente fracionário é perdido. Por exemplo, se o valor 1,23 for atribuído a um inteiro, o valor resultante será simplesmente 1. O valor 0,23 será truncado. Obviamente, se o tamanho do componente do número for muito grande para caber no tipo inteiro de destino, então esse valor será reduzido pelo módulo do intervalo do tipo de destino.

O programa a seguir demonstra algumas conversões de tipo que são necessárias:

```
// Demonstração de conversões.
class Conversão
        {
                class static void main(String args[])
                        {
                                byte b;
                                int i=257;
                                double d=323,142;

                                System.out.println("\nConversão do tipo int para byte.");
                                b=(byte) i;
                                System.out.println("i e b " + i + " " + b);

                                System.out.println("\nConversão do tipo double para
int.");
                                i=(int) d;
                                System.out.println("d e i " + d + " " + i);

                                System.out.println("\nConversão do tipo double para
byte.");
                                b=(byte) d;
                                System.out.println("d e b " + d + " " + b);
                        }
        }
```

Esse programa gera a saída a seguir:

```
Conversão do tipo int para byte.
i e b 257 1
```

Conversão do tipo double para int.

d e i 323,142 323

Conversão do tipo double para byte.

d e b 323,142 67

Quando valor 257 é convertido para uma variável **byte**, o resultado é o resto da divisão de 257 por 256 (o intervalo de um **byte**), que é 1 neste caso. Quando **d** é convertido para **int**, seu componente fracionário é perdido. Quando **d** é convertido para **byte**, seu componente fracionário é perdido, e o valor é reduzido ao módulo de 256, que neste caso é 67.

Promoção Automática de Tipo em Expressões

Além das atribuições, há outro local em que determinadas conversões devem ocorrer: em expressões. Em uma expressão, a precisão exigida para um valor intermediário excederá o intervalo de qualquer operando. Por exemplo, na expressão seguinte:

```
byte a=40;
byte b=50;
byte c=100;
int d=a*b/c;
```

O resultado do termo intermediário **a * b** excede facilmente o intervalo de quaisquer operandos de **byte**. Para manipular esse tipo de programa, Java promove automaticamente cada operando **byte**, **short**, ou **char** para **int** quando avalia uma expressão. Isso significa que a subexpressão **a*b** é realizada usando inteiros – não bytes. Assim, 2.000, é o resultado da expressão intermediária, **50*40**, está correto mesmo que **a** e **b** são ambas especificadas como **byte**.

Por mais úteis que sejam as promoções automáticas, elas podem causar erros confusos em tempo de compilação. Por exemplo, esse código aparentemente correto causa um problema.

```
byte b=50;
b=b*2;
// Erro! Não é possível atribuir um int a um byte!
```

O código está tentando armazenar 50 * 2, um valor de **byte** perfeitamente válido, retornado para uma variável **byte**. No entanto, como os operandos foram promovidos automaticamente para **int** quando a expressão foi avaliada, o resultado também foi promovido para **int**. Assim, o resultado da expressão agora é do tipo **int**, que não pode ser atribuído para **byte** sem o uso de uma conversão. Isso é verdade mesmo que, como nesse caso específico, o valor que está sendo atribuído ainda caiba no tipo de destino.

Em casos em que se entende as consequências do *overflow* (estouro), usa-se uma conversão explícita, como:

```
byte b=50;
b=(byte)(b*2);
```

que gera o valor correto de 100.

As Regras de Promoção de Tipo

Java define várias regras de *promoção de tipos* que se aplicam a expressões. Elas são as seguintes: Primeiro, todos os valores **byte**, **short**, e **char** são promovidos para **int**, como já descrito. Então, se um operando é um **long**, toda a expressão é promovida para **long**. Se um operando é um **float**, toda a expressão será promovida para **float**. Se qualquer dos operandos for **double**, o resultado será **double**.

O programa a seguir demonstra como cada valor na expressão é promovido para corresponder ao segundo argumento para cada operador binário:

```
class Promote
    {
        public static void main(String args[])
            {
                byte b=42;
                char c='a';
                short s=1024;
                int i=50000;
                float f=5.67f;
                double d=.1234;
                double result=(f*b)+(i/c)-d*s);
                System.out.println((f * b) + " + " + (i / c) + " – " + (d * s));
                System.out.println("resultado = " + result);
            }
    }
```

Um olhar mais atento às promoções de tipo que ocorrem nessa linha do programa:

```
double result=(f*b)+(i/c)-(d*s);
```

Na primeira subexpressão, **f * b, b** é promovido para **float** e o resultado da expressão é **float**. Depois, na expressão **i**/**c**, **c** é promovido para **int**, e o resultado é do tipo **int**. Então, em **d*s**, o valor de **s** é promovido para **double**, e o tipo da subexpressão é **double**. Finalmente, esses três valores intermediários, **float**, **int**, e **double**, são considerados. O

resultado de **float** mais um **int** é um **float**. Então o resultado de um **float** menos o último **double** é promovido para **double**, que é o tipo para o resultado final da expressão.

Matrizes

Uma *matriz* é um grupo de variáveis do mesmo tipo que são referidas por um nome comum. Matrizes de qualquer tipo podem ser criada e devem ter uma ou mais dimensões. Um elemento específico em uma matriz é acessado por seu índice. Matrizes oferecem um meio conveniente de agrupar informações relacionadas.

Matrizes unidimensionais

Uma *matriz unidimensional* é, essencialmente, uma lista de variáveis do mesmo tipo. Para criar uma matriz, deve-se criar uma matriz variável do tipo desejado. A forma geral de uma matriz unidimensional é:

tipo nome[ ];

Aqui, *tipo* declara o tipo de elemento (também chamado de tipo base) da matriz. O tipo de elemento determina o tipo de dados de cada elemento que compreende a matriz. Assim, o tipo de elemento para a matriz determina qual tipo de dados a matriz conterá. Por exemplo, a seguir é declarada uma matriz chamada **month_days** com o tipo “matriz de inteiros”:

Int month_days[ ];

Embora essa declaração estabeleça o fato de que **month_days** é uma matriz variável, nenhuma matriz existe realmente. Para vincular **month_days** com uma matriz física de inteiros, deve-se alocar uma usando **new** e atribui-la a **month_days**. **New** é um operador especial que aloca memória.

Em um capítulo posterior **new** será discutido em mais detalhes, mas deve-se usá-lo agora para alocar memória para matrizes. A forma geral de **new** como se aplica a matrizes unidimensionais aparece da seguinte maneira:

matriz-variável=new tipo[tamanho];

Aqui, *tipo* especifica o tipo de dados que estão sendo alocados, *tamanho* especifica o número de elementos na matriz, e *matriz-variável* é a matriz que está sendo vinculada à matriz. Ou seja, para usar **new** para alocar uma matriz, deve-se especificar o tipo e o número de elemento para alocar. Os elementos na matriz alocados por **new** serão inicializados em zero (para tipos numéricos), **false** (para **booleano**), ou **null** (para referenciar tipos, que serão descritos no próximo capítulo). Esse exemplo aloca uma matriz de 12 elementos inteiros e vincula-os a **month_days**:

month_days=new int[12];

Após a execução dessa instrução, **month_days** se referirá a uma matriz de 12 inteiros.

Além disso, todos os elementos na matriz serão inicializados em zero.

A obtenção de uma matriz é um processo de dois passos. Primeiro, deve-se declarar uma matriz variável do tipo desejado. Segundo, deve-se alocar a memória que armazenará a matriz, usando **new**, e atribuindo-o para a matriz variável. Assim, em Java todas as matrizes são alocadas dinamicamente (esse conceito será detalhado mais adiante neste livro).

Depois de alocada uma matriz, pode-se acessar um elemento específico por seu índice entre colchetes. Todos os índices das matrizes começam em zero. Por exemplo, esta instrução atribui o valor 28 para o segundo elemento de **month_days**:

```
month_days[1]=28;
```

A próxima linha mostra o valor armazenado no índice 3:

```
System.out.println(month_days[3]);
```

Reunindo todas as partes, aqui está um programa que cria uma matriz do número de dias em cada mês:

```
// Demonstra uma matriz unidimensional.
    class Matriz
        {
            public static void main(String args[])
                {
                    int month_days[12];
                    month_days[0]=31;
                    month_days[1]=28;
                    month_days[2]=31;
                    month_days[3]=30;
                    month_days[4]=31;
                    month_days[5]=30;
                    month_days[6]=31;
                    month_days[7]=31;
                    month_days[8]=30;
                    month_days[9]=31;
                    month_days[10]=30;
                    month_days[11]=31;
                    System.out.println("Abril tem " + month_days[3] + "
dias.");
```

```
                }
        }
```

Quando executado, esse programa imprime o numero de dias em abril. Como mencionado, o índice da matriz inicia em zero, portanto, o número de dias em abril é **month_days[3]** ou 30.

É possível combinar a declaração da matriz variável com a alocação da própria matriz, como mostrado aqui:

```
int month_days[]=new int[12];
```

Matrizes podem ser inicializadas quando são declaradas. O processo é praticamente o mesmo usado para inicializar os tipos simples. Um *inicializador de matriz* é uma lista de expressões separadas por vírgulas e separadas por chaves. As vírgulas separam os valores dos elementos da matriz. A matriz será criada automaticamente grande o bastante para armazenar o número de elementos especificado no inicializador de matriz. Não há necessidade de usar **new**. Por exemplo, para armazenar o número de dias em cada mês, o código a seguir cria um inicializador de inteiros.

```
// Uma versão melhorada do programa anterior.
class AutoArray
    {
        public static void main(String args[])
            {
                int month_days[]={31,28,31,30,31,30,31,31,30,31,30,31};
                System.out.println("Abril tem " + month_days[3] + " dias.");
            }
    }
```

Quando executado esse programa o resultado será o mesmo gerado pela versão anterior.

Java verifica rigorosamente para garantir que não se tente armazenar ou referenciar valores fora do intervalo da matriz. O sistema em tempo de execução Java verifica se todos os índices da matriz estão no intervalo correto. Por exemplo, o sistema em tempo de execução verificará o valor de cada índice em **month_days** para garantir que estão entre 0 e 11. Um erro em tempo de execução será causado se houver a tentativa de acesso a elementos fora do intervalo da matriz.

Aqui está mais um exemplo que usa uma matriz unidimensional. Ele encontra a média de um conjunto de números.

```
// Média em uma matriz de valores.
```

```
class Average
    {
        public static void main(String args[])
            {
                double nums[]={10.1,11.2,12.3,13.4,14.5};
                double result=0;
                int i;
                for(i=0;i<5;i++)
                    result=result+nums[i];
                System.out.println("A média é " + result / 5);
            }
    }
```

Matrizes Multidimensionais

Em Java, *matrizes multidimensionais* são implementadas como matrizes de matrizes. Para declarar uma matriz multidimensional variável, especifica-se cada índice adicional usando outro conjunto de colchetes. Por exemplo, a seguir é declarada uma matriz bidimensional variável chamada **twoD**:

```
int twoD[][]=new int[4][5];
```

Isso aloca uma matriz 4 por 5 e a atribui a **twoD**. Internamente, essa matriz é implementada como uma *matriz* de *matrizes* de **int**. Conceitualmente, essa matriz se parece com uma tabela de quatro linhas e cinco colunas.

O programa a seguir numera cada elemento na matriz da esquerda para a direita, de cima para baixo, e então mostra esses valores:

```
// Demonstração de uma matriz bidimensional.
class TwoDArray
    {
        public static void main(String args[])
            {
                int twoD[][]=new int[4][5];
                int i, j, k=0;

                for(i=0; i<4; i++)
                    for(j=0; j<5; j++
                        {
                            twoD[i][i]=k;
```

```
                    k++;
                }

                for(i=0; i<4; i++;
                    {
                        for(j=0; j<5; j++)
                            Sys-
tem.out.print(“twod[i][j] + “ “);
                        System.out.println( );

                    }
            }
        }
```

Esse programa gera a seguinte saída:

0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

Quando a memória é alocada para uma matriz multidimensional, é preciso especificar apenas a memória da primeira dimensão (mais a esquerda). Pode-se alocar as dimensões restantes separadamente. Por exemplo, o código a seguir aloca memória para a primeira dimensão de **twoD** quando é declarado. Ele aloca a segunda dimensão separadamente.

```
Int twoD[][]=new int[4][];
twoD[0]=new int[5];
twoD[1]=new int[5];
twoD[2]=new int[5];
twoD[3]=new int[5];
```

Embora não haja vantagem para alocar individualmente a segunda dimensão das matrizes nessa situação, pode haver em outras. Por exemplo, quando se aloca dimensões individualmente, não é preciso alocar o mesmo número de dimensões individualmente, não é preciso alocar o mesmo número de elementos para cada dimensão. Como afirmado anteriormente, como as matrizes multidimensionais são na verdade matrizes de matrizes, o comprimento de cada matriz é variável. Por exemplo, o programa a seguir cria uma matriz bidimensional na qual o tamanho da segunda dimensão é desigual:

```
// Alocação manual de diferentes tamanhos para as dimensões secundárias.
class TwoDAgain
        {
                public static void main(String args[])
                        {
                                int twoD[][]=new int[4][];
                                twoD[0]=new int[1];
                                twoD[1]=new int[2];
                                twoD[2]=new int[3];
                                twoD[3]=new int[4];

                                int i, j, k=0;

                                for(i=0; i<4; i++)
                                        for(j=0; j<i+1; j++)
                                                {
                                                        twoD[i][j]=k;
                                                        k++;
                                                }

                                for(i=0; i<4; i++)
                                        {
                                                for(i=0; j<i+1; j++
                                                        System.out.print(twoD[i][j] + " ");
                                                System.out.println( );
                                        }
                        }
        }
```

Esse programa gera a seguinte saída:

```
0
1 2
3 4 5
6 7 8 9
```

A matriz criada por esse programa se parece com isso:

[0][0]

[1][0] [1][1]

[2][0] [2][1] [2][2]

[3][0] [3][1] [3][2] [3][3]

O uso de matrizes multidimensionais desiguais (ou irregulares) pode não ser apropriado para muitas aplicações, porque é contraria ao que as pessoas esperam encontrar quando uma matriz multidimensional é encontrada. No entanto, matrizes irregulares podem ser usadas efetivamente em algumas situações. Por exemplo, se for necessária uma matriz bidimensional muito grande que seja escassamente povoada (ou seja, aquela na qual nem todos os elementos serão usados), uma matriz irregular poderá ser a solução perfeita.

É possível inicializar matrizes multidimensionais. Para fazer isso, basta fechar o inicializador de cada dimensão dentro de seu próprio conjunto de chaves. O programa a seguir cria uma matriz em que cada elemento contém o produto dos índices de linha e coluna. Pode-se usar expressões e valores literais dentro dos inicializadores de matriz.

```
// Inicializa uma matriz bidimensional.
class Matrix
	{
		public static void main(String args[])
			{
				double m[][]=
					{
						{0*0, 1*0, 2*0, 3*0},
						{0*1, 1*1, 2*1, 3*1},
						{0*2, 1*2, 2*2, 3*2},
						{0*3, 1*3, 2*3, 3*3}
					};
					int i, j;

					for(i=0; i<4; i++)
						{
							for(j=0; j<4; j++)
								System.out.print(m[i][j] + " ");
							System.out.println( );
						}
			}
	}
```

Quando executado esse programa exibirá a seguinte saída:

0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0

Cada linha na matriz é inicializada como especificado nas listas de inicialização.

O programa a seguir cria uma matriz tridimensional 3 por 4 por 5. Ele então carrega cada elemento com o produto dos índices. Finalmente, esses produtos são mostrados.

```
// Demonstração de uma matriz tridimensional.
class ThreeDMatrix
	{
		public static void main(String args[])
			{
				int threeD[][][] = new int[3][4][5];
				int i, j, k;

				for(i=0; i<3; i++)
					for(j=0; j<4; j++)
						for(k=0; k<5; k++)
							threeD[i][j][k]=i*j*k;

				for(i=0; i<3; i++)
					{
						for(j=0; j<4; j++)
							{
								for(k=0; k<5; k++)
									Sys-
tem.out.print(threeD[i][j][k] + " ");
								System.out.println();
							}
						System.out.println( );
					}
			}
	}
```

Esse programa gera a seguinte saída:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Sintaxe Alternativa para Declaração de Matriz

Há uma segunda forma que pode ser usada para declarar uma matriz:

tipo[] nome;

Aqui, os colchetes seguem o especificador de tipo, e não o nome da matriz variável. Por exemplo, as duas declarações seguintes são equivalentes:

```
int al[]=new int[3];
int[] a2=new int[3];
```

As declarações a seguir também são equivalentes:

```
char twod1[][]=new char[3][4];
char[][] twod2=new char[3][4];
```

Essa forma de declaração alternativa é conveniente ao declarar várias matrizes ao mesmo tempo. Por exemplo:

```
// cria três matrizes
int[] nums, nums2,nums3;
```

cria três matrizes variáveis do tipo **int**. É o mesmo que escrever:

```
// cria três matrizes
int nums[], nums2[], nums3[];
```

A forma alternativa de declaração também é útil quando se especifica uma matriz como um tipo de retorno para um método. Ambas as formas são usadas neste livro.

Introdução à Inferência de Tipo com Variáveis Locais

Recentemente, um novo recurso interessante chamado de *inferência de tipo de variável local* foi adicionado para a linguagem Java. Para começar é importante rever dois aspectos importantes de variáveis. Primeiro, todas as variáveis em Java devem ser declaradas antes de serem usadas. Segundo, uma variável pode ser inicializada com um valor quando é declarada. Além disso, quando uma variável é inicializada, o tipo do inicializador

deve ser igual ao (ou convertido para) o tipo declarado da variável. Assim, a princípio, não seria necessário especificar um tipo explicito para uma variável inicializada porque ela poderia ser inferida pelo tipo de inicializador. Obviamente, no passado, essa inferência não era suportada, e todas as variáveis exigiam uma declaração de tipo explicita, independentemente de serem inicializadas ou não.

A partir da JDK 10, agora é possível deixar o compilador inferir o tipo de variável local com base no tipo de inicializador, evitando assim a necessidade de especificar explicitamente o tipo. A inferência de tipo de variável local oferece várias vantagens. Por exemplo, ela pode otimizar o código eliminando a necessidade de especificar redundantemente o tipo de uma variável quando ela pode ser inferida pelo inicializador. Ele pode simplificar declarações nos casos em que o nome do tipo é bastante longo, como pode ser o caso de alguns nomes de classe (um exemplo de tipo que não pode ser indicado é o tipo de uma classe anônima, discutida no Capítulo 24). Além disso, a inferência de tipo de variáveis locais tornou-se uma parte comum do ambiente de programação contemporâneo. Sua inclusão em Java ajuda a manter a linguagem atualizada com as tendências em evolução no design da linguagem. Para suportar a inferência de tipo de variável local, o identificador caso-sensitivo **var** foi adicionado à Java como um *nome de tipo reservado.*

Para usar a inferência de tipo de variável local, a variável deve ser declarada com **var** como nome do tipo e deve incluir um inicializador. Por exemplo, no passado, seria declarada uma variável local **double** chamando **avg** que é inicializada com o valor 10.0, como mostrado aqui:

```
double avg=10.0;
```

Usando a inferência de tipo, essa declaração pode agora ser escrita assim:

```
var avg=10.0;
```

Em ambos os casos, **avg** será do tipo **double**. No primeiro caso, seu tipo é especificado explicitamente. No segundo, seu tipo é inferido como **double** porque o inicializador 10.0 é do tipo **double**.

Como mencionado, **var** foi adicionado como um identificador caso-sensitivo. Quando ele é usado como o nome do tipo no contexto de uma declaração de variável local, ele diz ao compilador para usar a inferência de tipo para determinar o tipo de variável que está sendo declarada com base no tipo do inicializador. Assim, na declaração da variável local, **var** é um espaço reservado para o tipo real e inferido. No entanto, quando usado na maioria dos outros lugares, **var** é simplesmente um identificador definido pelo usuário sem significado especial. Por exemplo, a seguinte declaração ainda é válida:

```
// Neste caso, var é simplesmente um identificador definido pelo usuário.
```

```
int var=1;
```

Neste caso, o tipo é explicitamente especifica como **int** e **var** é o nome da variável que está sendo declarada. Mesmo sendo um identificador caso-sensitivo, existem poucos lugares em que o uso de **var** seja correto. Ele não pode ser usado como o nome de uma classe, por exemplo.

O programa a seguir coloca a discussão anterior em ação:

```
// Uma demonstração simples de inferência de tipo de variável local.
class VarDemo
        {
                public static void main(String args[])
                        {
                                // Use a inferência de tipo para determinar o tipo de
                                // variável chamada avg. Neste caso, double é inferido.
                                var avg=10.0;
                                System.out.println("O valor de avg: " + avg);

                                // No contexto a seguir, var não é um identificador prede-
finido.
                                // Ele é simplesmente um nome de variável definido pelo
usuário.
                                int var=1;
                                System.out.println("O valor de var: " + var);

                                // Interessantemente, na sequencia a seguir, var é usado
                                // como ambos: o tipo de declaração e como um nome de
variável
                                // no inicializador.
                                var k=-var;
                                System.out.println("O valor de k: " + k);
                        }
        }
```

Aqui está a saída:

```
O valor de avg: 10.0
O valor de var: 1
O valor de k: -1
```

O exemplo anterior usa **var** para declarar apenas variáveis simples, mas não se pode usar **var** para declarar uma matriz. Por exemplo:

```
// Isto é válido.
var myArray=new int[10];
```

Observe que nem **var** e nem **myArray** têm colchetes. Em vez disso, o tipo de **myArray** é inferido para ser **int[]**. Além disso, *não* se pode usar colchetes no lado esquerdo de uma declaração **var**. Assim, ambas as declarações são inválidas:

```
// Errado!
var[] myArray=new int[10];
var myArray[]=new int[10];
```

Na primeira linha, é feita uma tentativa de colchete em **var**. Na segunda, é feita uma tentativa de colchete em **myArray**. Em ambos os casos, o uso de colchetes está incorreto porque o tipo é inferido do tipo do inicializador.

É importante enfatizar que **var** pode ser usado para declara uma variável apenas quando essa variável é inicializada. Por exemplo, a seguinte instrução está incorreta:

```
// Errado! É necessário um Inicializador.
var counter;
```

É necessário lembrar que **var** pode ser usado apenas para declarar variáveis locais. Ele não pode ser usado ao declarar variáveis de instância, parâmetros, ou tipos de retorno, por exemplo.

Embora a discussão e os exemplos tenham introduzido o básico da inferência de tipo de variável local, eles não mostram todo o seu poder. Como mostrado no Capítulo 7, a inferência de tipo de variável local é especialmente eficaz em encurtar declarações que envolver nomes longos de classe. Também pode ser usado com tipos genéricos (discutidos no Capítulo 14), em uma declaração **try**-with-resources (discutida no Capítulo 13), e com um loop **for** (discutido no Capítulo 5).

Algumas Restrições var

Além das restrições já mencionadas na discussão anterior, várias outras se aplicam ao uso de **var**. Apenas uma variável pode ser declarada por vez; uma variável não pode usar **null** como um inicializador; e a variável que está sendo declarada não pode ser usada pela expressão do inicializador. Embora possa-se declarar um tipo de matriz usando **var**, não se pode usar **var** com um inicializador de matriz. Por exemplo, isso é válido:

```
// Isso é válido.
var myArray=new int[10];
```

Mas isso não é:

// Errado

var myArray={1, 2, 3};

Como mencionado anteriormente, **var** não pode ser usado como o nome de uma classe. Ele também não pode ser usado como o nome de outros tipos de referência, incluindo uma interface, enumeração, ou anotação, ou como o nome de um parâmetro de tipo genérico, todos descritos mais adiantes neste livro. Aqui estão outras duas restrições relacionadas aos recursos Java descritos nos capítulos seguintes, mas mencionados aqui no interesse da integridade. A inferência de tipo de variável local não pode ser usada para declarar o tipo de exceção capturado por uma declaração **catch**. Além disso, nem expressões lambda nem referências a métodos podem ser usadas como inicializadores.

Nota: No momento da redação deste livro, a inferência de tipo de variável local é bastante nova e muitos dos exemplos deste livro estarão usando ambientes Java que não o suportam. Para que o maior número possível de exemplos de código seja compilado e executado para todos os leitores, a inferência de tipo de variável local não será usada pela maioria dos programas no restante desta edição do livro. O uso da sintaxe da declaração completa também deixa muito claro, de uma maneira geral, que tipo de variável está sendo criada, o que é importante para o código de exemplo. Obviamente, daqui para frente, deve-se considerar o uso de inferência de tipo de variável loca, quando apropriado, em seu próprio código.

Algumas Palavras Sobre Strings

Na discussão anterior sobre tipos de dados e matrizes, não foram mencionadas strings ou um tipo de dado string. Isso não ocorreu porque Java não suporta como um tipo – ele suporta. Só que o tipo string, chamado **String**, não é um tipo primitivo. Nem é simplesmente uma matriz de caracteres. Em vez disso, **String** define um objeto, e uma descrição completa dele requer uma compreensão de vários recursos relacionados a objetos. Como tal, será abordado mais adiantes neste livro, após a descrição dos objetos. No entanto, para que você possa usar sequências simples em programas de exemplo, a vreve introdução a seguir está em ordem.

O tipo **String** é usado para declarar variáveis string. Pode-se também declarar matrizes de strings. Uma constante de string entre aspas pode ser atribuída a uma variável **String**.

Uma variável do tipo **String** pode ser atribuída para outra variável do tipo **String**. Pode-se usar um objeto do tipo **String** como um argumento para **println()**. Por exemplo, considere o fragmento a seguir:

String str=”Isto é um teste”; System.out.println(str);

Aqui, **str** é um objeto do tipo **String**. É atribuída a sequência “Isto é um teste”. Essa string é mostrada pelo método **println()**.

Como mostrado mais adiante, os objetos **String** têm muitos recursos especiais e atributos que os tornam bastante poderosos e fáceis de usar. No entanto, para os próximos capítulos, eles serão usados apenas da forma mais simples.