Aprendendo Java

Aprendendo Java

Uma introdução à programação do mundo real com Java

Por Jideon Marques

Sob nenhuma circunstância qualquer culpa ou responsabilidade legal será responsabilizada contra o editor, ou autor, por quaisquer danos, reparações ou perdas monetárias devido às informações contidas neste livro, seja direta ou indiretamente. Notícia legal:

Aviso de isenção de responsabilidade:

Observe que as informações contidas neste documento são apenas para fins educacionais e de entretenimento. Todos os esforços foram realizados para apresentar informações precisas, atualizadas, confiáveis e completas. Nenhuma garantia de qualquer tipo é declarada ou implícita. Os leitores reconhecem que o autor não está envolvido na prestação de aconselhamento jurídico, financeiro, médico ou profissional. O conteúdo deste livro foi derivado de diversas fontes. Consulte um profissional licenciado antes de tentar qualquer técnica descrita neste livro.

Ao ler este documento, o leitor concorda que sob nenhuma circunstância o autor é responsável por quaisquer perdas, diretas ou indiretas, que sejam

incorridas como resultado do uso das informações contidas neste documento, incluindo, mas não limitado a, erros, omissões ou imprecisões.

Prefácio

Este livro apresenta a linguagem e o ambiente de programação Java. Seja você um desenvolvedor de software ou apenas alguém que usa a internet no dia a dia, sem dúvida já ouviu falar em Java. Sua chegada foi um dos desenvolvimentos mais emocionantes da história da web, e os aplicativos Java continuam a impulsionar os negócios na Internet. Java é, sem dúvida, a linguagem de programação mais popular do mundo, usada por milhões de desenvolvedores em quase todos os tipos de computador imagináveis. Java ultrapassou linguagens como C++ e Visual Basic em termos de demanda de desenvolvedores e se tornou a linguagem de fato para certos tipos de desenvolvimento – especialmente para serviços baseados na web. A maioria das universidades agora usa Java em seus cursos introdutórios junto com outras linguagens modernas importantes. Talvez você esteja usando este texto em uma de suas aulas agora mesmo!

Este livro fornece uma base completa em fundamentos e gramática de Java. Learning Java, Sexta Edição, tenta fazer jus ao seu nome mapeando a linguagem Java e suas bibliotecas de classes, técnicas de programação e expressões idiomáticas. Iremos nos aprofundar em áreas interessantes e pelo menos arranhar a superfície de outros tópicos populares.

Sempre que possível, fornecemos exemplos atraentes, realistas e divertidos e evitamos apenas catalogar recursos. Os exemplos são simples, mas sugerem o que

pode ser feito. Não desenvolveremos o próximo grande “aplicativo matador” nestas páginas, mas esperamos fornecer um ponto de partida para muitas horas de experimentação e ajustes inspirados que o levarão a desenvolver um você mesmo.

Quem deveria ler esse livro

Este livro é para profissionais de informática, estudantes, técnicos e hackers finlandeses. É para todos que precisam de experiência prática no uso de Java com o objetivo de criar aplicativos reais. Este livro também pode ser considerado um curso intensivo em programação orientada a objetos, threads e interfaces de usuário. Ao aprender sobre Java, você também aprenderá uma abordagem poderosa e prática para desenvolvimento de software, começando com uma compreensão profunda dos fundamentos de Java.

Superficialmente, Java se parece com C ou C++, então você terá uma pequena vantagem no uso deste livro se tiver alguma experiência com uma dessas linguagens.

Se não, não se preocupe. Em muitos aspectos, Java atua como linguagens mais dinâmicas, como Smalltalk e Lisp. O conhecimento de outra linguagem de programação orientada a objetos certamente ajudará, embora você possa ter que mudar algumas ideias e desaprender alguns hábitos. Java é consideravelmente mais simples que linguagens como C++ e Smalltalk. Se você aprender bem com exemplos concisos e experimentações pessoais, gostará deste livro.

Novos desenvolvimentos

Cobrimos todos os recursos importantes da versão mais recente de “suporte de longo prazo” do Java,

oficialmente chamada Java Standard Edition (SE) 21, OpenJDK 21. A Sun Microsystems (detentora do Java antes da Oracle) mudou o esquema de nomenclatura muitas vezes ao longo do anos. A Sun cunhou o termo Java 2 para cobrir os principais novos recursos introduzidos na versão 1.2 do Java e abandonou o termo JDK em favor do SDK. Com o sexto lançamento, a Sun pulou do Java versão 1.4 para o Java 5.0, mas reviveu o termo JDK e manteve sua convenção de numeração. Depois disso, tivemos o Java 6, o Java 7 e o Java 8. A partir do Java 9, a Oracle anunciou uma cadência de lançamento regular (acelerada). Novas versões são lançadas duas vezes por ano e estamos no Java 21 enquanto escrevemos em 2023.

Esta versão do Java reflete uma linguagem madura com alterações sintáticas ocasionais e atualizações em pacotes e bibliotecas. Tentamos capturar esses novos recursos e atualizar cada exemplo deste livro para refletir o estilo e as práticas recomendadas atuais do Java.

Novidade nesta edição (Java 15, 16, 17, 18, 19, 20, 21)

Esta edição do livro continua nossa tradição de retrabalho para ser o mais atualizado possível. Ele incorpora alterações de versões recentes do Java, do Java 15 ao Java 21

(acesso antecipado). Os novos tópicos desta edição incluem:

•

Threads virtuais que permitem ganhos de desempenho impressionantes em cenários que exigem muitos, muitos threads

- Nova cobertura de fluxos funcionais para processamento de dados
- Cobertura expandida de expressões lambda
- Exemplos atualizados e análises ao longo do livro
- Revise perguntas e exercícios para ajudar a reforçar os tópicos discutidos em cada capítulo

Usando este livro

Este livro está organizado da seguinte forma:

- Capítulos 1 e 2 fornecem uma introdução básica aos conceitos Java e um tutorial para fornecer um início rápido na programação Java.
- Capítulo 3 discute ferramentas fundamentais para desenvolvimento com Java (o compilador, o interpretador, jshell e o pacote de arquivos JAR).
- Capítulos 4 e 5 apresente os fundamentos da programação e, em seguida, descreva a própria linguagem Java,

começando com a sintaxe básica e

abrangendo classes e objetos, exceções, matrizes, enumerações, anotações e muito mais.

- Capítulo 6cobre exceções, erros e recursos de log nativos do Java.

- Capítulo 7cobre coleções junto com genéricos e tipos parametrizados em Java.

- Capítulo 8abrange processamento de texto, formatação, digitalização, utilitários de string e muitos dos principais utilitários da API.

- Capítulo 9cobre os recursos de thread integrados da linguagem, incluindo os novos threads virtuais.

- Capítulo 10cobre E/S de arquivo Java e o pacote NIO.

- Capítulo 11cobre técnicas de programação de funções em Java.

-

Capítulo 12cobre os fundamentos do desenvolvimento de interface gráfica do usuário (GUI) com Swing.

•

Capítulo 13cobre a comunicação de rede para clientes e servidores, bem como o acesso a recursos da web.

Se você é como nós, você não lê livros do início ao fim. Se você é realmente como nós, geralmente nem lê o prefácio. No entanto, caso você perceba isso a tempo, aqui estão algumas sugestões:

•

Se você já é um programador e só precisa aprender Java nos próximos cinco minutos, provavelmente está procurando exemplos. Você pode querer começar dando uma olhada no tutorial emCapítulo 2. Se isso não faz seu barco flutuar, você deve pelo menos olhar as informações emCapítulo 3, que explica como usar o compilador e o interpretador. Isso deve começar.

•

Capítulo 12discute os recursos gráficos e a arquitetura de componentes do Java. Você deve ler isto se estiver interessado em escrever aplicativos Java gráficos para desktop.

•

Capítulo 13é o lugar certo se você estiver interessado em escrever aplicativos de rede ou interagir com serviços baseados na web. A rede continua sendo uma das partes mais interessantes e importantes do Java.

Convenções utilizadas neste livro

As convenções de fontes usadas neste livro são bastante simples.

itálico é usado para:

•

Nomes de caminhos, nomes de arquivos e nomes de programas

•

Endereços da Internet, como nomes de domínio e URLs

•

Novos termos onde são definidos

•

Nomes de programas, compiladores, interpretadores, utilitários e comandos

•

Ênfase em pontos importantes

Largura constanteé usado para:

•

Qualquer coisa que possa aparecer em um programa Java, incluindo nomes de métodos, nomes de variáveis e nomes de classes

•

Tags que podem aparecer em um documento HTML ou XML

•

Palavras-chave, objetos e variáveis de ambiente

Largura constante em negritoé usado para:

•

Texto digitado pelo usuário na linha de comando ou em uma caixa de diálogo *Largura constante em itálico* é usado para:

•

Itens substituíveis no código

Dica

Este elemento significa uma dica ou sugestão.

Observação

Este elemento significa uma nota geral.

Aviso

Este elemento indica um aviso ou cuidado.

No corpo principal do texto, sempre usamos um par de parênteses vazios após o nome de um método para distinguir métodos de variáveis, classes e outras criaturas.

Nas listagens de fontes Java, seguimos as convenções de codificação usadas com mais frequência na comunidade

Java. Os nomes das classes começam com letras maiúsculas; nomes de variáveis e métodos começam com letras minúsculas. Todas as letras nos nomes das constantes são maiúsculas. Não usamos sublinhados para separar palavras em nomes longos; seguindo a prática comum, colocamos palavras individuais em maiúscula (depois da primeira) e juntamos as palavras. Por exemplo: thisIsAVariable, thisIsAMethod(), ThisIsAClass e THIS_IS_A_CONSTANT. Observe também que diferenciamos entre métodos estáticos e não estáticos quando nos referimos a eles. Ao contrário de alguns livros, nunca escrevemos Foo.bar() para significar o método bar() de Foo, a menos que bar() seja um método estático (paralelamente à sintaxe Java nesse caso).

Para listagens de fontes de programas de exemplo, a listagem começará com um comentário indicando o nome do arquivo relacionado (e o nome do método, se necessário):

```
//nome do arquivo: ch02/examples/HelloWorld.java

public static void main(String args[]) {

System.out.println("Olá, mundo!");

}
```

Você deve ficar à vontade para consultar o arquivo anotado em seu editor ou IDE. Nós encorajamos você a compilar e executar os exemplos. E encorajamos especialmente os ajustes!

Para trabalhar em jshell, sempre manteremos o prompt do jshell:

```
jshell> System.out.println("Olá, jshell!")

Olá, jshell!
```

Outros trechos sem nome de arquivo ou prompt jshell têm como objetivo ilustrar sintaxe e estrutura válidas ou apresentar uma abordagem hipotética para lidar com uma tarefa de programação. Essas listagens não decoradas não são necessariamente feitas para serem executadas, embora você seja sempre incentivado a criar suas próprias aulas para experimentar qualquer tópico do livro.

Usando exemplos de código

Este livro está aqui para ajudá-lo a realizar seu trabalho. Em geral, se um código de exemplo for oferecido com este livro, você poderá usá-lo em seus programas e documentação. Você não precisa entrar em contato conosco para obter permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que utilize vários trechos de código deste livro não requer permissão.

Capítulo 1. Uma Linguagem Moderna

Os maiores desafios e as oportunidades mais interessantes para os desenvolvedores de software hoje residem em aproveitar o poder das redes. As aplicações criadas hoje, qualquer que seja o âmbito ou público-alvo pretendido, quase certamente serão executadas em máquinas ligadas por uma rede global de recursos computacionais. A crescente importância das redes está a impor novas exigências às ferramentas existentes e a alimentar a procura de uma lista cada vez maior de tipos de aplicações completamente novos.

Como usuários, queremos um software que funcione — de forma consistente, em qualquer lugar, em qualquer plataforma — e que funcione bem com outros aplicativos. Queremos aplicações dinâmicas que tirem partido de um mundo conectado, capazes de aceder a fontes de informação díspares e distribuídas.

Queremos software verdadeiramente distribuído que possa ser estendido e atualizado sem problemas. Queremos aplicações inteligentes que possam percorrer a nuvem por nós, descobrindo informações e servindo como emissários eletrônicos. Já sabemos há algum tempo que tipo de software queremos, mas foi apenas nos últimos anos que começamos a obtê-lo.

O problema, historicamente, tem sido que as ferramentas para construir estas aplicações têm sido insuficientes. Os requisitos de velocidade e portabilidade têm sido, na sua maior parte, mutuamente exclusivos, e a segurança tem sido largamente ignorada ou mal compreendida. No passado, linguagens verdadeiramente portáveis eram volumosas, interpretadas e lentas. Essas linguagens eram populares tanto por sua funcionalidade de alto nível quanto por sua portabilidade. Linguagens rápidas geralmente forneciam velocidade vinculando-se a plataformas específicas, de modo que atendiam ao requisito de portabilidade apenas pela metade. Houve até algumas linguagens que incentivaram os programadores a escrever códigos melhores e mais seguros, mas elas eram principalmente ramificações das linguagens portáteis e sofriam dos mesmos problemas. Java é uma linguagem moderna que aborda todas essas três frentes: portabilidade, velocidade e segurança. É por isso que continua a ser uma linguagem dominante no mundo da programação quase três décadas após a sua introdução.

Entre em Java

A linguagem de programação Java foi projetada para ser uma linguagem de programação independente de máquina, segura o suficiente para atravessar redes e poderosa o suficiente para substituir código executável nativo. Java aborda as questões levantadas aqui e desempenhou um papel de destaque no crescimento da Internet, levando até onde estamos hoje.

Java se tornou a principal plataforma para aplicativos e serviços baseados na Web.

Esses aplicativos usam tecnologias como Java Servlet API, Java Web Services e muitos servidores e estruturas de aplicativos Java comerciais e de código aberto populares. A portabilidade e a velocidade do Java fazem dele a plataforma preferida para aplicativos de negócios modernos. Os servidores Java executados em plataformas Linux de código aberto estão no centro do mundo empresarial e financeiro hoje.

Inicialmente, a maior parte do entusiasmo pelo Java centrava-se nas suas capacidades de construção de aplicações incorporadas para a web, chamadas miniaplicativos. Mas no início, os applets e outras interfaces gráficas de usuário (GUIs) do lado do cliente escritas em Java eram limitados. Hoje, Java possui o Swing, um sofisticado kit de ferramentas para construção de GUIs. Este desenvolvimento permitiu que o Java se tornasse uma plataforma viável para o desenvolvimento de software de aplicação

tradicional do lado do cliente, embora muitos outros concorrentes tenham entrado neste campo concorrido.

Este livro mostrará como usar Java para realizar tarefas de programação do mundo real. Nos próximos capítulos apresentaremos uma ampla seleção de recursos Java, incluindo processamento de texto, rede, manipulação de arquivos e construção de aplicativos de desktop com Swing.

Origens do Java

As sementes do Java foram plantadas em 1990 pelo patriarca e pesquisador-chefe da Sun Microsystems, Bill Joy. Na época, a Sun estava competindo em um mercado de estações de trabalho relativamente pequeno, enquanto a Microsoft estava começando a dominar o mundo mais popular dos PCs baseados em Intel. Quando a Sun perdeu o barco na revolução dos PCs, Joy retirou-se para Aspen, Colorado, para trabalhar em pesquisas avançadas. Ele estava comprometido com a ideia de realizar tarefas complexas com software simples e fundou a apropriadamente chamada Sun Aspen Smallworks.

Dos membros originais da pequena equipe de programadores que Joy reuniu em Aspen, James Gosling será lembrado como o pai do Java. Gosling fez seu nome pela primeira vez no início dos anos 1980 como o autor de Gosling Emacs, a primeira versão do popular editor Emacs que foi escrito em C e executado em Unix. Gosling Emacs logo foi eclipsado por uma versão gratuita, GNU Emacs, escrita pelo designer original do Emacs. Naquela época, Gosling havia passado a projetar o Network extensible Window System (NeWS) da Sun, que disputou brevemente com o X

Window System para controle do desktop Unix GUI em 1987. Embora algumas pessoas argumentassem que o

NeWS era superior ao X, o NeWS perdido porque a Sun o manteve proprietário e não publicou o código-fonte, enquanto os principais desenvolvedores do X formaram o X Consortium e adotaram a abordagem oposta.

Projetar o NeWS ensinou a Gosling o poder de integrar uma linguagem expressiva com uma GUI de janelas com reconhecimento de rede. Também ensinou à Sun que a comunidade de programação da Internet acabará por se recusar a aceitar padrões proprietários, por melhores que sejam. O fracasso do NeWS lançou as sementes do esquema de licenciamento e do código aberto (se não exatamente de “código aberto”) do Java. Gosling trouxe o que aprendeu para o projeto nascente de Bill Joy em Aspen.

Em 1992, o trabalho no projeto levou à fundação da subsidiária da Sun, FirstPerson, Inc. Sua missão era levar a Sun ao mundo da eletrônica de consumo.

A equipe FirstPerson trabalhou no desenvolvimento de software para dispositivos de informação, como telefones celulares e assistentes pessoais digitais (PDAs). O objetivo era permitir a transferência de informações e aplicações em tempo real através de redes infravermelhas baratas e tradicionais baseadas em pacotes. As limitações de memória e largura de banda ditaram um código pequeno e eficiente. A natureza das aplicações também exigia que fossem seguras e robustas. Gosling e seus colegas de equipe começaram a programar em C++, mas logo se viram confundidos por uma linguagem que era muito complexa, pesada e insegura para a tarefa. Eles decidiram

começar do zero, e Gosling começou a trabalhar em algo que ele apelidou de “C++

menos menos”.

Com o naufrágio do Apple Newton (o primeiro computador portátil da Apple), tornou-se evidente que o navio do PDA ainda não havia chegado, então a Sun transferiu os esforços da FirstPerson para a TV interativa (ITV). A linguagem de programação escolhida para decodificadores ITV seria a quase ancestral do Java, uma linguagem chamada Oak. Mesmo com sua elegância e capacidade de fornecer interatividade segura, a Oak não conseguiu salvar a causa perdida do ITV. Os clientes não queriam isso e a Sun logo abandonou o conceito.

Naquela época, Joy e Gosling se reuniram para decidir uma nova estratégia para sua linguagem inovadora. Era 1993 e a explosão de interesse pela web apresentou uma nova oportunidade. Oak era pequeno, seguro, independente de arquitetura e orientado a objetos. Na verdade, esses também são alguns dos requisitos para uma linguagem de programação universal e compatível com a Internet. A Sun rapidamente mudou o foco e, com uma pequena reformulação, Oak tornou-se Java.

Crescendo

Não seria um exagero dizer que o Java (e seu pacote focado no desenvolvedor, o Java Development Kit, ou JDK) pegou como um incêndio. Mesmo antes de seu primeiro lançamento oficial, quando o Java ainda não era um produto, quase todos os principais players da indústria aderiram ao movimento Java. Os licenciados Java incluíam Microsoft, Intel, IBM e praticamente todos os principais fornecedores de hardware e software. No entanto, mesmo com todo esse suporte, o Java sofreu muitos golpes e passou por algumas dificuldades de crescimento durante seus primeiros anos.

Uma série de quebras de contrato e ações judiciais antitruste entre a Sun e a Microsoft sobre a distribuição do Java e seu uso no Internet Explorer dificultou sua implantação no sistema operacional de desktop mais comum do mundo – o Windows. O

envolvimento da Microsoft com Java também se tornou o foco de um processo federal maior sobre práticas anticompetitivas graves na empresa. O depoimento no tribunal revelou que a gigante do software tentou minar o Java introduzindo

incompatibilidades em sua versão da linguagem. Enquanto isso, a Microsoft introduziu sua própria linguagem derivada de Java chamada C# (C-sharp) como parte de sua iniciativa .NET e retirou o Java da inclusão no Windows. C# tornou-se uma linguagem muito boa por si só, desfrutando de mais inovações nos últimos anos do que Java.

Mas o Java continua a se espalhar em uma ampla variedade de plataformas. À medida que começarmos a examinar a arquitetura Java, você verá que muito do que há de interessante em Java vem do ambiente de máquina virtual independente no qual os aplicativos Java são executados. Java foi cuidadosamente projetado para que esta arquitetura de suporte possa ser implementada tanto em software, para plataformas computacionais existentes, quanto em hardware customizado. Implementações de hardware de Java são usadas em alguns cartões inteligentes e outros sistemas embarcados. Você pode até comprar dispositivos “vestíveis”, como anéis e etiquetas

de identificação, que possuem interpretadores Java incorporados. Implementações de software de Java estão

disponíveis para todas as plataformas de computadores modernas, até mesmo para dispositivos de computação portáteis. Hoje, uma ramificação da plataforma Java é a base do sistema operacional Android, do Google, que alimenta bilhões de telefones e outros dispositivos móveis.

Em 2010, a Oracle Corporation comprou a Sun Microsystems e tornou-se

administradora da linguagem Java. Em um início de mandato um tanto difícil, a Oracle processou o Google pelo uso da linguagem Java no Android e perdeu. Em julho de 2011 a Oracle lançou o Java Standard Edition 71uma versão Java significativa que incluía um novo pacote de E/S. Em 2017, o Java 9 introduziu módulos para resolver alguns problemas antigos relacionados à forma como os aplicativos Java eram compilados, distribuídos e executados. O Java 9 também iniciou um rápido processo de atualização, fazendo com que algumas versões do Java fossem designadas como

“suporte de longo prazo” e o restante como versões padrão de curto prazo. (Mais sobre estas e outras versões em“Um roteiro Java”.) A Oracle continua liderando o desenvolvimento Java; no entanto, ele também bifurcou o mundo Java ao migrar o principal ambiente de implantação Java para uma licença comercial cara, ao mesmo tempo em que oferece uma opção OpenJDK subsidiária gratuita que mantém a acessibilidade que muitos desenvolvedores amam e esperam.

Uma máquina virtual

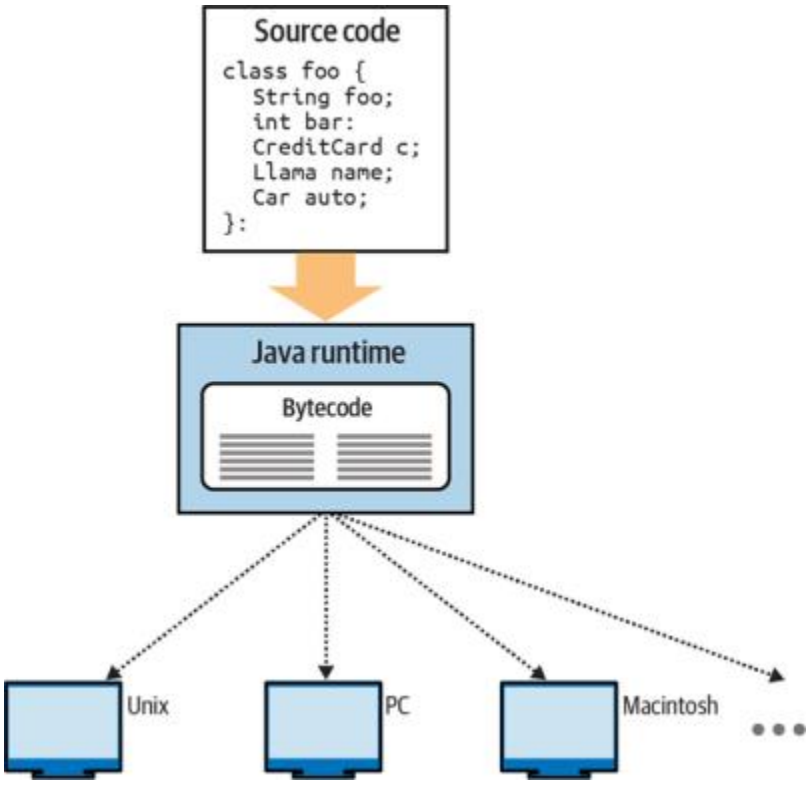
Antes de prosseguirmos, é útil saber um pouco mais sobre o ambiente que Java precisa para fazer sua mágica. Tudo bem se você não entender tudo o que abordaremos nas próximas seções. Qualquer termo desconhecido que você possa ver receberá o devido valor nos capítulos posteriores. Queremos apenas fornecer uma visão geral do ecossistema Java. No centro desse ecossistema está a Java Virtual Machine (JVM).

Java é uma linguagem compilada e interpretada. O código-fonte Java é transformado em instruções binárias simples, bem como o código de máquina de microprocessador comum. No entanto, enquanto a fonte C ou C++ é reduzida a instruções nativas para um modelo específico de processador, a fonte Java é compilada em um formato universal – instruções para a máquina virtual conhecida como bytecode.

O bytecode Java é executado por um interpretador de tempo de execução Java. O

sistema de tempo de execução executa todas as atividades normais de um processador de hardware, mas o faz em um ambiente virtual seguro. Ele executa um conjunto de instruções baseado em pilha e gerencia a memória como um sistema operacional. Ele cria e manipula tipos de dados primitivos e carrega e invoca blocos de código recém-referenciados. Mais importante ainda, ele faz tudo isso de acordo com uma especificação aberta estritamente definida que pode ser implementada por qualquer pessoa que queira produzir uma máquina virtual compatível com Java.

Juntas, a máquina virtual e a definição da linguagem fornecem uma especificação completa. Não há recursos da linguagem Java base indefinidos ou dependentes de



implementação. Por exemplo, Java especifica os tamanhos e propriedades matemáticas de todos os seus tipos de dados primitivos, em vez de deixar isso para a implementação da plataforma.

O interpretador Java é relativamente leve e pequeno; pode ser implementado em qualquer forma que seja desejável para uma plataforma específica. O interpretador pode ser executado como um aplicativo separado ou pode ser incorporado em outro software, como um navegador da web. Juntos, isso significa que o código Java é implicitamente portátil. O mesmo bytecode de aplicativo Java pode ser executado em qualquer plataforma que forneça um ambiente de tempo de execução Java, conforme mostrado emFigura 1-1. Você não precisa produzir versões alternativas do seu aplicativo para diferentes plataformas e não precisa distribuir o código-fonte aos usuários finais.

Figura 1-1. O ambiente de tempo de execução Java

A unidade fundamental do código Java é a classe. Como em outras linguagens orientadas a objetos, as classes são componentes de aplicativos pequenos e modulares que contêm código e dados executáveis. As classes Java compiladas são distribuídas em um formato binário universal que contém bytecode Java e outras informações de classe. As aulas podem ser mantidas discretamente e armazenadas em arquivos ou arquivos localmente ou em um servidor de rede. As classes são localizadas e carregadas dinamicamente em tempo de execução conforme a necessidade de um aplicativo.

Além do sistema de tempo de execução específico da plataforma, Java possui diversas classes fundamentais que contêm métodos dependentes da arquitetura. Esses métodos nativos servem como porta de entrada entre a máquina virtual Java e o mundo real. Eles são implementados em uma linguagem compilada nativamente na plataforma host e fornecem acesso de baixo nível a recursos como rede, sistema de janelas e sistema de arquivos host. A grande maioria do Java, entretanto, é escrita no próprio Java – inicializado a partir dessas partes básicas – e, portanto, é portátil. Isso inclui ferramentas Java importantes, como o compilador Java, também escrito em Java e, portanto, disponível em todas as plataformas Java exatamente da mesma maneira, sem portabilidade.

Historicamente, os intérpretes foram considerados lentos, mas Java não é uma linguagem interpretada tradicional. Além de compilar o código-fonte até o bytecode portátil, o Java também foi cuidadosamente projetado para que as implementações de software do sistema de tempo de execução possam otimizar ainda

mais seu desempenho, compilando o bytecode para o código de máquina nativo em tempo real.

Isso é chamado de compilação dinâmica ou just-in-time (JIT). Com a compilação JIT, o código Java pode ser executado tão rapidamente quanto o código nativo e manter sua transportabilidade e segurança.

Esse recurso JIT é um ponto frequentemente mal compreendido entre aqueles que desejam comparar o desempenho do idioma. Há apenas uma penalidade intrínseca de desempenho que o código Java compilado sofre em tempo de execução por questões de segurança e design de máquina virtual: verificação de limites de array. Todo o resto pode ser otimizado para código nativo, assim como acontece com uma linguagem compilada estaticamente. Indo além disso, a linguagem Java inclui mais informações estruturais do que muitas outras linguagens, proporcionando mais tipos de otimizações. Lembre-se também que essas otimizações podem ser feitas em tempo de execução, levando em consideração o comportamento e as características reais da aplicação. O que pode ser feito em tempo de compilação que não pode ser feito melhor em tempo de execução? Bem, há uma compensação: o tempo.

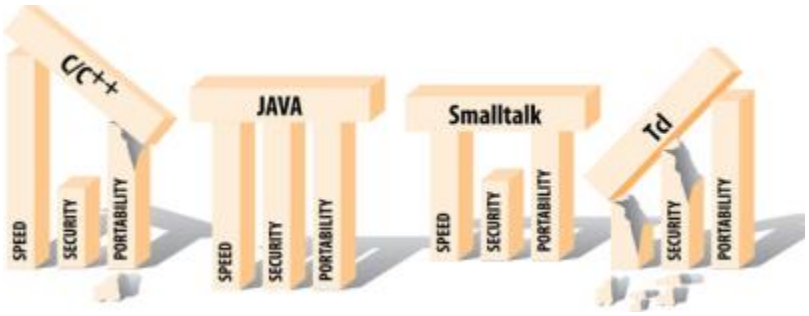
O problema com uma compilação JIT tradicional é que a otimização do código leva tempo. Embora um compilador JIT possa produzir resultados decentes, ele pode sofrer latência significativa quando o aplicativo é inicializado. Geralmente, isso não é um problema para aplicativos do lado do servidor de longa execução, mas é um problema sério para softwares do lado do cliente e aplicativos executados em dispositivos menores com recursos limitados. Para resolver isso, a tecnologia do compilador Java, chamada HotSpot, usa um truque chamado

compilação adaptativa. Se você observar o que os programas realmente gastam seu tempo fazendo, verá que eles gastam quase todo o tempo executando uma parte relativamente pequena do código repetidas vezes. O pedaço de código executado repetidamente pode representar apenas uma pequena fração do programa total, mas seu comportamento determina o desempenho geral do programa. A compilação adaptativa permite que o tempo de execução Java aproveite novos tipos de otimizações que simplesmente não podem ser feitas em uma linguagem compilada estaticamente, daí a afirmação de que o código Java pode ser executado mais rápido que C/C++ em alguns casos.

Para aproveitar essa capacidade adaptativa, o HotSpot começa como um interpretador de bytecode Java normal, mas com uma diferença: ele mede (perfilia) o código à medida que ele é executado para ver quais partes estão sendo executadas repetidamente. Depois de saber quais partes do código são cruciais para o desempenho, o HotSpot compila essas seções em um código de máquina nativo ideal.

Como ele compila apenas uma pequena parte do programa em código de máquina, ele pode gastar o tempo necessário para otimizar essas partes. O resto do programa pode não precisar ser compilado – apenas interpretado – economizando memória e tempo.

Na verdade, o Java VM pode ser executado em um dos dois modos: cliente e servidor, que determinam se ele enfatiza o tempo de inicialização rápido e a conservação de



memória ou o desempenho total. A partir do Java 9, você também pode colocar a compilação antecipada (AOT) em uso se minimizar o tempo de inicialização do aplicativo for realmente importante.

Uma pergunta natural a ser feita neste momento é: por que jogar fora todas essas boas informações de criação de perfil sempre que um aplicativo é encerrado? Bem, a Sun abordou parcialmente esse tópico com o lançamento do Java 5.0 por meio do uso de classes compartilhadas somente leitura que são armazenadas persistentemente em um formato otimizado. Isso reduziu significativamente o tempo de inicialização e a sobrecarga de execução de muitos aplicativos Java em uma determinada máquina. A tecnologia para fazer isso é complexa, mas a ideia é simples: otimize as partes do programa que precisam ser executadas rapidamente e não se preocupe com o resto.

É claro que “o resto” contém código que poderia ser otimizado ainda mais. Em 2022, o OpenJDKProjeto Leidencomeçou com a intenção de reduzir ainda mais o tempo de inicialização, minimizar o grande tamanho dos aplicativos Java e reduzir o tempo que leva para que todas as otimizações mencionadas anteriormente tenham efeito total.

Os mecanismos propostos pelo Projeto Leyden são bastante complexos, por isso não iremos discuti-los neste

livro. Mas queríamos destacar o trabalho constante no desenvolvimento e melhoria do Java e de seu ecossistema. Mesmo cerca de 30 anos após sua estreia, Java continua sendo uma linguagem moderna.

Java comparado com outras linguagens

Os desenvolvedores de Java basearam-se em muitos anos de experiência em programação com outras linguagens na escolha de recursos. Vale a pena reservar um momento para comparar o Java em alto nível com algumas dessas linguagens, tanto para o benefício daqueles com outra experiência em programação quanto para os recém-chegados que precisam colocar as coisas em contexto. Embora este livro espere que você tenha algum conforto com computadores e aplicativos de software em um sentido genérico, não esperamos que você tenha conhecimento de nenhuma linguagem de programação específica. Quando nos referimos a outras línguas a título de comparação, esperamos que os comentários sejam autoexplicativos.

Atualmente, pelo menos três pilares são necessários para apoiar uma linguagem de programação universal: portabilidade, velocidade e segurança.Figura 1-2mostra como o Java se compara a algumas das linguagens que eram populares quando foi criado.

Figura 1-2. Linguagens de programação comparadas

Você pode ter ouvido que Java é muito parecido com C ou C++, mas isso não é verdade, exceto em um nível superficial. Ao observar o código Java pela primeira vez, você verá

que a sintaxe básica se parece com C ou C++. Mas é aí que as semelhanças terminam.

Java não é de forma alguma um descendente direto de C ou de um C++ de próxima geração. Se você comparar os recursos da linguagem, verá que Java, na verdade, tem mais em comum com linguagens altamente dinâmicas, como Smalltalk e Lisp. Na verdade, a implementação do Java está tão distante do C nativo quanto você pode imaginar.

Se você estiver familiarizado com o cenário atual da linguagem, notará que C#, uma linguagem popular, está faltando nesta comparação. C# é em grande parte a resposta da Microsoft ao Java, reconhecidamente com uma série de sutilezas em camadas.

Dados seus objetivos e abordagem de design comuns (como o uso de uma máquina virtual, bytecode e sandbox), as plataformas não diferem substancialmente em termos de velocidade ou características de segurança. C# é mais ou menos tão portátil quanto Java. Assim como o Java, o C# se baseia muito na sintaxe C, mas é, na verdade, um parente mais próximo das linguagens dinâmicas. A maioria dos desenvolvedores Java acha relativamente fácil aprender C# e vice-versa. A maior parte do tempo que você gastará mudando de um para outro será aprendendo a biblioteca padrão.

As semelhanças superficiais com essas linguagens são dignas de nota, no entanto. Java toma muito emprestado da sintaxe C e C++, então você verá construções de linguagem concisas, incluindo uma abundância de chaves e ponto e vírgula. Java segue a filosofia C de que uma boa linguagem deve ser compacta; em outras palavras, deve ser suficientemente pequeno e regular para que um programador possa manter todas as suas capacidades na cabeça de uma só vez. Assim como C é extensível com bibliotecas, pacotes de classes Java

podem ser adicionados aos componentes principais da linguagem para estender seu vocabulário.

C tem sido bem-sucedido porque fornece um ambiente de programação

razoavelmente repleto de recursos, com alto desempenho e um grau aceitável de portabilidade. Java também tenta equilibrar funcionalidade, velocidade e portabilidade, mas o faz de uma maneira muito diferente. C troca funcionalidade por portabilidade; Java inicialmente trocou velocidade por portabilidade. Java também aborda questões de segurança que C não aborda (embora em sistemas modernos muitas dessas preocupações sejam agora abordadas no sistema operacional e no hardware).

Linguagens de script como Perl, Python e Ruby continuam populares. Não há razão para que uma linguagem de script não seja adequada para aplicativos seguros em rede. Mas a maioria das linguagens de script não são adequadas para programação séria e em grande escala. A atração das linguagens de script é que elas são dinâmicas; são ferramentas poderosas para um desenvolvimento rápido. Algumas linguagens de script, como Tcl (mais popular quando o Java estava sendo desenvolvido), também ajudam os programadores a realizar tarefas específicas, como a criação rápida de interfaces gráficas, que linguagens de uso mais geral consideram difíceis de manejar.

As linguagens de script também são altamente portáveis, embora no nível do código-fonte.

Não deve ser confundido com Java, JavaScript é uma linguagem de script baseada em objetos desenvolvida

originalmente pela Netscape para o navegador da web. Ele serve como uma linguagem residente no navegador da web para aplicativos dinâmicos, interativos e baseados na web. JavaScript leva o nome de sua integração e semelhanças com Java, mas a comparação realmente termina aí. Existem, no entanto, aplicações significativas de JavaScript fora do navegador, como Node.js,2e sua popularidade continua crescendo para desenvolvedores em diversos campos. Para obter mais informações sobre JavaScript.

O problema com linguagens de script é que elas são bastante casuais em relação à estrutura do programa e à digitação de dados. Eles têm sistemas de tipos simplificados e geralmente não fornecem escopo sofisticado de variáveis e funções.

Essas características os tornam menos adequados para a construção de aplicações modulares grandes. A velocidade é outro problema com linguagens de script; a natureza de alto nível, geralmente interpretada pela fonte, dessas linguagens muitas vezes as torna bastante lentas.

Os defensores de linguagens de script individuais discordariam de algumas dessas generalizações e, sem dúvida, estariam certos em alguns casos. As linguagens de script melhoraram nos últimos anos – especialmente o JavaScript, que teve uma enorme quantidade de pesquisas investidas em seu desempenho. Mas a compensação fundamental é inegável: as linguagens de script nasceram como alternativas flexíveis e menos estruturadas às linguagens de programação de sistemas e geralmente não são ideais para projetos grandes ou complexos por vários motivos.

Java oferece algumas das vantagens essenciais de uma linguagem de script: é altamente dinâmica e possui os benefícios adicionais de uma linguagem de nível inferior. Java possui um poderoso pacote de expressões regulares que compete com Perl para trabalhar com texto. Ele também possui recursos de linguagem que agilizam a codificação com coleções, listas de argumentos variáveis, importações estáticas de métodos e outros recursos sintáticos que o tornam mais conciso.

O desenvolvimento incremental com componentes orientados a objetos, aliado à simplicidade do Java, possibilita desenvolver aplicações rapidamente e alterá-las facilmente. Estudos descobriram que desenvolver em Java é mais rápido do que em C

ou C++, estritamente baseado nos recursos da linguagem.3Java também vem com uma grande base de classes principais padrão para tarefas comuns, como construção de GUIs e manipulação de comunicações de rede. Maven Central é um recurso externo com uma enorme variedade de bibliotecas e pacotes que podem ser rapidamente agrupados em seu ambiente para ajudá-lo a resolver todos os tipos de novos problemas de programação. Junto com esses recursos, Java tem as vantagens de escalabilidade e engenharia de software de linguagens mais estáticas. Ele fornece uma estrutura segura para construir estruturas de nível superior (e até mesmo outras linguagens).

Como já dissemos, Java é semelhante em design a linguagens como Smalltalk e Lisp.

No entanto, essas linguagens foram usadas principalmente como veículos de pesquisa, e não para o

desenvolvimento de sistemas em larga escala. Uma razão é que essas

linguagens nunca desenvolveram uma ligação portátil padrão para serviços do sistema operacional, como a biblioteca padrão C ou as classes principais Java.

Smalltalk é compilado em um formato de bytecode interpretado e pode ser compilado dinamicamente em código nativo em tempo real, assim como Java. Mas Java melhora o design usando um verificador de bytecode para garantir a exatidão do código Java compilado. Este verificador dá ao Java uma vantagem de desempenho sobre o Smalltalk porque o código Java requer menos verificações de tempo de execução. O

verificador de bytecode do Java também ajuda com questões de segurança, algo que o Smalltalk não aborda.

Ao longo do restante deste capítulo, apresentaremos uma visão panorâmica da linguagem Java. Explicaremos o que há de novo e o que não há de novo em Java e por quê.

Segurança do Design

Você sem dúvida já ouviu falar muito sobre o fato de que Java foi projetado para ser uma linguagem segura. Mas o que queremos dizer com seguro? A salvo de quê ou de quem? Os recursos de segurança Java que mais atraem a atenção são aqueles que tornam possíveis novos tipos de software dinamicamente portáveis. Java fornece várias camadas de proteção contra códigos perigosamente falhos, bem como contra coisas mais prejudiciais, como vírus e cavalos de Tróia. Na próxima seção, veremos como a arquitetura da máquina virtual Java avalia a segurança do código antes de sua execução e como o

carregador de classes Java (o mecanismo de carregamento de bytecode do interpretador Java) constrói uma barreira em torno de classes não confiáveis. Esses recursos fornecem a base para políticas de segurança de alto nível que podem permitir ou proibir vários tipos de atividades, aplicativo por aplicativo.

Nesta seção, porém, veremos alguns recursos gerais da linguagem de programação Java. Talvez mais importante do que os recursos de segurança específicos, embora muitas vezes esquecidos no debate sobre segurança, seja a segurança que o Java fornece ao resolver problemas comuns de design e programação. O objetivo do Java é ser o mais seguro possível contra os erros simples que os programadores cometem, bem como contra aqueles que herdamos de software legado. O objetivo do Java tem sido manter a linguagem simples, fornecer ferramentas que demonstrem sua utilidade e permitir que os usuários construam recursos mais complicados sobre a linguagem quando necessário.

Simplifique, simplifique, simplifique…

Com Java, a simplicidade impera. Como o Java começou do zero, ele evitou recursos que se mostraram confusos ou controversos em outras linguagens. Por exemplo, Java não permite sobrecarga de operadores definida pelo programador (o que, em algumas linguagens, permite que os programadores redefinam os significados de símbolos básicos como + e –). Java não possui um pré-processador de código-fonte, portanto, não possui recursos como macros, instruções #define ou compilação de código-fonte

condicional. Essas construções existem em outras linguagens principalmente para suportar dependências

de plataforma; portanto, nesse sentido, elas não deveriam ser necessárias em Java. A compilação condicional também é comumente usada para depuração, mas as sofisticadas otimizações de tempo de execução do Java e recursos como asserções resolvem o problema com mais elegância.4

Java fornece uma estrutura de pacote bem definida para organizar arquivos de classe.

O sistema de pacotes permite que o compilador lide com algumas das funcionalidades do utilitário make tradicional (uma ferramenta para construir executáveis a partir do código-fonte). O compilador também pode trabalhar diretamente com classes Java compiladas porque todas as informações de tipo são preservadas; não há necessidade de arquivos de “cabeçalho” de origem estranhos, como em C/C++. Tudo isso significa que o código Java requer menos contexto para ser lido. Na verdade, às vezes você pode achar mais rápido olhar o código-fonte Java do que consultar a documentação da classe.

Java também adota uma abordagem diferente para alguns recursos estruturais que têm sido problemáticos em outras linguagens. Por exemplo, Java suporta apenas uma única hierarquia de classes de herança (cada classe pode ter apenas uma classe “pai”), mas permite herança múltipla de interfaces. Uma interface, como uma classe abstrata em C++, especifica o comportamento de um objeto sem definir sua implementação. É

um mecanismo muito poderoso que permite ao desenvolvedor definir um “contrato”

para o comportamento do objeto que pode ser usado e referido independentemente de qualquer implementação específica do objeto. As interfaces em Java eliminam a necessidade de herança múltipla de classes e os problemas associados.

Como você verá emCapítulo 4, Java é uma linguagem de programação bastante simples e elegante, e isso ainda é uma grande parte de seu apelo.

Tipo de segurança e associação de método

Um atributo de uma linguagem é o tipo de verificação de tipo que ela usa. Geralmente, as linguagens são categorizadas como estáticas ou dinâmicas, o que se refere à quantidade de informações sobre variáveis conhecidas em tempo de compilação versus o que é conhecido enquanto o aplicativo está em execução.

Em uma linguagem estritamente estaticamente tipada, como C ou C++, os tipos de dados são gravados em pedra quando o código-fonte é compilado. O compilador se beneficia disso por ter informações suficientes para detectar muitos tipos de erros antes que o código seja executado. Por exemplo, o compilador não permitiria armazenar um valor de ponto flutuante em uma variável inteira. O código não requer verificação de tipo de tempo de execução, portanto pode ser compilado para ser pequeno e rápido. Mas as linguagens de tipo estaticamente são inflexíveis. Eles não oferecem suporte a coleções tão naturalmente quanto linguagens com verificação dinâmica de tipo e tornam impossível para um aplicativo importar novos tipos de dados com segurança enquanto está em execução.

Em contraste, uma linguagem dinâmica como Smalltalk ou Lisp possui um sistema de tempo de execução que gerencia os tipos de objetos e executa a verificação de tipo necessária enquanto um aplicativo está em execução. Esses tipos de linguagens permitem um comportamento mais complexo e são, em muitos aspectos, mais poderosos. No entanto, eles também são geralmente mais lentos, menos seguros e mais difíceis de depurar.

As diferenças nos idiomas foram comparadas às diferenças entre os tipos de automóveis.5Linguagens de tipo estaticamente como C++ são análogas a um carro esportivo: razoavelmente seguras e rápidas, mas úteis apenas se você estiver dirigindo em uma estrada bem pavimentada. Linguagens altamente dinâmicas como Smalltalk são mais parecidas com um veículo off-road: elas proporcionam mais liberdade, mas podem ser um tanto pesadas. Pode ser divertido (e às vezes mais rápido) rugir pelo sertão, mas você também pode ficar preso em uma vala ou ser atacado por ursos.

Outro atributo de uma linguagem é a maneira como ela vincula chamadas de métodos às suas definições. Em uma linguagem estática como C ou C++, as definições de métodos são normalmente vinculadas em tempo de compilação, a menos que o programador especifique o contrário. Linguagens como Smalltalk, por outro lado, são chamadas de ligação tardia porque localizam as definições de métodos

dinamicamente em tempo de execução. A vinculação antecipada é importante por motivos de desempenho; ele permite que um aplicativo seja executado sem a sobrecarga incorrida pela pesquisa de métodos em tempo de execução. Mas a vinculação tardia é mais

flexível. Também é necessário em uma linguagem orientada a objetos onde novos tipos podem ser carregados dinamicamente e somente o sistema de tempo de execução pode determinar qual método executar.

Java oferece alguns dos benefícios do C++ e do Smalltalk; é uma linguagem de ligação tardia e digitada estaticamente. Cada objeto em Java possui um tipo bem definido que é conhecido em tempo de compilação. Isso significa que o compilador Java pode fazer o mesmo tipo de verificação de tipo estático e análise de uso que C++. Como resultado, você não pode atribuir um objeto ao tipo errado de variável ou chamar métodos inexistentes em um objeto. O compilador Java vai ainda mais longe e evita que você use variáveis não inicializadas e crie instruções inacessíveis (vejaCapítulo 4).

No entanto, Java também é totalmente digitado em tempo de execução. O sistema de tempo de execução Java monitora todos os objetos e possibilita determinar seus tipos e relacionamentos durante a execução. Isso significa que você pode inspecionar um objeto em tempo de execução para determinar o que ele é. Ao contrário de C ou C++, o sistema de tempo de execução Java verifica conversões de um tipo de objeto para outro e é possível usar novos tipos de objetos carregados dinamicamente com um certo grau de segurança de tipo. E como Java usa ligação tardia, é possível escrever código que substitua algumas definições de método em tempo de execução.

Desenvolvimento Incremental

Java carrega consigo todos os tipos de dados e informações de assinatura de método, desde seu código-

fonte até seu formato de bytecode compilado. Isso significa que as classes Java podem ser desenvolvidas de forma incremental. Seu próprio código-fonte Java também pode ser compilado com segurança com classes de outras fontes que seu compilador nunca viu. Em outras palavras, você pode escrever um novo código que faça referência a arquivos de classe binária sem perder a segurança de tipo que você ganha por ter o código-fonte.

Java não sofre do problema da “classe base frágil”. Em linguagens como C++, a implementação de uma classe base pode ser efetivamente congelada porque possui muitas classes derivadas; alterar a classe base pode exigir a recompilação de todas as classes derivadas. Este é um problema especialmente difícil para desenvolvedores de bibliotecas de classes. Java evita esse problema localizando campos dinamicamente dentro das classes. Contanto que uma classe mantenha uma forma válida de sua estrutura original, ela pode evoluir sem quebrar outras classes derivadas dela ou usadas.

Gerenciamento Dinâmico de Memória

Algumas das diferenças mais importantes entre Java e linguagens de nível inferior (como C ou C++) envolvem como Java gerencia a memória. Java elimina referências ad hoc a áreas arbitrárias da memória (ponteiros, em outras linguagens) e adiciona algumas estruturas de dados de alto nível à linguagem. Java também limpa objetos não utilizados (um processo conhecido como coleta de lixo) de forma eficiente e automática. Esses recursos eliminam muitos problemas de segurança, portabilidade e otimização que de outra forma seriam intransponíveis.

A coleta de lixo por si só salvou inúmeros programadores da maior fonte de erros de programação em C ou C++: alocação e desalocação explícita de memória. Além de manter objetos na memória, o sistema de tempo de execução Java controla todas as referências a esses objetos. Quando um objeto não está mais em uso, Java o remove automaticamente da memória. Você pode, na maioria das vezes, simplesmente ignorar objetos que não usa mais, com a confiança de que o intérprete os limpará no momento apropriado.

Java usa um coletor de lixo sofisticado que é executado em segundo plano, o que significa que a maior parte da coleta de lixo ocorre durante tempos ociosos: entre pausas de E/S, cliques do mouse ou toques no teclado. Alguns sistemas de tempo de execução, como o HotSpot, possuem coleta de lixo mais avançada que pode diferenciar os padrões de uso de objetos (como vida curta versus vida longa) e otimizar sua coleta. O tempo de execução Java agora pode se ajustar automaticamente para a distribuição ideal de memória para diferentes tipos de aplicativos com base em seu comportamento. Com esse tipo de perfil de tempo de execução, o gerenciamento automático de memória pode ser muito mais rápido do que os recursos gerenciados

mais diligentemente pelos programadores, algo que alguns programadores da velha escola ainda acham difícil de acreditar.

Dissemos que Java não possui ponteiros. A rigor, esta afirmação é verdadeira, mas também é enganosa. O que Java fornece são referências – um tipo mais seguro de ponteiro. Uma referência é um identificador fortemente tipado para um objeto. Todos os objetos em Java, com exceção dos tipos numéricos primitivos, são acessados

por meio de referências. Você pode usar referências para construir todos os tipos normais de estruturas de dados que um programador C estaria acostumado a construir com ponteiros, como listas vinculadas, árvores e assim por diante. A única diferença é que, com referências, você deve fazê-lo de maneira segura.

As referências em Java não podem ser alteradas da mesma forma que você altera ponteiros em linguagens como C. Uma referência é algo atômico; você não pode manipular o valor de uma referência, exceto atribuindo-a a um objeto. As referências são passadas por valor e você não pode fazer referência a um objeto por meio de mais de um único nível de indireção. Proteger referências é um dos aspectos mais fundamentais da segurança Java. Isso significa que o código Java deve seguir as regras; não pode espiar lugares que não deveria para contornar essas regras.

Finalmente, devemos mencionar que arrays (essencialmente listas indexadas) em Java são objetos verdadeiros e de primeira classe. Eles podem ser alocados e atribuídos dinamicamente como outros objetos. As matrizes conhecem seu próprio tamanho e tipo. Embora você não possa definir ou subclassificar diretamente classes de array, elas têm um relacionamento de herança bem definido com base no relacionamento de seus tipos base. Ter arrays verdadeiros na linguagem alivia grande parte da necessidade de aritmética de ponteiros, como a usada em C ou C++.

Manipulação de erros

As raízes do Java estão em dispositivos de rede e sistemas embarcados. Para essas aplicações, é

importante ter um gerenciamento de erros robusto e inteligente. Java possui um mecanismo poderoso para lidar com exceções, semelhante ao das implementações mais recentes de C++. As exceções fornecem uma maneira mais natural e elegante de lidar com erros. As exceções permitem separar o código de tratamento de erros do código normal, o que torna os aplicativos mais limpos e legíveis.

Quando ocorre uma exceção, ela faz com que o fluxo de execução do programa seja transferido para um bloco de código “catch” pré-designado. A exceção carrega consigo um objeto que contém informações sobre a situação que causou o problema. O

compilador Java exige que um método declare as exceções que pode gerar ou capture e lide com elas ele mesmo. Isso promove as informações de erro ao mesmo nível de importância que os argumentos e tipos de retorno dos métodos. Como programador Java, você sabe exatamente com quais condições excepcionais deve lidar e conta com a ajuda do compilador para escrever o software correto que não as deixe sem tratamento.

Tópicos

As aplicações modernas requerem um alto grau de paralelismo. Mesmo um aplicativo muito obstinado pode ter uma interface de usuário complexa, que requer atividades simultâneas. À medida que as máquinas ficam mais rápidas, os usuários ficam menos pacientes com tarefas não relacionadas que controlam seu tempo. Threads fornecem multiprocessamento eficiente e distribuição de tarefas para aplicativos cliente e servidor. Java torna os threads fáceis de usar porque o suporte para eles está embutido na linguagem.

A simultaneidade é boa, mas programar com threads envolve mais do que apenas executar várias tarefas simultaneamente. Na maioria dos casos, os threads precisam ser sincronizados (coordenados), o que pode ser complicado sem suporte explícito de linguagem. Java suporta sincronização baseada no modelo de monitor – uma espécie de sistema de bloqueio e chave para acessar recursos. A palavra-chave sincronizada designa métodos e blocos de código para acesso seguro e serializado dentro de um objeto. Existem também métodos simples e primitivos para espera e sinalização explícita entre threads interessadas no mesmo objeto.

Java possui um pacote de simultaneidade de alto nível que fornece utilitários poderosos que abordam padrões comuns em programação multithread, como pools de threads, coordenação de tarefas e bloqueio sofisticado. Com a adição do pacote de simultaneidade e utilitários relacionados, Java fornece alguns dos utilitários relacionados a threads mais avançados de qualquer linguagem. E quando você precisar de muitos, muitos threads, poderá entrar no mundo dos threads virtuais do Project Loom começando como um recurso de visualização no Java 19.

Embora alguns desenvolvedores talvez nunca precisem escrever código multithread, aprender a programar com threads é uma parte importante do domínio da programação em Java e algo que todos os desenvolvedores devem compreender.

VerCapítulo 9para uma discussão sobre este tema.“Tópicos Virtuais”em particular, apresenta threads virtuais e destaca alguns de seus ganhos de desempenho.

Escalabilidade

Como observamos anteriormente, os programas Java consistem principalmente em classes. Sobre as classes, Java fornece pacotes, uma camada de estrutura que agrupa classes em unidades funcionais. Os pacotes fornecem uma convenção de

nomenclatura para organizar classes e um segundo nível de controle organizacional sobre a visibilidade de variáveis e métodos em aplicativos Java.

Dentro de um pacote, uma classe é visível publicamente ou protegida contra acesso externo. Os pacotes formam outro tipo de escopo mais próximo do nível do aplicativo.

Isso permite a construção de componentes reutilizáveis que funcionam juntos em um sistema. Os pacotes também ajudam a projetar um aplicativo escalável que pode crescer sem se tornar um ninho de pássaros com código fortemente acoplado. Os problemas de reutilização e escala são realmente aplicados apenas com o sistema de módulos adicionado no Java 9.6

Segurança de Implementação

Uma coisa é criar uma linguagem que impeça você de dar um tiro no próprio pé; outra bem diferente é criar um que evite que outros atiram no seu pé.

Encapsulamento é o conceito de ocultar dados e comportamento dentro de uma classe; é uma parte importante do design orientado a objetos. Ajuda você a escrever software limpo e modular. Na maioria das linguagens, entretanto, a visibilidade dos itens de dados é simplesmente parte do relacionamento entre o programador e o compilador.

É uma questão de semântica, não de uma afirmação sobre a segurança real dos dados no contexto do ambiente do programa em execução.

Quando Bjarne Stroustrup, o criador do C++, escolheu a palavra-chave private para designar membros ocultos de classes em C++, ele provavelmente estava pensando em proteger um desenvolvedor dos detalhes confusos do código de outro desenvolvedor, e não em proteger as classes e objetos desse desenvolvedor contra ataques de vírus e cavalos de Tróia de outra pessoa. A conversão arbitrária e a aritmética de ponteiros em C ou C++ tornam trivial violar permissões de acesso em classes sem quebrar as regras da linguagem. Considere o seguinte código:

```
//código C++

classe Finanças {

privado:

char númeroCartãodeCrédito[16];

// ...

};

principal() {

Finanças financeiras;

// Forja um ponteiro para espiar dentro da classe

char *cardno = (char *)&finanças;

printf("Número do cartão = %.16s\n", cardno);
```

```
}
```

Neste pequeno drama C++, escrevemos um código que viola o encapsulamento da classe Finances e extrai algumas informações secretas. Esse tipo de travessura –

abusar de um ponteiro não digitado – não é possível em Java. Se este exemplo parecer irrealista, considere o quão importante é proteger as classes básicas (sistema) do ambiente de tempo de execução contra tipos semelhantes de ataques. Se um código não confiável puder corromper os componentes que fornecem acesso a recursos reais, como sistema de arquivos, rede ou sistema de janelas, ele certamente terá uma chance de roubar os números do seu cartão de crédito.

Java cresceu com a Internet – e com todas as fontes não confiáveis que abundam lá.

Costumava exigir mais segurança do que agora, mas mantém alguns recursos de segurança: um carregador de classes lida com o carregamento de classes do armazenamento local ou da rede e, abaixo disso, toda a segurança do sistema

depende, em última análise, do verificador Java, que garante a integridade de aulas recebidas.

O verificador de bytecode Java é um módulo especial e uma parte fixa do sistema de tempo de execução Java. Os carregadores de classe, entretanto, são componentes que podem ser implementados de maneira diferente por diferentes aplicativos, como servidores ou navegadores da web. Todas essas peças precisam funcionar corretamente para garantir a segurança no ambiente Java.

O Verificador

A primeira linha de defesa do Java é o verificador de bytecode. O verificador lê o bytecode antes de ser executado e garante que ele se comporte bem e obedeça às regras básicas da especificação do bytecode Java. Um compilador Java confiável não produzirá código diferente. No entanto, é possível que uma pessoa travessa monte deliberadamente um bytecode Java incorreto. É função do verificador detectar isso.

Depois que o código for verificado, ele será considerado protegido contra certos erros inadvertidos ou maliciosos. Por exemplo, o código verificado não pode falsificar referências ou violar permissões de acesso a objetos (como em nosso exemplo de cartão de crédito). Ele não pode realizar lançamentos ilegais ou usar objetos de maneira não intencional. Ele nem pode causar certos tipos de erros internos, como estouro ou estouro insuficiente da pilha interna. Essas garantias fundamentais fundamentam toda a segurança do Java.

Você deve estar se perguntando: esse tipo de segurança não está implícito em muitas linguagens interpretadas? Bem, embora seja verdade que você não deveria ser capaz de corromper um interpretador BASIC com uma linha falsa de código BASIC, lembre-se de que a proteção na maioria das linguagens interpretadas acontece em um nível superior. É provável que essas linguagens tenham intérpretes pesados que realizam uma grande quantidade de trabalho em tempo de execução, portanto são

necessariamente mais lentas e complicadas.

Em comparação, o bytecode Java é um conjunto de instruções relativamente leve e de baixo nível. A capacidade de verificar estaticamente o bytecode Java antes da execução permite que o interpretador Java execute em velocidade máxima posteriormente com total segurança, sem verificações dispendiosas em tempo de execução. Esta foi uma das inovações fundamentais em Java.

O verificador é um tipo de “provador de teoremas” matemático. Ele percorre o bytecode Java e aplica regras simples e indutivas para determinar certos aspectos de como o bytecode se comportará. Esse tipo de análise é possível porque o bytecode Java compilado contém muito mais informações de tipo do que o código-objeto de outras linguagens desse tipo. O bytecode também precisa obedecer a algumas regras extras que simplificam seu comportamento. Primeiro, a maioria das instruções de bytecode opera apenas em tipos de dados individuais. Por exemplo, com operações de pilha, existem instruções separadas para referências de objetos e para cada um dos

tipos numéricos em Java. Da mesma forma, há uma instrução diferente para mover cada tipo de valor para dentro e para fora de uma variável local.

Segundo, o tipo de objeto resultante de qualquer operação é sempre conhecido antecipadamente. Nenhuma operação de bytecode consome valores e produz mais de um tipo possível de valor como saída. Como resultado, é sempre possível observar a próxima instrução e seus operandos e saber o tipo de valor que resultará.

Como uma operação sempre produz um tipo conhecido, é possível determinar os tipos de todos os itens na pilha e nas variáveis locais em qualquer ponto no futuro observando o estado inicial. A coleção de todas essas informações de tipo em um determinado momento é chamada de estado de tipo da pilha. Isto é o que Java tenta analisar antes de executar um aplicativo. Java não sabe nada sobre os valores reais da pilha e dos itens variáveis neste momento; ele só sabe que tipo de itens eles são. No entanto, esta informação é suficiente para fazer cumprir as regras de segurança e garantir que os objetos não sejam manipulados ilegalmente.

Para tornar viável a análise do estado de tipo da pilha, Java impõe uma restrição adicional sobre como suas instruções de bytecode são executadas: todos os caminhos para o mesmo ponto no código devem chegar exatamente com o mesmo estado de tipo.

Carregadores de classes

Java adiciona uma segunda camada de segurança com um carregador de classes. Um carregador de classes é responsável por trazer o bytecode das classes Java para o interpretador. Cada aplicativo que carrega classes da rede deve usar um carregador de classes para realizar esta tarefa.

Depois que uma classe é carregada e passada pelo verificador, ela permanece associada ao seu carregador de classes. Como resultado, as classes são efetivamente particionadas em namespaces separados com base em sua origem. Quando uma classe carregada faz referência a outro nome de classe, o local da nova classe é fornecido pelo carregador de classes original. Isso significa que as classes recuperadas de uma fonte

específica podem ser restritas para interagir apenas com outras classes recuperadas do mesmo local. Por exemplo, um navegador da Web habilitado para Java pode usar um carregador de classes para construir um espaço separado para todas as classes carregadas de um determinado URL. Segurança sofisticada baseada em classes assinadas criptograficamente também pode ser implementada usando carregadores de classes.

A busca por classes sempre começa com as classes integradas do sistema Java. Essas classes são carregadas a partir dos locais especificados pelo classpath do interpretador Java (consulteCapítulo 3). As classes no classpath são carregadas pelo sistema apenas uma vez e não podem ser substituídas. Isso significa que é impossível para um aplicativo substituir classes fundamentais do sistema por versões próprias que alterem sua funcionalidade.

Segurança em nível de aplicativo e usuário

Existe uma linha tênue entre ter poder suficiente para fazer algo útil e ter todo o poder para fazer o que quiser. Java fornece a base para um ambiente seguro no qual códigos não confiáveis podem ser colocados em quarentena, gerenciados e executados com segurança. No entanto, a menos que você se contente em manter esse código em uma pequena caixa preta e executá-lo apenas para seu próprio benefício, você terá que conceder-lhe acesso a pelo menos alguns recursos do sistema para que possa ser útil. Todo tipo de acesso traz consigo certos riscos e benefícios. Por exemplo, no ambiente de serviço em nuvem, as vantagens de conceder acesso de código não confiável (desconhecido) ao sistema de arquivos do servidor em nuvem são que ele pode localizar e processar arquivos grandes mais

rapidamente do que você poderia baixá-los e processá-los localmente. Os riscos associados são que o código pode, em vez disso, esgueirar-se pelo servidor em nuvem e possivelmente descobrir informações confidenciais que não deveria ver.

Em um extremo, o simples ato de executar um aplicativo fornece a ele um recurso –

tempo de computação – que pode ser bem utilizado ou queimado levianamente. É

difícil evitar que um aplicativo não confiável desperdice seu tempo ou até mesmo tente um ataque de “negação de serviço”. No outro extremo, um aplicativo poderoso e confiável pode merecer, justificadamente, acesso a todos os tipos de recursos do sistema (como sistema de arquivos, criação de processos ou interfaces de rede); um aplicativo malicioso pode causar estragos nesses recursos. A mensagem aqui é que você deve abordar questões de segurança importantes e às vezes complexas em seus programas.

Em algumas situações, pode ser aceitável simplesmente pedir ao usuário que dê “ok”

às solicitações. A linguagem Java fornece as ferramentas para implementar quaisquer políticas de segurança desejadas. No entanto, as políticas que você escolhe dependem, em última análise, de você confiar ou não na identidade e na integridade do código em questão. É aqui que as assinaturas digitais entram em jogo.

As assinaturas digitais, juntamente com os certificados, são técnicas para verificar se os dados realmente vêm da fonte de onde afirmam ter vindo e não foram modificados no caminho. Se o Banco de Boofa assinar

seu aplicativo de talão de cheques, você poderá verificar se o aplicativo realmente veio do banco, e não de um impostor, e não foi modificado. Portanto, você pode dizer ao seu sistema para confiar no código que possui a assinatura do Banco de Boofa.

Um roteiro Java

Com as constantes atualizações do Java, é difícil acompanhar quais recursos estão disponíveis agora, o que foi prometido e o que já existe há algum tempo. As seções a seguir constituem um roteiro que impõe alguma ordem ao passado, presente e futuro do Java. Quanto às versões do Java, as notas de lançamento da Oracle contêm bons

resumos com links para mais detalhes. Se você estiver usando versões mais antigas para trabalho, considere ler oDocumentos de recursos de tecnologia Oracle.

O Passado: Java 1.0 – Java 20

Java 1.0 forneceu a estrutura básica para o desenvolvimento Java: a própria linguagem, além de pacotes que permitem escrever miniaplicativos e aplicativos simples. Embora 1.0 esteja oficialmente obsoleto, ainda existem alguns miniaplicativos que estão em conformidade com sua API.

Java 1.1 substituiu 1.0, incorporando grandes melhorias no pacote Abstract Window Toolkit (AWT) (recurso GUI original do Java), um novo padrão de evento, novos recursos de linguagem, como reflexão e classes internas, e muitos outros recursos críticos. Java 1.1 é a versão suportada nativamente pela maioria das versões do Netscape e do Microsoft Internet Explorer por muitos anos. Por diversas razões políticas, o mundo dos

navegadores ficou congelado nesta condição por muito tempo.

Java 1.2, apelidado de “Java 2” pela Sun, foi um grande lançamento em dezembro de 1998. Ele forneceu muitas melhorias e adições, principalmente em termos do conjunto de APIs que foram agrupadas nas distribuições padrão. As adições mais notáveis foram a inclusão do pacote Swing GUI como uma API principal e uma nova API de desenho 2D completa. Swing é o kit de ferramentas de UI avançado do Java com recursos que excedem em muito os antigos AWTs. (Swing, AWT e alguns outros pacotes foram chamados de JFC ou Java Foundation Classes.) O Java 1.2 também adicionou uma API de coleções adequada ao Java.

Java 1.3, lançado no início de 2000, adicionou recursos menores, mas se concentrou principalmente no desempenho. Com a versão 1.3, o Java ficou significativamente mais rápido em muitas plataformas e o Swing recebeu muitas correções de bugs.

Nesse período, as APIs corporativas Java, como Servlets e Enterprise JavaBeans, também amadureceram.

Java 1.4, lançado em 2002, integrou um novo conjunto importante de APIs e muitos recursos há muito aguardados. Isso incluiu asserções de linguagem, expressões regulares, preferências e APIs de registro, um novo sistema de E/S para aplicativos de alto volume, suporte padrão para XML, melhorias fundamentais em AWT e Swing e uma API de Servlets Java bastante amadurecida para aplicativos da web.

Java 5, lançado em 2004, foi um lançamento importante que introduziu muitos aprimoramentos de sintaxe de

linguagem há muito esperados, incluindo genéricos, enumerações de tipo seguro, loop for aprimorado, listas de argumentos variáveis, importações estáticas, autoboxing e unboxing de primitivos, também como metadados avançados em classes. Uma nova API de simultaneidade forneceu recursos poderosos de threading, e APIs para impressão e análise formatada semelhantes às de C foram adicionadas. A Invocação de Método Remoto (RMI) também foi reformulada para eliminar a necessidade de stubs e esqueletos compilados. Houve também adições importantes nas APIs XML padrão.

Java 6, lançado no final de 2006, foi uma versão relativamente menor que não adicionou novos recursos sintáticos à linguagem Java, mas incluiu novas APIs de extensão, como aquelas para XML e serviços da web.

Java 7, lançado em 2011, representou uma atualização bastante importante. Vários pequenos ajustes na linguagem, como permitir strings em instruções switch (mais sobre essas duas coisas mais tarde!), juntamente com adições importantes, como a nova biblioteca de E/S java.nio, foram incluídas nos cinco anos após o lançamento do Java 6 .

O Java 8, lançado em 2014, completou alguns dos recursos, como lambdas e métodos padrão, que foram eliminados do Java 7, pois a data de lançamento dessa versão foi adiada continuamente. Esta versão também teve alguns trabalhos feitos no suporte de data e hora, incluindo a capacidade de criar objetos de data imutáveis, úteis para uso nos lambdas agora suportados.

Java 9, lançado após vários atrasos em 2017, introduziu o Module System (Project Jigsaw), bem como um Read-

Evaluate-Print Loop (REPL) para Java: jshell. Usaremos jshell em grande parte de nossas explorações rápidas de vários recursos do Java ao longo do restante deste livro. Java 9 também removeu o JavaDB do JDK.

O Java 10, lançado logo após o Java 9 no início de 2018, atualizou a coleta de lixo e trouxe outros recursos, como certificados raiz, para as compilações do OpenJDK. O

suporte para coleções não modificáveis foi adicionado e o suporte para pacotes antigos de aparência (como o Aqua da Apple) foi removido.

Java 11, lançado no final de 2018, adicionou um cliente HTTP padrão e Transport Layer Security (TLS) 1.3. Os módulos JavaFX e Java EE foram removidos. (JavaFX foi redesenhado para funcionar como uma biblioteca independente.) Os miniaplicativos Java também foram removidos. Junto com o Java 8, o Java 11 faz parte do suporte de longo prazo (LTS) da Oracle. Certas versões — Java 8, Java 11, Java 17 e Java 21 —

serão mantidas por períodos mais longos. A Oracle está tentando mudar a maneira como clientes e desenvolvedores se envolvem com novos lançamentos, mas ainda existem bons motivos para manter as versões conhecidas. Você pode ler mais sobre os pensamentos e planos da Oracle para versões LTS e não-LTS no Oracle Technology Network'sRoteiro de suporte do Oracle Java SE.

Java 12, lançado no início de 2019, adicionou pequenos aprimoramentos de sintaxe de linguagem, como uma visualização de expressões de troca.

O Java 13, lançado em setembro de 2019, inclui mais visualizações de recursos da linguagem, como blocos de

texto, bem como uma grande reimplementação da API Sockets. De acordo com os documentos oficiais de design, este esforço impressionante fornece “uma implementação mais simples e moderna que é fácil de manter e depurar”.

Java 14, lançado em março de 2020, adicionou mais visualizações de aprimoramento de sintaxe de linguagem, como registros, atualizou o recurso de coleta de lixo e removeu as ferramentas e API Pack200. Ele também moveu a expressão switch

visualizada pela primeira vez no Java 12 de seu estado de visualização para a linguagem padrão.

O Java 15, lançado em setembro de 2020, tirou o suporte para blocos de texto (strings multilinhas) da visualização e adicionou classes ocultas e seladas que permitem novas maneiras de restringir o acesso a determinados códigos. (As classes seladas foram mantidas como um recurso de visualização.) O suporte à codificação de texto também foi atualizado para Unicode 13.0.

O Java 16, lançado em março de 2021, manteve as classes lacradas na visualização, mas retirou os registros da visualização. APIs de rede foram expandidas para incluir soquetes de domínio Unix. Ele também adicionou uma opção de saída de lista à API Streams.

Java 17, lançado em setembro de 2021 com LTS, atualizou as classes seladas para um recurso regular da linguagem. Uma prévia da correspondência de padrões para instruções switch foi adicionada junto com diversas melhorias no macOS. Soquetes de datagrama agora podem ser usados para ingressar em grupos multicast.

Java 18, lançado em março de 2022, finalmente tornou o UTF-8 o conjunto de caracteres padrão para APIs Java SE. Ele introduziu um servidor web simples e estático apropriado para prototipagem ou teste e expandiu as opções de resolução de endereços IP.

Java 19, lançado em setembro de 2022, apresentou uma visualização de threads virtuais, simultaneidade estruturada e padrões de registro. O suporte Unicode mudou para a versão 14.0 e alguns formatos adicionais de data e hora foram adicionados.

O Java 20, lançado em março de 2023, finalmente removeu várias operações de threading (parar/pausar/retomar) que foram descontinuadas como inseguras 20

anos antes no JDK 1.2. A análise de strings foi aprimorada para oferecer suporte a grafemas, como símbolos de emoji compostos.

O presente: Java 21

Este livro inclui todas as melhorias mais recentes e maiores até o lançamento do Java 21 em setembro de 2023. Com uma cadência de lançamento de seis meses em vigor, versões mais recentes do JDK quase certamente estarão disponíveis no momento em que você ler isto. Conforme observado acima, a Oracle deseja que os desenvolvedores tratem esses lançamentos como atualizações de recursos. Com exceção dos exemplos que cobrem threads virtuais, Java 17 é suficiente para trabalhar com o código deste livro. Nos raros casos em que utilizarmos um recurso mais recente, anotaremos a versão mínima necessária. Você não precisará “acompanhar” enquanto lê, mas se estiver usando Java

para projetos publicados, considere revisar o roteiro oficial da Oracle para ver se faz sentido permanecer atualizado.

Visão geral dos recursos

Aqui está uma breve visão geral dos recursos mais importantes da API Java principal atual que residem fora da biblioteca padrão:

Conectividade de banco de dados Java (JDBC)

Uma facilidade geral para interagir com bancos de dados (introduzida em Java 1.1).

Invocação de Método Remoto (RMI)

Sistema de objetos distribuídos do Java. RMI permite chamar métodos em objetos hospedados por um servidor executado em algum outro lugar da rede (introduzido em Java 1.1).

Segurança Java

Um recurso para controlar o acesso aos recursos do sistema, combinado com uma interface uniforme para criptografia. Java Security é a base para classes assinadas.

Área de Trabalho Java

Um resumo para um grande número de recursos começando com Java 9, incluindo os componentes Swing UI; “aparência conectável”, que permite adaptar e tematizar toda a interface do usuário; arrastar e soltar; Gráficos 2D; impressão; exibição, reprodução e manipulação de imagens e sons; e recursos de

acessibilidade que podem ser integrados com software e hardware especiais para pessoas com deficiência visual ou outras deficiências.

Internacionalização

A capacidade de escrever programas que se adaptem ao idioma e localidade que o usuário deseja usar. O programa exibe automaticamente o texto na linguagem apropriada (introduzida em Java 1.1).

Interface de nomenclatura e diretório Java (JNDI)

Um serviço geral para procurar recursos. JNDI unifica o acesso a serviços de diretório, como LDAP, NDS da Novell e outros.

A seguir estão APIs de “extensão padrão”. Alguns, como aqueles para trabalhar com XML e serviços web, são fornecidos com a edição padrão do Java; alguns devem ser baixados separadamente e implantados com seu aplicativo ou servidor:

JavaMail

Uma API uniforme para escrever software de email.

Estrutura de mídia Java

Outro recurso para coordenar a exibição de muitos tipos diferentes de mídia que inclui Java 2D, Java 3D, Java Speech (para reconhecimento e síntese de fala), Java Sound (áudio de alta qualidade), Java TV (para televisão interativa e aplicações similares) , e outros.

Servlets Java

Um recurso que permite escrever aplicativos da web do lado do servidor em Java.

Criptografia Java

Implementações reais de algoritmos criptográficos. (Este pacote foi separado do Java Security por motivos legais.)

Linguagem de marcação eXtensible/linguagem de folha de estilo eXtensible

(XML/XSL)

Ferramentas para criar e manipular documentos XML, validá-los, mapeá-los de e para objetos Java e transformá-los com folhas de estilo.

Tentaremos abordar alguns desses recursos. Infelizmente para nós (mas felizmente para os desenvolvedores de software Java), o ambiente Java tornou-se tão rico que é impossível cobrir tudo em um único livro. Notaremos outros livros e recursos que cobrem quaisquer tópicos que não possamos abordar em profundidade.

O futuro

Java certamente não é a novidade atualmente, mas continua a ser uma das plataformas mais populares para desenvolvimento web e de aplicativos. Isto é especialmente verdadeiro nas áreas de serviços web, estruturas de aplicativos web e ferramentas XML. Embora o Java não tenha dominado as plataformas móveis da maneira que parecia destinado, você pode usar a linguagem Java e as principais APIs para programar para o sistema operacional móvel Android do Google, que é usado em bilhões de dispositivos em todo o mundo. No campo da Microsoft, a linguagem C#

derivada de Java assumiu grande parte do desenvolvimento do .NET e trouxe a sintaxe e os padrões principais do Java para essas plataformas.

A própria JVM também é uma área interessante de exploração e crescimento. Novas linguagens estão surgindo para aproveitar o conjunto de recursos e a onipresença da JVM.Clojureé uma linguagem funcional robusta com uma base de fãs crescente surgindo em uma variedade de trabalhos, desde amadores até as maiores lojas.

EKotliné uma linguagem de uso geral que assume o desenvolvimento do Android com entusiasmo. Ele está ganhando força em novos ambientes, ao mesmo tempo que mantém boa interoperabilidade com Java.

Provavelmente, as áreas de mudança mais interessantes em Java hoje são encontradas nas tendências em direção a estruturas mais leves e simples para negócios e à integração da plataforma Java com linguagens dinâmicas para scripts de páginas e extensões da Web. Há muito mais trabalhos interessantes por vir.

Você tem diversas opções para ambientes de desenvolvimento Java e sistemas de tempo de execução. O Java Development Kit da Oracle está disponível para macOS, Windows e Linux. VisitaSite Java da Oraclepara obter mais informações sobre como obter o JDK oficial mais recente.

Desde 2017, a Oracle oferece suporte oficial a atualizações do OpenJDK de código aberto. Pessoas físicas e pequenas (ou mesmo médias) empresas podem achar esta versão gratuita suficiente. Os lançamentos ficam atrás do lançamento comercial do JDK e não

incluem o suporte técnico da Oracle, mas a Oracle declarou um firme

compromisso em manter o acesso livre e aberto ao Java. Todos os exemplos neste livro foram escritos e testados usando o OpenJDK. Você pode obter mais detalhes diretamente da boca do cavalo (Oráculo?)Site OpenJDK.

Para instalação rápida de uma versão gratuita do Java 19 (suficiente para quase todos os exemplos deste livro, embora observemos alguns recursos de linguagem de versões posteriores), a Amazon oferece seuCorretodistribuição on-line com instaladores amigáveis e familiares para todas as três plataformas principais.Capítulo 2orientará você na instalação básica do Corretto no Windows, macOS e Linux.

Há também uma variedade de ambientes de desenvolvimento integrados (IDEs) Java populares. Discutiremos um neste livro: a Community Edition gratuita do JetBrains' sIDEIA do IntelliJ. Esse ambiente de desenvolvimento completo permite escrever, testar e empacotar software com ferramentas avançadas ao seu alcance.

Exercícios

No final de cada capítulo, forneceremos algumas perguntas e exercícios de código para você revisar. As respostas às perguntas podem ser encontradas emApêndice B. As soluções para exercícios de código estão incluídas nos outros exemplos de código

emGitHub. (Apêndice Afornece detalhes sobre como baixar e usar o código deste livro.) Recomendamos que você responda às perguntas e experimente os exercícios.

Não se preocupe se precisar voltar a um capítulo e ler um pouco mais para encontrar uma resposta ou procurar algum nome de método. Essa é a questão! Aprender como usar este livro como referência será útil no futuro.

1. Qual empresa mantém Java atualmente?

2. Qual é o nome do kit de desenvolvimento de código aberto para Java?

3. Cite os dois componentes principais que desempenham um papel na

abordagem Java para executar bytecode com segurança.

1O apelido Standard Edition (SE) apareceu no início da história do Java, quando a Sun lançou a plataforma J2EE, ou Java 2 Enterprise Edition. A Enterprise Edition agora atende pelo nome de “Jakarta EE”.

4Asserções estão além do escopo deste livro, mas são um tópico digno de ser explorado depois que você tiver adquirido maior domínio em Java. Você encontrará alguns detalhes básicos noDocumentação do Oracle Java SE.

5O crédito pela analogia do carro vai para Marshall P. Cline, autor de C++ FAQ.

Capítulo 2. Uma primeira aplicação

Antes de mergulhar em nossa discussão completa sobre a linguagem Java, vamos começar a trabalhar com algum código funcional e brincar um pouco. Neste capítulo, construiremos um pequeno aplicativo amigável que ilustra muitos dos conceitos usados ao longo do livro. Aproveitaremos esta oportunidade para apresentar recursos gerais da linguagem e dos aplicativos Java.

Este capítulo também serve como uma breve introdução aos aspectos orientados a objetos e multithread do Java. Se esses conceitos são novos para você, esperamos que encontrá-los aqui em Java pela primeira vez seja uma experiência simples e agradável.

Se você trabalhou com outro ambiente de programação orientado a objetos ou multithread, deverá apreciar especialmente a simplicidade e elegância do Java. Este capítulo destina-se apenas a fornecer uma visão geral da linguagem Java e uma ideia de como ela é usada. Se você tiver problemas com algum dos conceitos apresentados aqui, tenha certeza de que eles serão abordados com mais detalhes posteriormente neste livro.

Não podemos enfatizar o suficiente a importância da experimentação à medida que você aprende novos conceitos aqui e ao longo do livro. Não apenas leia os exemplos –

execute-os. Onde pudermos, mostraremos como usar o jshell (mais sobre isso

em“Experimentando Java”) para experimentar coisas em tempo real. O código-fonte dos exemplos deste livro pode ser encontrado emGitHub. Compile os programas e experimente-os. Em seguida, transforme nossos exemplos em seus exemplos: brinque com eles, mude seu comportamento, quebre-os, conserte-os e, com sorte, divirta-se ao longo do caminho.

Ferramentas e ambiente Java

Embora seja possível escrever, compilar e executar aplicativos Java com nada mais do que o Java Development Kit (OpenJDK) de código aberto da Oracle e um editor de texto simples (como vi ou Notepad), hoje a

grande maioria do código Java é escrita com o benefício de um Ambiente de Desenvolvimento Integrado (IDE). Os benefícios de usar um IDE incluem uma visão completa do código-fonte Java com destaque de sintaxe, ajuda de navegação, controle de origem, documentação integrada, construção, refatoração e implantação, tudo ao seu alcance. Portanto, vamos pular um tratamento acadêmico de linha de comando e começar com um IDE popular e gratuito – IntelliJ

IDEA CE (Community Edition). Se você não quiser usar um IDE, sinta-se à vontade para usar os comandos de linha de comando javac HelloJava.java para compilação e java HelloJava para executar os próximos exemplos.

O IntelliJ IDEA requer a instalação do Java. Este livro aborda os recursos da linguagem Java 21, portanto, embora os exemplos neste capítulo funcionem com versões mais antigas, é melhor ter o JDK 21 instalado para garantir que todos os exemplos do livro sejam compilados. (Java 19 também tem todos os recursos mais importantes disponíveis, embora muitos estejam tecnicamente no modo de “visualização”.) O JDK

inclui várias ferramentas de desenvolvedor que discutiremos emCapítulo 3. Você pode verificar qual versão, se houver, você instalou digitando java -version na linha de comando. Se o Java não estiver presente ou se for uma versão anterior ao JDK 19, você desejará instalar uma versão mais recente, conforme discutido em“Instalando o JDK”.

Tudo o que você precisa para os exemplos deste livro é o JDK básico.

Instalando o JDK

Deve ser dito desde o início que você é livre para baixar e usar o software oficial e comercialJDK da Oraclepara uso pessoal. As versões disponíveis na página de download da Oracle incluem a versão mais recente e a versão de suporte de longo prazo mais recente (ambas são a versão 21 no momento da redação deste artigo), com links para versões mais antigas se a compatibilidade legada for algo que você deve gerenciar. Tanto o Java 8 quanto o Java 11, por exemplo, continuam sendo burros de carga nos back-ends de grandes organizações.

No entanto, se você planeja usar Java em qualquer capacidade comercial ou compartilhada, o Oracle JDK agora vem com termos de licenciamento rigorosos (e pagos). Por esta e outras razões mais filosóficas, usamos principalmente o OpenJDK

mencionado anteriormente em"Crescendo". Lamentavelmente, esta versão de código aberto não inclui instaladores para todas as diferentes plataformas. Ser de código aberto, no entanto, significa que outros grupos são bem-vindos para intervir e fornecer as peças que faltam, e existem vários pacotes baseados no instalador OpenJDK. A Amazon lançou consistentemente instaladores oportunos sob

oCorretoapelido. Passaremos pelas etapas básicas de instalação do Corretto no Windows, Mac e Linux neste capítulo.

Para quem deseja a versão mais recente e não se importa com um pouco de trabalho de configuração, dê uma olhada na instalação do OpenJDK. Embora não seja tão simples quanto usar instaladores nativos típicos, instalar o OpenJDK no sistema operacional escolhido geralmente é apenas uma questão de descompactar o

arquivo baixado em uma pasta e certificar-se de que algumas variáveis de ambiente (JAVA_HOME e PATH) estejam configuradas corretamente. Independentemente do sistema operacional que você usa, se for usar o OpenJDK, você irá paraPágina de

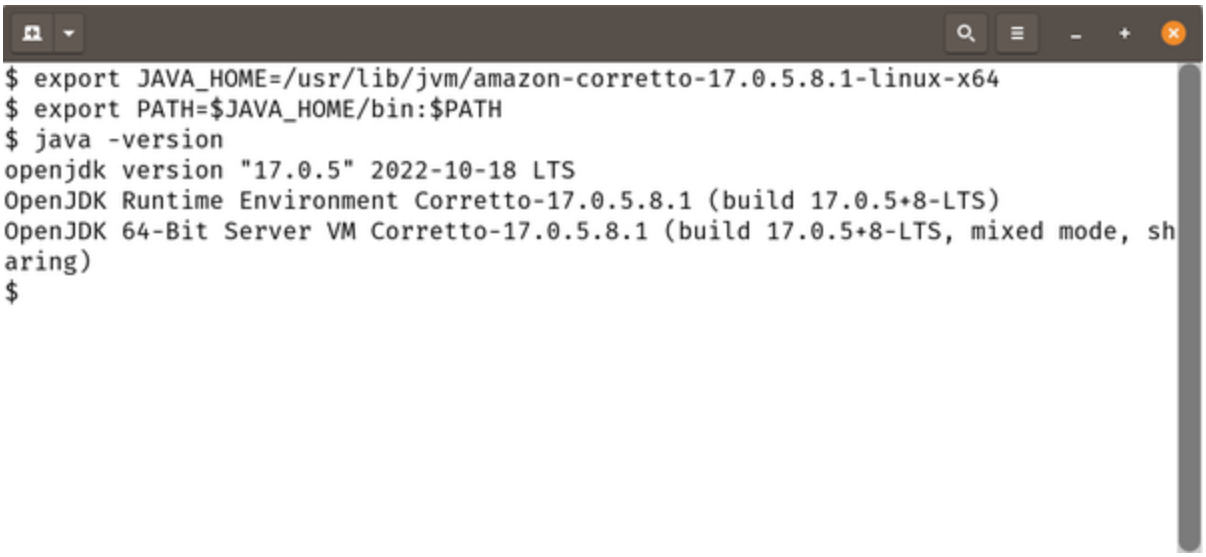
download do OpenJDK da Oracle. Lá eles listam os lançamentos atuais, bem como quaisquer versões de acesso antecipado disponíveis.

Instalando Corretto no Linux

Para as distribuições populares Debain e Red Hat, você pode baixar o arquivo apropriado (.deb ou .rpm, respectivamente) e usar seu gerenciador de pacotes usual para instalar o JDK. O arquivo para sistemas Linux genéricos é um arquivo tar compactado (tar.gz) que pode ser descompactado em qualquer diretório

compartilhado de sua escolha. Seguiremos as etapas para descompactar e configurar esse arquivo tar compactado, pois ele funciona na maioria das distribuições Linux.

Essas etapas usam a versão 17 do Java, mas se aplicam a todos os downloads atuais e do LTS Corretto.



Decida onde você deseja instalar o JDK. Armazenaremos o nosso em /usr/lib/jvm, mas outras distros podem usar outros locais, como /opt, /usr/share ou /usr/local. Se você for o único a usar Java em seu sistema, poderá até mesmo descompactar o arquivo em seu diretório inicial. Usando seu aplicativo de terminal favorito, mude para o diretório onde você baixou o arquivo e execute os seguintes comandos para instalar o Java:

~$ downloads de cd

~/Downloads$ sudo tar xzf amazon-corretto-17.0.5.8.1-linux-x64.tar.gz \

--diretório /usr/lib/jvm

~/Downloads$ /usr/lib/jvm/amazon-corretto-17.0.5.8.1-linux-x64/bin/java -

version

versão openjdk "17.0.5" 2022-10-18 LTS

Ambiente de tempo de execução OpenJDK Corretto-17.0.5.8.1 (compilação 17.

0.5+8-LTS)

VM do servidor OpenJDK de 64 bits Corretto-17.0.5.8.1 (compilação 17.0.5+

8-LTS, modo misto,

compartilhamento)

Você pode ver que a primeira linha das informações da versão termina com as iniciais LTS. Esta é uma maneira fácil de determinar se você está usando uma versão de suporte de longo prazo. Com o Java descompactado com sucesso, você pode configurar seu terminal para usar esta versão definindo as variáveis de ambiente JAVA_HOME e PATH:

$ exportar JAVA_HOME=/usr/lib/jvm/amazon-corretto-17.0.5.8.1-linux-x64

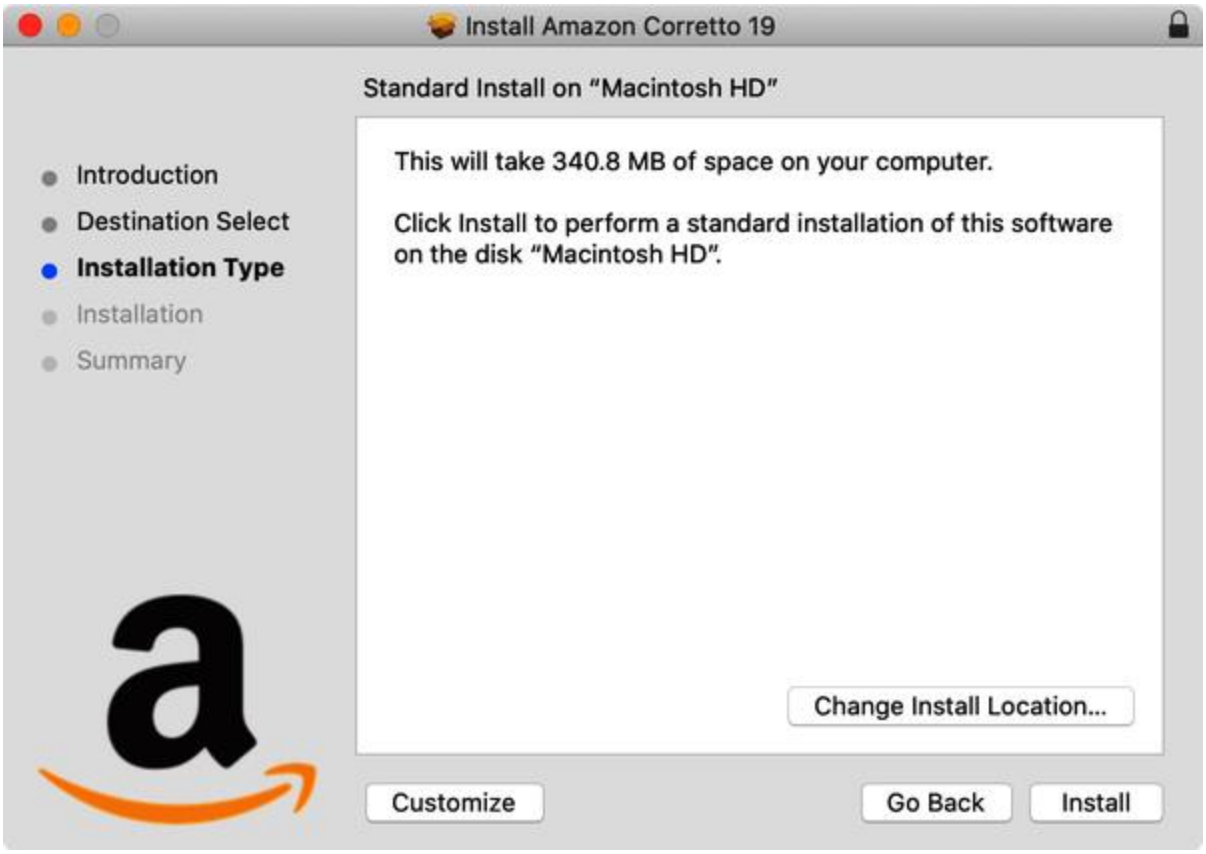
$ exportar PATH=$JAVA_HOME/bin:$PATH

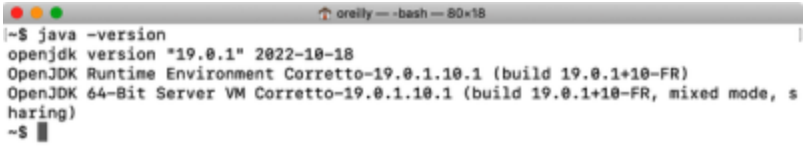
Você pode testar se esta configuração está funcionando verificando a versão do Java usando o sinalizador -version, conforme mostrado emFigura 2-1.

Você desejará tornar essas alterações JAVA_HOME e PATH permanentes, atualizando os scripts de inicialização ou rc do seu shell. Por exemplo, se você usar bash como shell, poderá adicionar ambas as linhas de exportação emFigura 2-1para o seu arquivo

.bashrc.

Figura 2-1. Verificando sua versão Java no Linux





Instalando o Corretto no macOS

Para usuários de sistemas macOS, oDownload corretoe o processo de instalação é simples. Selecione a versão do JDK que deseja usar e selecione o link .pkg na página de download subsequente. Clique duas vezes no arquivo baixado para iniciar o assistente.

O assistente de instalação mostrado emFigura 2-2não permite muita personalização real. O JDK será instalado no disco que executa o macOS em sua própria pasta no

diretório /Library/Java/JavaVirtualMachines. Ele estará simbolicamente vinculado a

/usr/bin/java. Embora você possa selecionar um local de instalação alternativo se tiver discos rígidos separados com o macOS, os padrões funcionam bem para os propósitos deste livro.

Figura 2-2. Assistente de instalação Corretto no macOS

Depois de concluir a instalação, você pode testar o Java abrindo o aplicativo Terminal normalmente encontrado na pasta Utilitários na pasta global Aplicativos. Digite java -

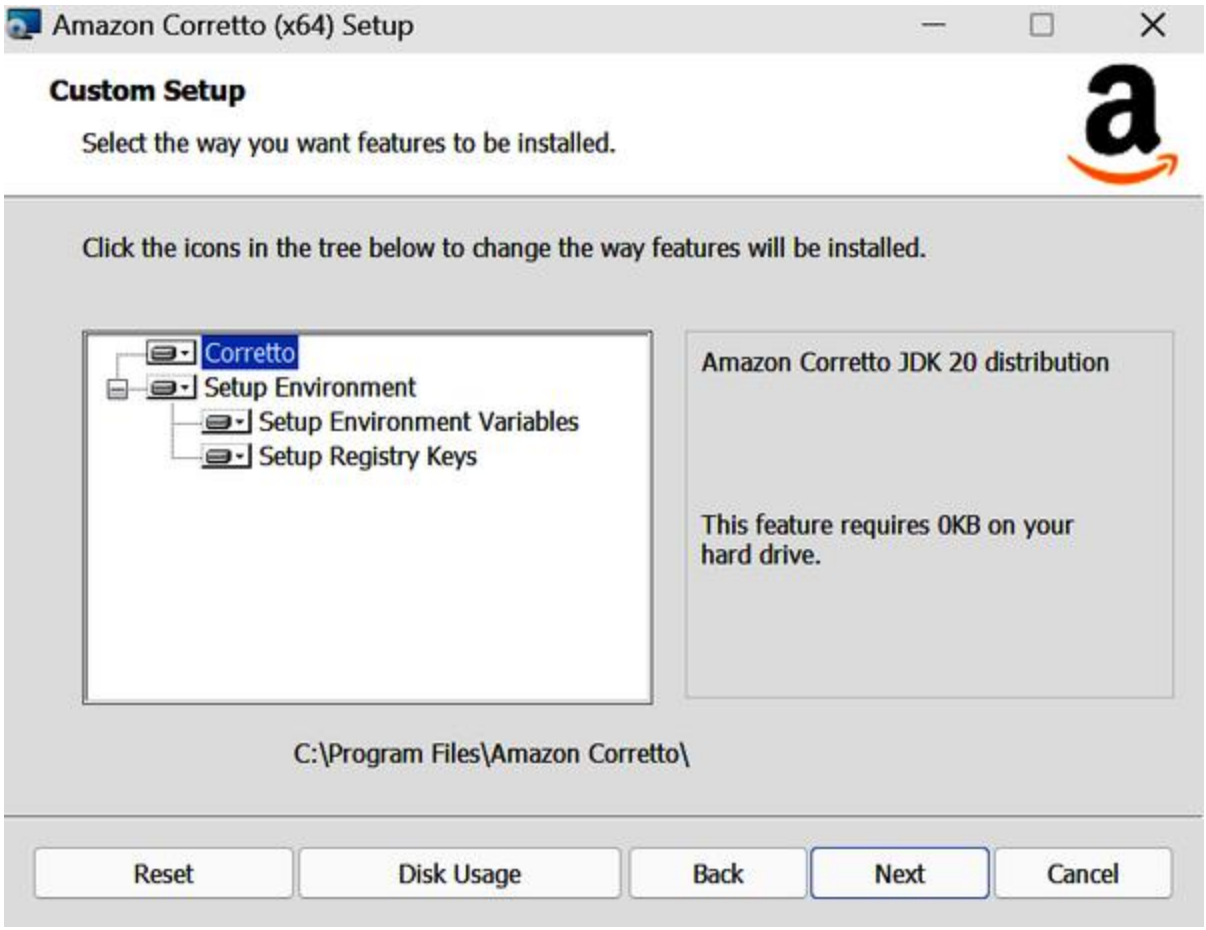
version e você verá uma saída semelhante aFigura 2-3. Instalamos a versão 19 neste sistema, mas você deverá ver a versão baixada refletida na saída.

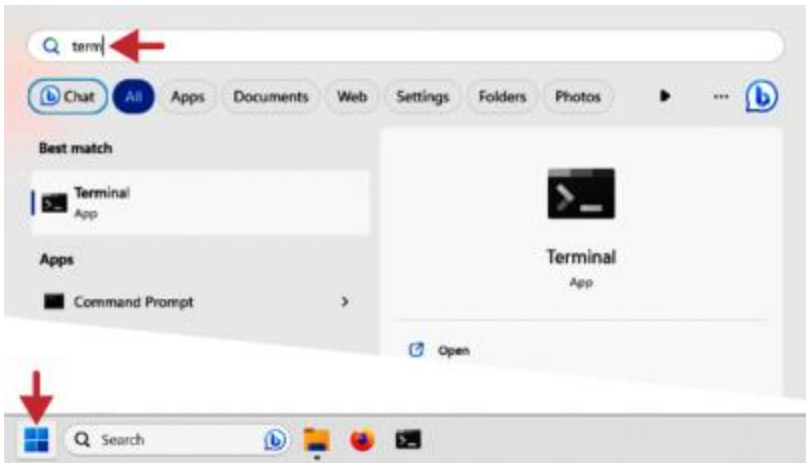
Figura 2-3. Verificando sua versão Java no macOS

Instalando o Corretto no Windows

O instalador do Corretto para Windows (baixe o arquivo .msi emSite da Amazon)

segue assistentes de instalação típicos do Windows, como visto emFigura 2-4. Você pode aceitar os padrões conforme segue os prompts curtos ou ajustar as coisas se estiver familiarizado com tarefas administrativas, como configurar variáveis de ambiente e entradas de registro. Se você for solicitado a permitir que o instalador faça alterações em seu sistema, vá em frente e diga sim.


Amazon Corretto (x64) Setup
Custom Setup
Select the way you want features to be installed.
Click the icons in the tree below to change the way features will be installed.
Corretto
Setup Environment
Setup Environment Variables
Setup Registry Keys
Amazon Corretto JDK 20 distribution
This feature requires 0KB on your hard drive.
C:\Program Files\Amazon Corretto\
Reset
Disk Usage
Back
Next
Cancel


term
Chat
All
Apps
Documents
Web
Settings
Folders
Photos
Best match
Terminal
App
Apps
Command Prompt
Terminal
App
Open
Search

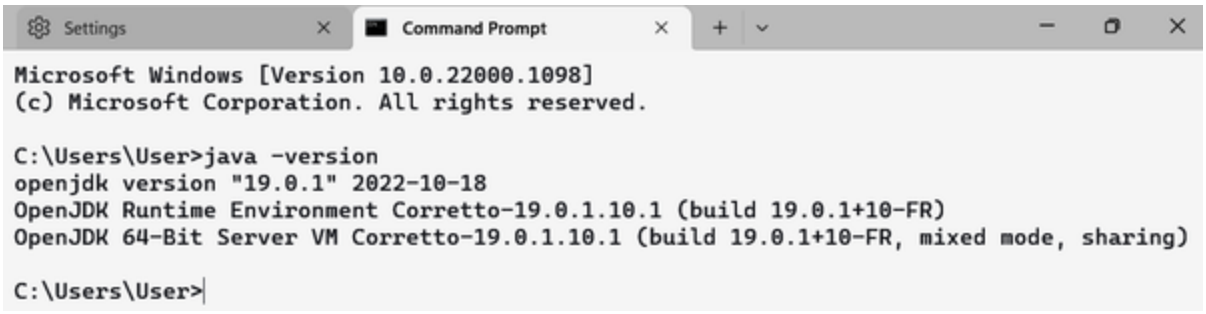


Figura 2-4. Assistente de instalação Corretto no Windows

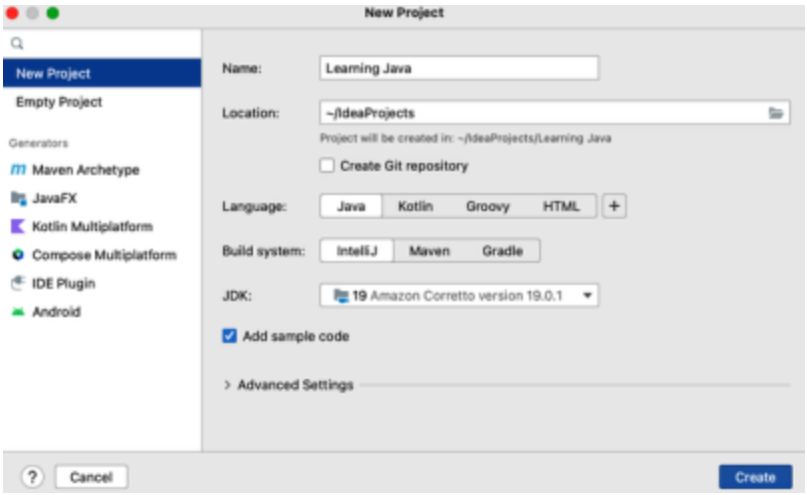
Talvez você não use uma linha de comando regularmente no Windows, mas o aplicativo Terminal em novas versões do Windows (ou o aplicativo Prompt de Comando em versões mais antigas) tem a mesma finalidade que aplicativos semelhantes no macOS ou Linux. No menu do Windows, você pode pesquisar por termo ou cmd, conforme mostrado emFigura 2-5.

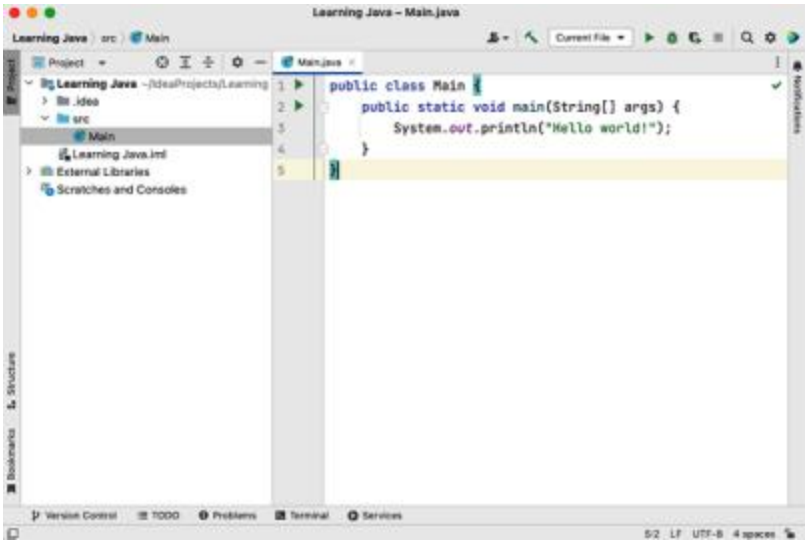
Figura 2-5. Localizando um aplicativo de terminal no Windows

Clique no resultado apropriado para iniciar seu terminal e verifique a versão do Java digitando java -version. Em nosso exemplo, estávamos executando a versão 19; você deve ver algo semelhante aFigura 2-6mas com o número da sua versão.

Figura 2-6. Verificando a versão Java no Windows

Você pode continuar usando o terminal, é claro, mas agora também está livre para apontar outros aplicativos, como o IntelliJ IDEA, para o JDK instalado e simplesmente trabalhar com essas ferramentas. E por falar em IntelliJ IDEA, vamos examinar suas etapas de instalação com mais detalhes.





Instalando o IntelliJ IDEA e criando um projeto

IntelliJ IDEA é um IDE disponível noSite da JetBrains. Para os propósitos deste livro, e para começar a usar Java em geral, a Community Edition é suficiente. O download é um instalador executável ou arquivo compactado: .exe para Windows, .dmg para macOS e .tar.gz para Linux. Todos os instaladores (e arquivos) seguem procedimentos padrão e devem parecer familiares. Se você quiser um pouco mais de orientação,

oGuia de instalaçãono site JetBrains é um ótimo recurso.

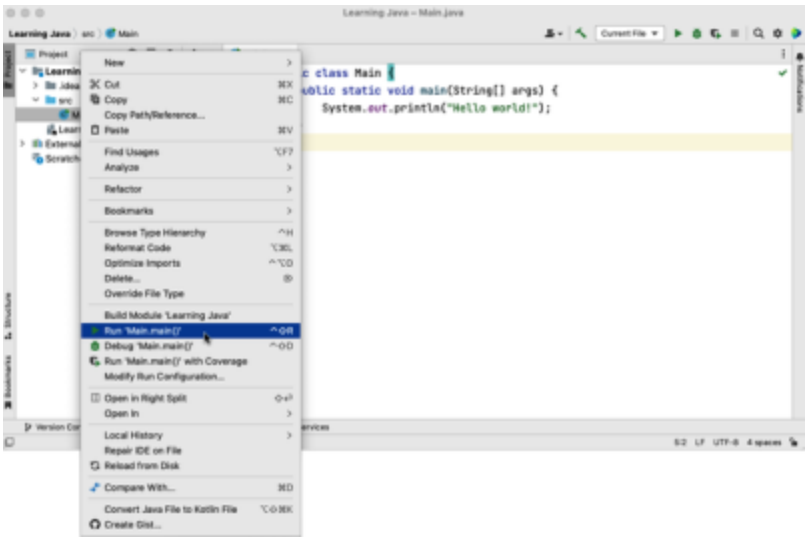
Vamos criar um novo projeto. Selecione Arquivo → Novo → Projeto no menu do aplicativo e digite Learning Java no campo “Nome” na parte superior da caixa de diálogo, conforme mostrado emFigura 2-7. Selecione um JDK (versão 19 ou posterior será suficiente) e certifique-se de que a caixa de seleção “Adicionar código de amostra” esteja marcada.

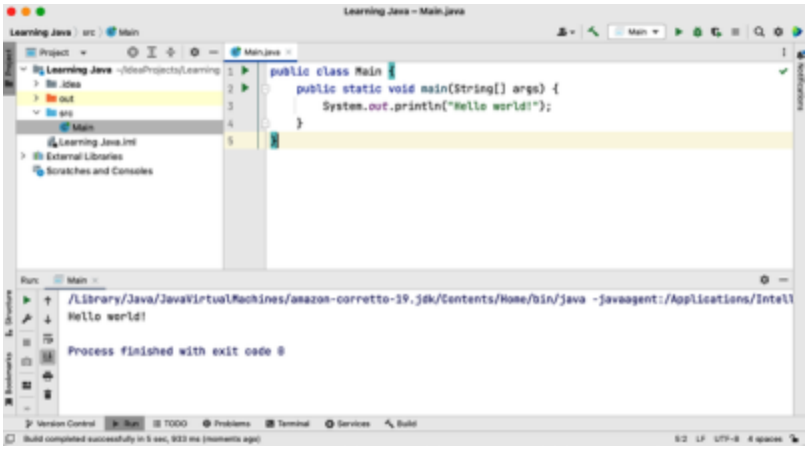
Figura 2-7. Caixa de diálogo Novo projeto Java

Você pode notar a lista de geradores no lado esquerdo da caixa de diálogo. O “Novo Projeto” padrão é perfeito para nossas necessidades. Mas você pode iniciar outros projetos com modelos para coisas como Kotlin ou Android. O padrão inclui uma classe Java mínima com um método main() que pode ser executado. Os próximos capítulos entrarão em muito mais detalhes sobre a estrutura dos programas Java e os comandos e instruções que você pode colocar nesses programas. Com a opção padrão selecionada à esquerda, vá em frente e clique no botão Criar. (Se você notar uma solicitação para baixar índices compartilhados, vá em frente e diga sim. Os índices compartilhados não são críticos, mas farão com que o IDEA seja executado um pouco mais rápido.) Você deve terminar com um projeto simples que inclui um arquivo Main.java , como mostrado emFigura 2-8.

Figura 2-8. A classe principal em IDEA

Parabéns! Agora você tem um programa Java. Você executará este exemplo e depois o expandirá para dar um pouco mais de estilo. Os próximos capítulos apresentarão





exemplos mais interessantes reunindo cada vez mais elementos de Java. Porém, sempre construiremos esses exemplos em uma configuração semelhante. Essas etapas iniciais são boas para você controlar.

Executando o Projeto

Começar com o modelo simples fornecido pelo IDEA deve deixá-lo em boa forma para executar seu primeiro programa. Olhe p[ara trásFigura 2-8. O](#)bserve os triângulos verdes nas linhas 1 e 2 ao longo do lado esquerdo do editor de código, próximo à classe Main e

também ao método main(). IDEA entende que Main pode ser executado.

Você pode clicar em qualquer um desses botões para executar seu código. (A classe Main listada na pasta src no esboço do projeto à esquerda também tem um pequeno botão verde “play”.) Você pode clicar com o botão direito na entrada da classe e selecionar a opção Executar 'Main.main()', como mostrado emFigura 2-9.

Figura 2-9. Executando seu projeto Java

Quer você use os botões de margem do editor ou o menu de contexto, vá em frente e execute seu código agora. Você deverá ver o seu “Hello World!” mensagem aparece na guia Executar na parte inferior do editor, semelhante aFigura 2-10.

Figura 2-10. Nossa primeira saída de um programa Java

Os IDEs também incluem uma opção de terminal útil. Isso permite que você abra uma guia ou janela que tenha um prompt de comando disponível. Talvez você não precise dessa opção com muita frequência, mas ela definitivamente pode ser útil. No IDEA, por exemplo, você pode abrir a guia do terminal na opção de menu Exibir → Janelas de ferramentas → Terminal ou clicando no atalho Terminal na parte inferior da janela principal, conforme mostrado emFigura 2-11.

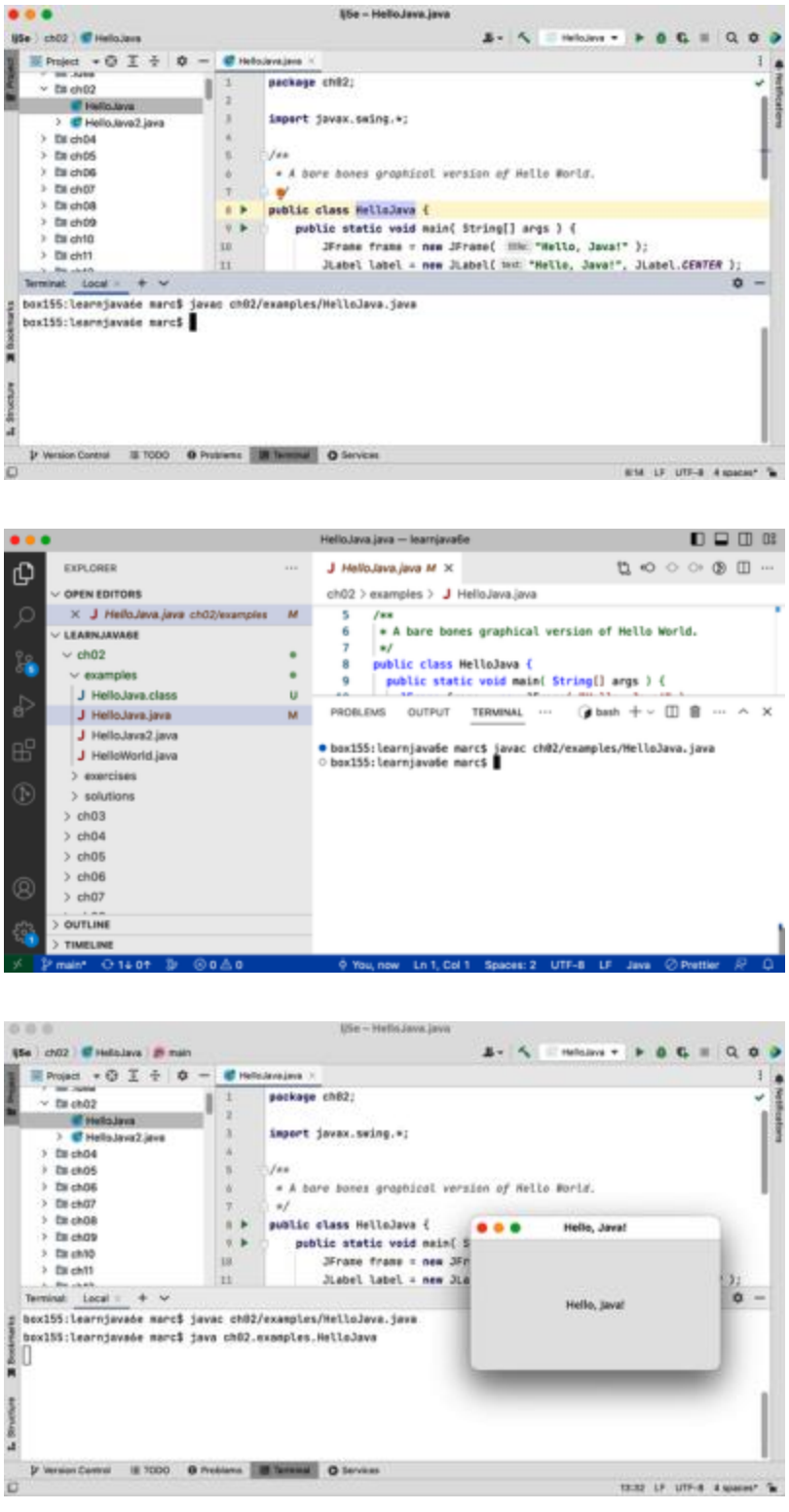


Figura 2-11. A guia Terminal no IntelliJ IDEA

No VS Code, você pode usar a opção de menu Terminal → Novo Terminal para abrir uma parte semelhante do IDE, conforme mostrado emFigura 2-12.

Figura 2-12. A guia Terminal no VS Code da Microsoft

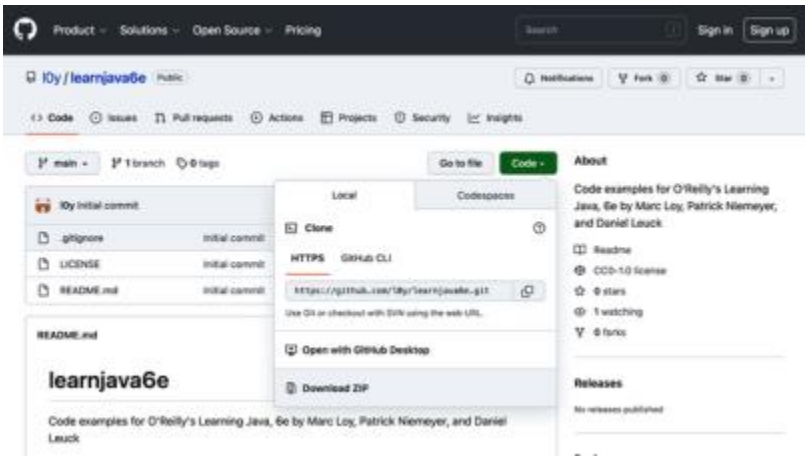
Sinta-se à vontade para experimentar o terminal você mesmo. Com uma janela de terminal aberta em seu IDE, navegue até a pasta Learning Java. (A maioria dos IDEs abrirá automaticamente o terminal no diretório base do seu projeto.) Use o comando java para executar nosso programa principal, conforme mostrado emFigura 2-13.

Figura 2-13. Executando um programa Java em uma guia de terminal

Seja qual for o caminho que você escolher, parabéns novamente – você executou seu primeiro programa Java!

Pegando os exemplos

Os exemplos de código e soluções de exercícios estão disponíveis online no site do livroRepositório GitHub. O GitHub se tornou o site de repositório em nuvem de fato para projetos de código aberto disponíveis ao público, bem como para projetos privados de código fechado. O GitHub tem muitas ferramentas úteis além do simples armazenamento e controle de versão de código-fonte. Se você desenvolver um aplicativo ou biblioteca que deseja compartilhar com outras pessoas, vale a pena criar uma conta no GitHub e explorá-la mais profundamente. Felizmente, você também



pode simplesmente obter arquivos ZIP de projetos públicos sem fazer login, conforme mostrado em Figura 2-14.

Figura 2-14. Baixando um arquivo ZIP do GitHub

Você deve obter um arquivo chamado learnjava6e-main.zip (já que você está pegando um arquivo do branch “principal” deste repositório). Se você estiver familiarizado com o GitHub de outros projetos, sinta-se à vontade para clonar o repositório, mas o arquivo ZIP estático contém tudo que você precisa para experimentar os exemplos enquanto lê o restante deste livro. Ao descompactar o download, você encontrará pastas para todos os capítulos que contêm exemplos, bem como uma pasta de jogo completa que contém um jogo divertido e alegre de lançar maçãs para ajudar a ilustrar muitos dos conceitos de programação apresentados ao longo do livro de uma forma coesa. aplicativo. Entraremos em mais detalhes sobre os exemplos e o jogo nos próximos capítulos.

Conforme mencionado anteriormente, você pode compilar e executar os exemplos do arquivo ZIP diretamente na linha de comando. Você também pode importar o código para seu IDE favorito. Apêndice

Acontém informações detalhadas sobre como importar melhor esses exemplos para o IntelliJ IDEA, mas outros IDEs populares, como o VS Code da Microsoft, também funcionarão.

OláJava

Na tradição dos textos introdutórios à programação, começaremos com o equivalente Java do arquétipo da aplicação “Hello World”, HelloJava.

Acabaremos dando algumas passadas neste exemplo antes de terminarmos

(HelloJava, HelloJava2, etc.), adicionando recursos e introduzindo novos conceitos ao longo do caminho. Mas vamos começar com a versão minimalista. Crie um novo arquivo chamado HelloJava.java em seu espaço de trabalho. (Se estiver usando IDEA, você pode fazer isso nos menus: Arquivo → Novo → Classe Java. Em seguida, dê a ele o nome HelloJava sem sufixo – a extensão .java será adicionada ao nome do arquivo automaticamente.) Vá em frente e preencha o mesmo método main() da

demonstração principal fornecida ao criar o novo projeto:

```
//ch02/examples/HelloJava.java

classe pública HelloJava {

public static void main(String[] args) {

System.out.println("Olá, Java!");

}

}
```

Este programa de cinco linhas declara uma classe chamada HelloJava e aquele importante método main(). Ele usa um método predefinido chamado println() para escrever algum texto como saída. Este é um programa de linha de comando, o que significa que ele é executado em um terminal ou janela DOS e imprime sua saída lá.

Essa abordagem é um pouco antiquada, então, antes de prosseguirmos, daremos ao HelloJava uma interface gráfica de usuário (GUI). Não se preocupe com o código ainda; basta acompanhar a progressão aqui e voltaremos para explicações em um momento.

No lugar da linha que contém o método println(), usaremos um objeto JFrame para colocar uma janela na tela. Podemos começar substituindo a linha println pelas três linhas a seguir:

//nome do arquivo: ch02/examples/HelloJava.java

//método: principal()

Quadro JFrame = new JFrame("Olá, Java!");

frame.setSize(300, 150);

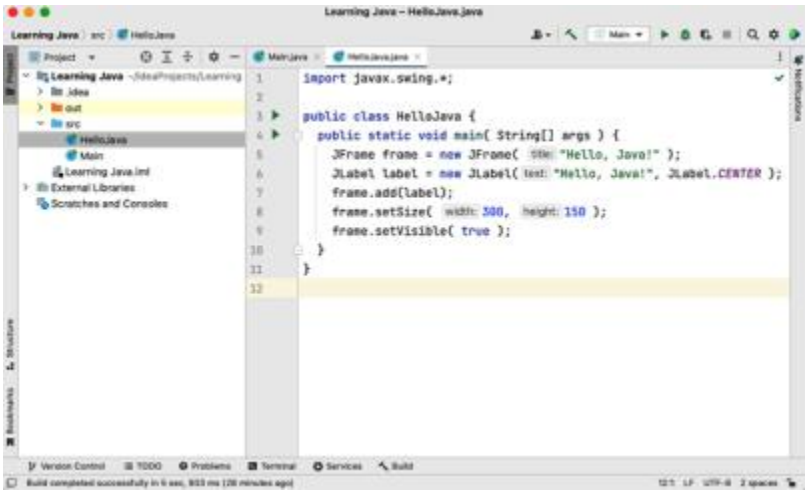
frame.setVisible (verdadeiro);

Este snippet cria um objeto JFrame com o título “Hello, Java!” JFrame representa uma janela gráfica. Para exibi-lo, basta configurar seu tamanho na tela usando o método setSize() e torná-lo visível chamando o método setVisible().

Se parássemos aqui, veríamos uma janela vazia na tela com nosso “Hello, Java!”

banner como seu título. Mas gostaríamos que a nossa mensagem ficasse dentro da janela e não apenas no topo. Para colocar algo na janela, precisamos de mais algumas linhas. O exemplo completo a seguir adiciona um objeto JLabel para exibir o texto centralizado em nossa janela. A linha de importação adicional na parte superior é necessária para informar ao compilador Java onde encontrar as definições dos objetos JFrame e JLabel que estamos usando:

```
//ch02/examples/HelloJava.java

pacote ch02.examples;

importar javax.swing.*;

classe pública HelloJava {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Olá, Java!");

frame.setSize(300, 150);

Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

frame.add(rótulo);

frame.setVisible (verdadeiro);

}

}
```



Agora, para compilar e executar esta fonte, clique com o botão direito em sua classe HelloJava.java no explorador de pacotes à esquerda e use o menu de contexto ou clique em uma das setas verdes na margem esquerda do editor. VerFigura 2-15.

Figura 2-15. Executando o aplicativo HelloJava

Você deverá ver a proclamação mostrada emFigura 2-16. Parabéns novamente, você executou seu segundo aplicativo Java! Reserve um momento para aproveitar o brilho do seu monitor.

Figura 2-16. A saída do aplicativo HelloJava

Esteja ciente de que quando você clica na caixa de fechamento da janela, a janela desaparece, mas o programa ainda está em execução. (Corrigiremos esse comportamento de desligamento em breve.) Para interromper o aplicativo Java no IDEA, clique no botão quadrado vermelho “parar” à direita do botão verde de reprodução que usamos para executar o programa. Se você estiver executando o exemplo na linha de comando, digite Ctrl-C.

OláJavapode ser um programa pequeno, mas há muita coisa acontecendo nos bastidores. Essas poucas linhas representam a ponta de um iceberg impressionante. O

que está por baixo da superfície são as camadas de funcionalidade fornecidas pela linguagem Java e suas bibliotecas Swing. Lembre-se de que neste capítulo abordaremos muito assunto rapidamente, em um esforço para lhe mostrar o panorama geral. Tentaremos oferecer detalhes suficientes para uma boa compreensão do que está acontecendo em cada exemplo, mas adiaremos explicações detalhadas até os capítulos apropriados. Isso vale tanto para os elementos da linguagem Java quanto para os conceitos orientados a objetos que se aplicam a eles. Dito isso, vamos dar uma olhada agora no que está acontecendo em nosso primeiro exemplo.

Aulas

O primeiro exemplo define uma classe chamada HelloJava:

```
classe pública HelloJava {

// ...
```

}

As classes são os blocos de construção fundamentais da maioria das linguagens orientadas a objetos. Uma classe é um grupo de itens de dados com funções associadas que podem realizar operações nesses dados. Os itens de dados em uma classe são chamados de variáveis ou, às vezes, de campos; em Java, as funções são chamadas de métodos. Os principais benefícios de uma linguagem orientada a objetos são esta associação entre dados e funcionalidade em unidades de classe e a capacidade das classes de encapsular ou ocultar detalhes, liberando o desenvolvedor de se preocupar com detalhes de baixo nível. Expandiremos esses benefícios emcapítulo 5onde preenchemos a estrutura das aulas.

Em um aplicativo, uma classe pode representar algo concreto, como um botão em uma tela ou as informações em uma planilha, ou algo mais abstrato, como um algoritmo de classificação ou talvez a sensação de tédio em um personagem de videogame. Uma classe que representa uma planilha pode, por exemplo, ter variáveis que representam os valores de suas células individuais e métodos que executam operações nessas células, como “limpar uma linha” ou “calcular valores”.

Nossa classe HelloJava é um aplicativo Java inteiro em uma única classe. Ele define apenas um método, main() , que contém o corpo do nosso programa:

classe pública HelloJava {

public static void main(String[] args) {

// ...

}

```
}
```

É este método main() que é chamado primeiro quando o aplicativo é iniciado. O bit denominado String [] args nos permite passar argumentos de linha de comando para o aplicativo. Examinaremos o método main() na próxima seção.

Finalmente, embora esta versão do HelloJava não defina nenhuma variável como parte de sua classe, ela usa duas variáveis, frame e label, dentro de seu método main().

Teremos mais a dizer sobre variáveis em breve também.

O método main()

Como você viu quando executamos nosso exemplo, executar um aplicativo Java significa escolher uma classe específica e passar seu nome como argumento para a máquina virtual Java. Quando fizemos isso, o comando java procurou em nossa classe HelloJava para ver se ela continha o método especial chamado main() com o formato correto. Isso aconteceu e o método foi executado. Se main() não estivesse lá, teríamos

recebido uma mensagem de erro. O método main() é o ponto de entrada para aplicativos. Cada aplicativo Java independente inclui pelo menos uma classe com um método main() que executa as ações necessárias para iniciar o restante do programa.

Nosso método main() configura uma janela (um JFrame) para armazenar a saída visual da classe HelloJava. No momento, main() está fazendo todo o trabalho no aplicativo. Mas em uma aplicação orientada a objetos, normalmente delegamos responsabilidades a muitas

classes diferentes. Na próxima encarnação do nosso exemplo, realizaremos exatamente essa divisão – criando uma segunda classe – e veremos que à medida que o exemplo evolui posteriormente, o método main() permanece mais ou menos o mesmo, simplesmente segurando o procedimento de inicialização.

Vamos percorrer rapidamente nosso método main(), só para você saber o que ele faz.

Primeiro, main() cria um JFrame, a janela que conterá nosso exemplo:

Quadro JFrame = new JFrame("Olá, Java!");

A palavra novo nesta linha de código é muito importante. JFrame é o nome de uma classe que representa uma janela na tela, mas a classe em si é apenas um modelo, como uma planta de construção. A palavra-chave new diz ao Java para alocar memória e realmente criar um objeto JFrame específico. Neste caso, o argumento entre parênteses informa ao JFrame o que exibir em sua barra de título. Poderíamos ter deixado de fora o “Olá, Java!” text e usei parênteses vazios para criar um JFrame sem título, mas apenas porque o JFrame nos permite fazer isso especificamente.

Quando as janelas de moldura são criadas pela primeira vez, elas são muito pequenas.

Antes de mostrarmos o JFrame, vamos definir seu tamanho para algo razoável: frame.setSize(300, 150);

Este é um exemplo de invocação de um método em um objeto específico. Neste caso, o método setSize() é definido pela classe JFrame e afeta o objeto JFrame específico que colocamos no quadro variável. Assim

como o frame, também criamos uma instância do JLabel para manter nosso texto dentro da janela:

Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

JLabelé muito parecido com um rótulo físico. Ele mantém algum texto em uma posição específica – neste caso, em nosso quadro. Este é um conceito muito orientado a objetos: usar um objeto para conter algum texto, em vez de simplesmente invocar um método para “desenhar” o texto e seguir em frente. A justificativa para isso ficará mais clara posteriormente.

A seguir, temos que colocar o rótulo no quadro que criamos:

frame.add(rótulo);

Aqui, estamos chamando um método chamado add() para colocar nosso rótulo dentro do JFrame. O JFrame é um tipo de contêiner que pode conter coisas. Falaremos mais

sobre isso mais tarde. A tarefa final de main() é mostrar a janela do quadro e seu conteúdo, que de outra forma seria invisível. Uma janela invisível torna a aplicação bastante chata:

frame.setVisible (verdadeiro);

Esse é todo o método main(). À medida que avançamos nos exemplos deste capítulo, ele permanecerá praticamente inalterado à medida que a classe HelloJava evoluir em torno dele.

Classes e objetos

Uma classe é um modelo para uma parte de um aplicativo; ele contém métodos e variáveis que compõem esse componente. Muitas cópias de trabalho individuais de uma determinada classe podem existir enquanto um aplicativo está ativo. Essas encarnações individuais são chamadas de instâncias da classe ou objetos. Duas instâncias de uma determinada classe podem conter dados diferentes, mas sempre possuem os mesmos métodos.

Por exemplo, considere uma classe Button. Existe apenas uma classe Button, mas um aplicativo pode criar muitos objetos Button diferentes, cada um deles uma instância da mesma classe. Além disso, duas instâncias de Button podem conter dados diferentes, talvez dando a cada uma delas uma aparência diferente e executando uma ação diferente. Nesse sentido, uma classe pode ser considerada um molde para fazer o objeto que representa, algo como um cortador de biscoitos que grava instâncias funcionais de si mesmo na memória do computador. Como você verá mais tarde, há um pouco mais do que isso – uma classe pode de fato compartilhar informações entre suas instâncias – mas esta explicação é suficiente por enquanto.capítulo 5tem toda a história sobre classes e objetos.

O termo objeto em Java é muito geral e às vezes é usado quase como sinônimo de classe. Objetos são entidades abstratas às quais todas as linguagens orientadas a objetos se referem de uma forma ou de outra. Usaremos objeto como um termo genérico para uma instância de uma classe. Poderíamos, portanto, nos referir a uma instância da classe Button como um botão, um objeto Button ou, indiscriminadamente, como um objeto. Você verá o termo usado com frequência nos próximos capítulos,

ecapítulo 5entrarei em muito mais detalhes sobre classes e objetos.

O método main() do exemplo anterior cria uma única instância da classe JLabel e a mostra em uma instância da classe JFrame. Você poderia modificar main() para criar muitas instâncias do JLabel, talvez cada uma em uma janela separada.

Variáveis e tipos de classe

Em Java, cada classe define um novo tipo (tipo de dados). Você pode declarar uma variável desse tipo e então ela pode conter instâncias dessa classe. Uma variável poderia, por exemplo, ser do tipo Button e conter uma instância da classe Button, ou do tipo SpreadSheetCell e conter um objeto SpreadSheetCell, assim como poderia ser

qualquer um dos tipos mais simples, como int ou char. O fato de variáveis terem tipos e não poderem simplesmente conter qualquer tipo de objeto é outra característica importante do Java que garante a segurança e a correção do código.

Deixando de lado as variáveis usadas dentro do método main() por enquanto, apenas uma outra variável é declarada em nosso exemplo simples de HelloJava. É encontrado na declaração do próprio método main():

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

// ...

}
```

Assim como as funções em outras linguagens, um método em Java declara uma lista de parâmetros (variáveis) que aceita como argumentos e especifica os tipos desses parâmetros. Nesse caso, o método principal exige que, ao ser invocado, seja passado um array de objetos String na variável chamada args. A String é o objeto fundamental que representa o texto em Java. Como sugerimos anteriormente, Java usa o parâmetro args para passar quaisquer argumentos de linha de comando fornecidos à máquina virtual Java para seu aplicativo. (Não os usamos aqui, mas usaremos mais tarde.) Até este ponto, falamos vagamente sobre variáveis como objetos de retenção. Na realidade, variáveis que possuem tipos de classe não contêm objetos – elas se referem a eles. Uma referência é um ponteiro ou um identificador para um objeto. Se você declarar uma variável do tipo classe sem atribuir um objeto a ela, ela não apontará para nada. É atribuído o valor padrão nulo, que significa “sem valor”. Se você tentar usar uma variável com valor nulo como se ela estivesse apontando para um objeto real, ocorrerá um erro de tempo de execução, NullPointerException.

É claro que as referências a objetos precisam vir de algum lugar. Em nosso exemplo, criamos dois objetos usando o operador new. Examinaremos a criação de objetos com mais detalhes um pouco mais adiante neste capítulo.

HelloComponent

Até agora, nosso exemplo HelloJava se continha em uma única classe. Na verdade, devido à sua natureza simples, ele serviu apenas como um método único e amplo.

Embora tenhamos usado alguns objetos para exibir nossa mensagem GUI, nosso próprio código não ilustra nenhuma estrutura orientada a objetos.

Bem, vamos corrigir isso agora adicionando uma segunda classe. Para nos dar algo em que construir ao longo deste capítulo, vamos assumir o trabalho da classe JLabel (tchau, JLabel!) e substituí-la por nossa própria classe gráfica: HelloComponent. Nossa classe HelloComponent começará de forma simples, apenas exibindo nosso “Hello, Java!” mensagem em uma posição fixa. Adicionaremos recursos mais tarde.

O código da nossa nova classe é simples; precisamos apenas de mais algumas linhas.

Primeiro, precisamos de outra instrução import no topo do arquivo HelloJava.java: importar java.awt.*;

Esta linha informa ao compilador onde encontrar as classes extras que precisamos para preencher a lógica de HelloComponent. E aqui está essa lógica:

classe HelloComponent estende JComponent {

public void paintComponent(Gráficos g) {

g.drawString("Olá, Java!", 125, 95);

}

}

A definição da classe HelloComponent pode ficar acima ou abaixo da nossa classe HelloJava. Então, para usar nossa nova classe no lugar do JLabel, basta substituir as

duas linhas que fazem referência ao rótulo no método main() por:

```
// Exclua ou comente essas duas linhas

//Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

//frame.add(rótulo);

// E adicione esta linha

frame.add(new HelloComponent());
```

Desta vez, ao compilar HelloJava.java, dê uma olhada nos arquivos .class gerados.

(Esses arquivos estarão localizados em seu diretório atual se você estiver usando um terminal ou na pasta Learn Java/out/production/Learn Java onde você escolheu colocar os projetos IDEA. No próprio IDEA, você também pode expandir a pasta out em o painel de navegação do projeto no lado esquerdo.) Independentemente de como você organizou as classes em sua fonte, você deverá ver dois arquivos de classe binários: HelloJava.class e HelloComponent.class. A execução do código deve se parecer muito com a versão JLabel, mas se você redimensionar a janela, notará que nosso novo componente não se ajusta automaticamente para centralizar o texto.

Então, o que fizemos e por que nos esforçamos tanto para insultar o componente JLabel perfeitamente bom? Criamos nossa nova classe HelloComponent, estendendo uma classe gráfica genérica chamada JComponent. Estender uma classe significa simplesmente adicionar funcionalidade a uma classe existente, criando uma nova.

Entraremos mais nesse processo na próxima seção.

Em nosso exemplo atual, criamos um novo tipo de JComponent que contém um método chamado paintComponent(), responsável por desenhar nossa mensagem. O

método paintComponent() recebe um argumento chamado (um tanto concisamente) g, que é do tipo Graphics. Quando o método paintComponent() é invocado, um objeto Graphics é atribuído a g, que usamos no corpo do método. Falaremos mais sobre paintComponent() e a classe Graphics em um momento. Quanto ao motivo, você entenderá quando adicionarmos todos os tipos de novos recursos ao nosso novo componente mais tarde.

Herança

As classes Java são organizadas em uma hierarquia pai-filho na qual o pai e o filho são conhecidos como superclasse e subclasse, respectivamente. Exploraremos mais esses

conceitos emcapítulo 5. Em Java, cada classe possui exatamente uma superclasse (um único pai), mas possivelmente muitas subclasses. A única exceção a esta regra é a classe Object, que fica no topo de toda a hierarquia de classes; não tem superclasse.

(Sinta-se à vontade para dar uma olhada na pequena fatia da hierarquia de classes Java mostrada emFigura 2-17.)

A declaração da nossa classe no exemplo anterior usa a palavra-chave extends para especificar que HelloComponent é uma subclasse da classe JComponent:

classe HelloComponent estende JComponent {...}

Uma subclasse pode herdar algumas ou todas as variáveis e métodos de sua superclasse. A herança fornece à subclasse acesso às variáveis e métodos de sua superclasse como se ela mesma os tivesse declarado. Uma subclasse pode adicionar variáveis e métodos próprios e também pode substituir ou alterar o significado de métodos herdados. Quando usamos uma subclasse, os métodos substituídos são ocultos (substituídos) pelas versões deles da própria subclasse. Dessa forma, a herança fornece um mecanismo poderoso pelo qual uma subclasse pode refinar ou estender a funcionalidade de sua superclasse.

Por exemplo, a hipotética classe de planilha pode ser subclassificada para produzir uma nova classe de planilha científica com constantes especiais integradas. Neste caso, o código-fonte da planilha científica pode declarar variáveis para as constantes especiais, mas a nova classe científica ainda possui todas as variáveis (e métodos) que constituem a funcionalidade normal de uma planilha. Novamente, esses elementos padrão são herdados da classe pai da planilha. Isso também significa que a planilha científica mantém sua identidade de planilha; ele ainda pode fazer tudo o que uma planilha mais simples poderia fazer. Essa ideia, de que uma classe mais específica ainda pode desempenhar todos os deveres de um pai ou ancestral mais geral, tem implicações profundas. Chamamos essa ideia de polimorfismo e continuaremos a explorá-la ao longo do livro. O polimorfismo é um dos fundamentos da programação orientada a objetos.

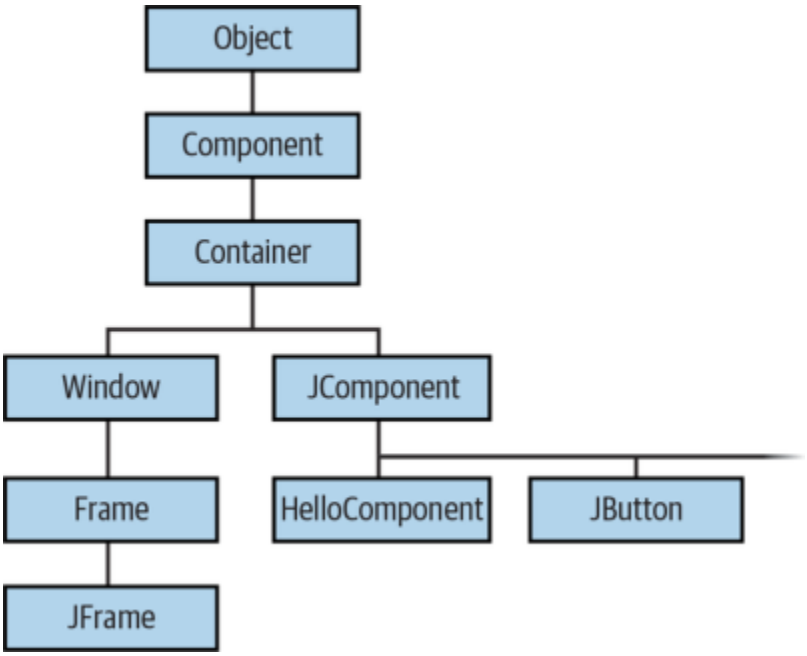
Nossa classe HelloComponent é uma subclasse da classe JComponent e herda muitas variáveis e métodos não

declarados explicitamente em nosso código-fonte. É isso que permite que nossa pequena classe sirva como componente em um JFrame, com apenas algumas customizações.

A classe JComponent

A classe JComponent fornece a estrutura para construir todos os tipos de componentes de UI. Componentes específicos – como botões, rótulos e caixas de listagem – são implementados como subclasses de JComponent.

Mencionamos que as subclasses podem pegar um método herdado e substituí-lo para implementar algum comportamento específico. Mas por que quereríamos mudar o comportamento de algo que presumivelmente já funciona para a superclasse? Muitas aulas começam com funcionalidade mínima. Os programadores originais esperam que



alguém apareça e adicione as partes interessantes. JComponent é exatamente uma dessas classes. Ele cuida

de grande parte da comunicação com o sistema de janelas do computador para você, mas deixa espaço para você adicionar detalhes específicos de apresentação e comportamento.

O método paintComponent() é um método importante da classe JComponent; nós o substituímos para implementar a maneira como nosso componente específico é exibido na tela. O comportamento padrão de paintComponent() não faz nenhum desenho. Se não o tivéssemos substituído em nossa subclasse, nosso componente simplesmente estaria vazio. Aqui, estamos substituindo paintComponent() para fazer algo um pouco mais interessante. Não substituímos nenhum dos outros membros herdados do JComponent porque eles fornecem funcionalidade básica e padrões razoáveis para este exemplo (trivial). À medida que o HelloJava cresce, nos aprofundaremos nos membros herdados e usaremos métodos adicionais. Também adicionaremos alguns métodos e variáveis específicos do aplicativo especificamente para as necessidades do HelloComponent.

JComponenté realmente a ponta de outro iceberg chamado Swing. Swing é o kit de ferramentas de UI do Java, representado em nosso exemplo pela instrução import na parte superior; discutiremos o Swing com mais detalhes emCapítulo 12.

Relacionamentos e acusações

Você pode pensar na subclasse como a criação de um relacionamento “é um”, no qual a subclasse “é um” tipo de sua superclasse. HelloComponent é, portanto, uma espécie de JComponent. Quando nos referimos a um tipo de objeto, queremos dizer qualquer instância da classe

desse objeto ou qualquer uma de suas subclasses. Posteriormente, examinaremos mais de perto a hierarquia de classes Java e veremos que JComponent é em si uma subclasse da classe Container, que é posteriormente derivada de uma classe chamada Component, e assim por diante, conforme mostrado emFigura 2-17.

Nesse sentido, um objeto HelloComponent é uma espécie de JComponent, que é uma espécie de Container, e todos estes podem, em última análise, ser considerados uma espécie de Componente. É dessas classes que HelloComponent herda sua

funcionalidade GUI básica e (como discutiremos mais tarde) a capacidade de ter outros componentes gráficos incorporados a ele também.

Figura 2-17. Parte da hierarquia de classes Java

Componenteé uma subclasse da classe Object de nível superior, portanto, todas essas classes são tipos de Object. Todas as outras classes na API Java herdam o comportamento de Object, que define alguns métodos básicos, como você verá

emcapítulo 5. Continuaremos a usar a palavra objeto (o minúsculo) de maneira genérica para nos referirmos a uma instância de qualquer classe; usaremos Object para nos referirmos especificamente ao tipo dessa classe.

Pacotes e Importações

Mencionamos anteriormente que a primeira linha do nosso exemplo informa ao Java onde encontrar algumas das classes que estamos usando:

importar javax.swing.*;

Especificamente, ele informa ao compilador que usaremos classes do kit de ferramentas Swing GUI (neste caso, JFrame, JLabel e JComponent). Essas classes são organizadas em um pacote Java chamado javax.swing. Em Java, um pacote é um grupo de classes relacionadas por propósito ou por aplicativo. As aulas no mesmo pacote têm privilégios de acesso especiais entre si e podem ser projetadas para trabalharem juntas.

Os pacotes são nomeados de forma hierárquica com componentes separados por pontos, como java.util e java.util.zip. As classes em um pacote normalmente residem em pastas aninhadas que correspondem ao nome do pacote. Eles também assumem o nome do pacote como parte de seu “nome completo” ou, para usar a terminologia adequada, seu nome totalmente qualificado. Por exemplo, o nome completo da classe JComponent é javax.swing.JComponent. Poderíamos ter nos referido a ele diretamente por esse nome, em vez de usar a instrução import:

classe HelloComponent estende javax.swing.JComponent {…}

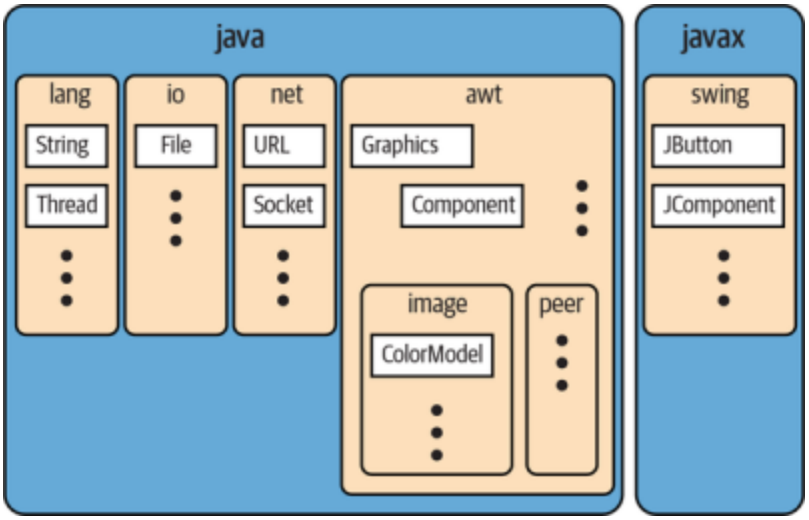
Usar nomes totalmente qualificados pode ser cansativo. A instrução import javax.swing.* nos permite referir-nos a todas as classes do pacote javax.swing por seus nomes simples. Não precisamos usar nomes totalmente qualificados para nos referirmos às classes JComponent, JLabel e JFrame.

Como vimos quando adicionamos nossa segunda classe de exemplo, pode haver uma ou mais instruções de importação em um determinado arquivo fonte Java. As

importações criam efetivamente um “caminho de pesquisa” que informa ao Java onde procurar classes às quais nos referimos por seus nomes simples e não qualificados.

(Não é realmente um caminho, mas evita nomes ambíguos que podem criar erros.) As importações que vimos usam a notação ponto estrela (.*) para indicar que o pacote inteiro deve ser importado. Mas você também pode especificar apenas uma classe. Por exemplo, nosso exemplo atual usa apenas a classe Graphics do pacote java.awt.

Poderíamos ter usado import java.awt.Graphics em vez de usar o curinga * para importar todas as classes do pacote Abstract Window Toolkit (AWT). No entanto, estamos prevendo o uso de várias outras classes deste pacote posteriormente.



O java. e javax. hierarquias de pacotes são especiais. Qualquer pacote que comece com java. faz parte da API Java principal e está disponível em qualquer plataforma que suporte Java. O javax. pacote normalmente denota uma extensão padrão para a plataforma principal, que

pode ou não ser instalada. No entanto, nos últimos anos, muitas extensões padrão foram adicionadas à API Java principal sem renomeá-las. O

pacote javax.swing é um exemplo; faz parte da API principal, apesar do nome.Figura 2-

18ilustra alguns dos principais pacotes Java, mostrando uma ou duas classes representativas de cada um.

Figura 2-18. Alguns pacotes Java principais

java.langcontém classes fundamentais necessárias à própria linguagem Java; este pacote é importado automaticamente e é por isso que não precisamos de uma instrução import para usar nomes de classes como String ou System em nossos exemplos. O pacote java.awt contém classes do antigo sistema de janelas gráficas; java.net contém as classes de rede; e assim por diante.

À medida que você ganha mais experiência com Java, perceberá que ter o comando dos pacotes disponíveis, o que eles fazem e quando e como usá-los é uma parte crítica para se tornar um desenvolvedor Java de sucesso.

O método paintComponent()

A fonte da nossa classe HelloComponent define um método, paintComponent(), que substitui o método paintComponent() da classe JComponent:

```
public void paintComponent(Gráficos g) {

g.drawString("Olá, Java!", 125, 95);

}
```

O método paintComponent() é chamado quando chega a hora de nosso exemplo se desenhar na tela. Leva um único argumento, um objeto Graphics, e não retorna nenhum tipo de valor (void) ao seu chamador.

Modificadores são palavras-chave colocadas antes de classes, variáveis e métodos para alterar sua acessibilidade, comportamento ou semântica. Aqui paintComponent() é declarado como público, o que significa que pode ser invocado (chamado) por métodos em classes diferentes de HelloComponent. Nesse caso, é o ambiente de janelas Java que está chamando nosso método paintComponent(). Um método ou variável declarado como privado, por outro lado, só é acessível a partir de sua própria classe.

O objeto Graphics, uma instância da classe Graphics, representa uma área específica de desenho gráfico. (Também é chamado de contexto gráfico.) Ele contém métodos que podem ser usados para desenhar nesta área e variáveis que representam características como recorte ou modos de desenho. O objeto Graphics específico que passamos no método paintComponent() corresponde à área da tela do nosso HelloComponent, dentro do nosso quadro.

A classe Graphics fornece métodos para renderizar formas, imagens e texto. Em HelloComponent, invocamos o método drawString() do nosso objeto Graphics para rabiscar nossa mensagem nas coordenadas especificadas.

Como vimos anteriormente, acessamos o método de um objeto anexando um ponto (.) e seu nome ao objeto que o contém. Invocamos o método drawString() do objeto Graphics (referenciado por nossa variável g) desta forma:

g.drawString("Olá, Java!", 125, 95);

Aqui podemos ver como a substituição de um método herdado fornece novas funcionalidades. Por si só, uma instância do JComponent não tem ideia de quais informações mostrar ao usuário ou como responder a coisas como cliques do mouse.

Estendemos o JComponent e adicionamos um pouquinho de lógica customizada: mostramos um pouco de texto na tela. Mas podemos fazer muito mais!

HelloJava2: a sequência

Agora que já aprendemos algumas noções básicas, vamos tornar nosso aplicativo um pouco mais interativo. A pequena atualização a seguir nos permite arrastar o texto da mensagem com o mouse. Porém, se você é novo em programação, a atualização pode não parecer tão pequena. Não tenha medo! Examinaremos atentamente todos os tópicos abordados neste exemplo em capítulos posteriores. Por enquanto, aproveite o exemplo e use-o como uma oportunidade para se sentir mais confortável criando e executando programas Java, mesmo que não se sinta tão confortável com o código interno.

Chamaremos este exemplo de HelloJava2 em vez de causar confusão continuando a expandir o antigo, mas as principais mudanças aqui e mais adiante residem na adição de capacidades à classe HelloComponent e simplesmente em fazer as alterações correspondentes nos nomes para mantê-los corretos (por exemplo ,

HelloComponent2, HelloComponent3 e assim por diante). Tendo acabado de ver a herança em funcionamento, você pode se perguntar por que não estamos criando uma subclasse de HelloComponent e explorando a

herança para desenvolver nosso exemplo anterior e estender sua funcionalidade. Bem, neste caso, isso não traria muita vantagem, então, para maior clareza, simplesmente recomeçamos.

Duas barras seguidas indicam que o resto da linha é um comentário. Adicionamos alguns comentários ao HelloJava2 para ajudá-lo a acompanhar tudo:

```
//arquivo: HelloJava2.java

importar java.awt.*;

importar java.awt.event.*;

importar javax.swing.*;

classe pública HelloJava2 {

public static void main(String[] args) {

Quadro JFrame = new JFrame("HelloJava2");

frame.add(new HelloComponent2("Olá, Java!"));

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setSize(300, 300);

frame.setVisible (verdadeiro);

}

}

classe HelloComponent2 estende JComponent

implementa MouseMotionListener {
```

```
String theMessage;

int mensagemX = 125, mensagemY = 95; //
Coordenadas da mensagem

public HelloComponent2(String mensagem) {

aMensagem = mensagem;

addMouseMotionListener(este);

}

public void paintComponent(Gráficos g) {

g.drawString(theMessage, mensagemX, mensagemY);

}

public void mouseDragged(MouseEvent e) {

// Salve as coordenadas do mouse e pinte a mensagem.

mensagemX = e.getX();

mensagemY = e.getY();

repintar();

}

public void mouseMoved(MouseEvent e) { }

}
```

Se você estiver usando IDEA, crie uma nova classe Java chamada HelloJava2 e copie o código acima. Se você preferir o terminal, coloque o texto deste exemplo em um

novo arquivo chamado HelloJava2.java. De qualquer forma, você deseja compilá-lo como antes. Como resultado, você deverá obter novos arquivos de classe, HelloJava2.class e HelloComponent2.class.

Se você estiver seguindo o IDEA, clique no botão Executar próximo a HelloJava2. Se você estiver usando um terminal, execute o exemplo usando o seguinte comando: C:\> java HelloJava2

Sinta-se à vontade para substituir o comentário “Olá, Java!” mensagem e desfrute de muitas horas de diversão, arrastando o texto com o mouse. Observe que agora, ao clicar no botão Fechar da janela, o aplicativo é encerrado corretamente; explicaremos isso mais tarde, quando falarmos sobre eventos. Vamos mergulhar e ver o que mudou.

Variáveis de instância

Adicionamos algumas variáveis à classe HelloComponent2 em nosso exemplo: int mensagemX = 125, mensagemY = 95;

String theMessage;

mensagemXe messageY são inteiros que contêm as coordenadas atuais de nossa mensagem móvel. Nós os definimos com valores padrão que devem colocar a mensagem aproximadamente próximo ao centro da janela. Os inteiros Java são números assinados de 32 bits, portanto, podem conter facilmente nossos valores de coordenadas. A variável theMessage é do tipo String e pode conter instâncias da classe String.

Você deve observar que essas três variáveis são declaradas entre colchetes da definição de classe, mas

não dentro de qualquer método específico dessa classe. Essas variáveis são chamadas de variáveis de instância e pertencem ao objeto como um todo. Especificamente, cópias separadas deles aparecem em cada instância separada da classe. Variáveis de instância são sempre visíveis (e utilizáveis por) todos os métodos dentro de sua classe. Dependendo de seus modificadores, eles também podem ser acessíveis fora da classe.

A menos que inicializadas de outra forma (jargão do programador para definir o primeiro valor em algo), as variáveis de instância são definidas com um valor padrão de 0, falso ou nulo, dependendo do seu tipo. Os tipos numéricos são definidos como 0, as variáveis booleanas são definidas como falso e as variáveis de tipo de classe sempre têm seu valor definido como nulo.

As variáveis de instância diferem dos argumentos do método e de outras variáveis declaradas dentro do escopo de um método específico. Estas últimas são chamadas de variáveis locais. Elas são efetivamente variáveis privadas que podem ser vistas apenas pelo código dentro de um método ou outro bloco de código. Java não inicializa variáveis locais, então você mesmo deve atribuir valores. Se você tentar usar uma variável local à qual ainda não foi atribuído um valor, seu código gerará um erro em tempo de compilação. Variáveis locais permanecem vivas apenas enquanto o método estiver em execução e depois desaparecem, a menos que algo salve seu valor. Cada vez que o método é invocado, suas variáveis locais são recriadas e devem receber valores atribuídos.

Usamos as novas variáveis para tornar nosso método paintComponent(),

anteriormente enfadonho, mais dinâmico. Agora todos os argumentos na chamada para drawString() são determinados por essas variáveis.

Construtores

A classe HelloComponent2 inclui um tipo especial de método chamado construtor. Um construtor é chamado para configurar uma nova instância de uma classe. Quando um novo objeto é criado, Java aloca armazenamento para ele, define as variáveis de instância com seus valores padrão e chama o método construtor da classe para fazer qualquer configuração necessária no nível do aplicativo.

Um construtor sempre tem o mesmo nome de sua classe. Por exemplo, o construtor da classe HelloComponent2 é chamado HelloComponent2(). Os construtores não têm um tipo de retorno, mas você pode pensar neles como criando um objeto do tipo de sua classe. Como outros métodos, os construtores podem ter parâmetros. Sua única missão na vida é configurar e inicializar instâncias de classe recém-nascidas, possivelmente usando informações passadas a elas nesses parâmetros.

Um objeto é criado com o operador new especificando o construtor da classe e quaisquer argumentos necessários.1A instância do objeto resultante é retornada como um valor. Em nosso exemplo, uma nova instância HelloComponent2 é criada no método main() por esta linha:

```
frame.add(new HelloComponent2("Olá, Java!"));
```

Esta linha na verdade faz duas coisas. Para deixar isso mais claro, poderíamos escrevê-los como duas linhas separadas que são um pouco mais fáceis de entender:

HelloComponent2 newObject = new HelloComponent2("Olá, Java!"); frame.add(novoObjeto);

A primeira linha é a mais importante, onde um novo objeto HelloComponent2 é criado. O construtor HelloComponent2 usa uma String como argumento e, conforme organizamos as coisas, usa esse argumento para definir a mensagem que é exibida na janela. Com um pouco de mágica do compilador Java, o texto citado no código-fonte Java é transformado em um objeto String. (VerCapítulo 8para uma discussão mais profunda da classe String.) A segunda linha simplesmente adiciona nosso novo componente ao quadro para torná-lo visível, como fizemos nos exemplos anteriores.

Já que estamos no assunto, se quiser tornar nossa mensagem configurável, você pode alterar a chamada do construtor para o seguinte:

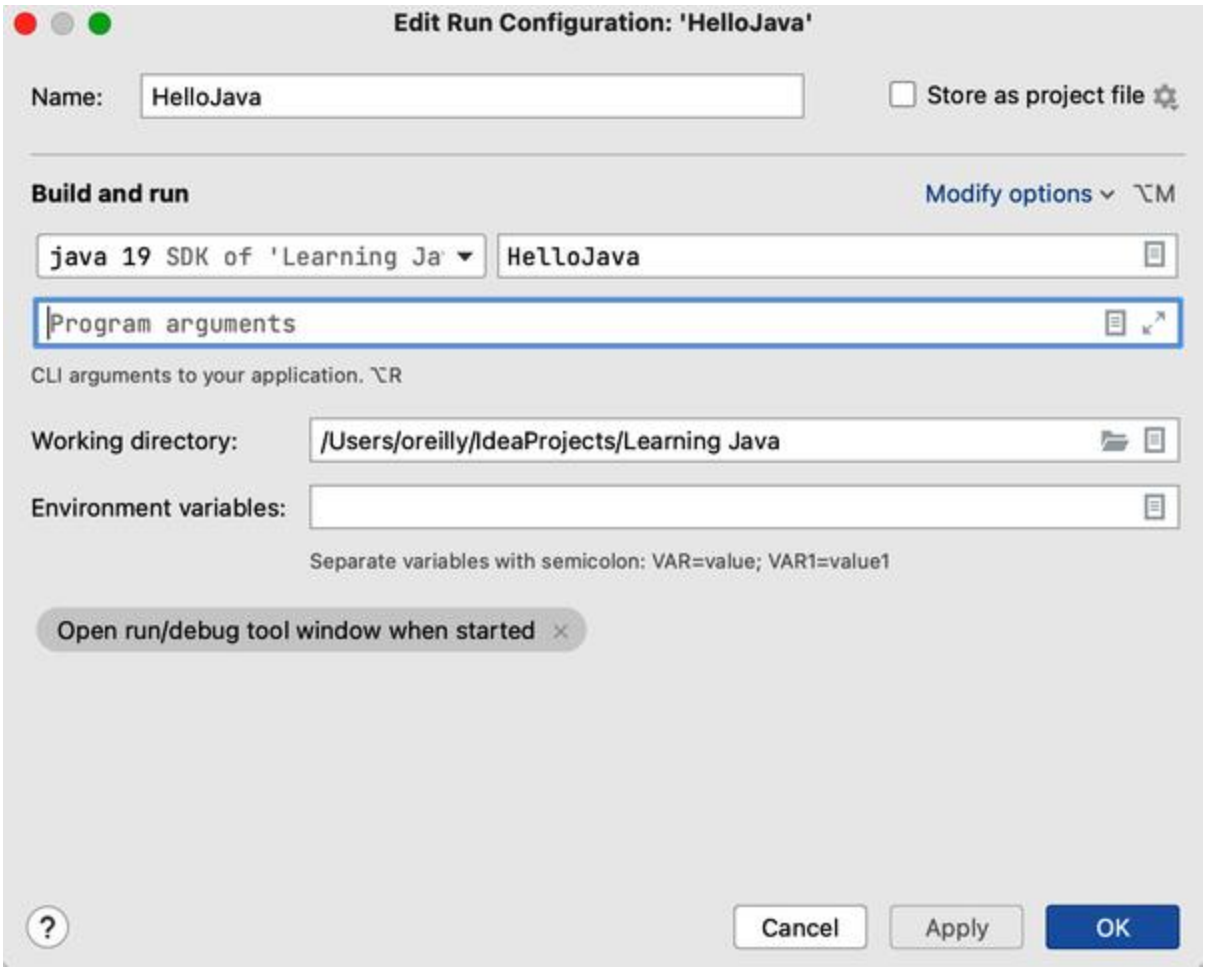
HelloComponent2 newobj = new HelloComponent2(args[0]);

Agora você pode passar o texto na linha de comando ao executar a aplicação usando o seguinte comando:

C:\> java HelloJava2 "Olá, Java!"

argumentos[0]refere-se ao primeiro parâmetro da linha de comando. Seu significado ficará mais claro quando discutirmos arrays emCapítulo 4. Se estiver usando um IDE, você precisará configurá-lo para aceitar seus parâmetros antes de executá-lo. O IntelliJ

IDEA tem algo chamado configuração de execução que você pode editar nos mesmos



menus que aparecem quando você clica nos botões verdes de reprodução. A configuração de execução tem diversas opções, mas nosso interesse está no campo de texto para “Argumentos do Programa”, conforme mostrado emFigura 2-19. Observe que tanto na linha de comando quanto no IDE, você deve colocar sua frase entre aspas duplas para garantir que o texto seja considerado um argumento. Se você deixar as aspas de fora, Olá e Java! seriam dois argumentos separados.

Figura 2-19. Caixa de diálogo IDEA para fornecer parâmetros de linha de comando OláComponent2O construtor de faz duas coisas: define o texto da variável de instância theMessage e chama addMouseMotionListener(). Este método faz parte do

mecanismo de eventos, que discutiremos a seguir. Ele diz ao sistema: “Ei, estou interessado em qualquer coisa que aconteça envolvendo o movimento do mouse”: public HelloComponent2(String mensagem) {

aMensagem = mensagem;

addMouseMotionListener(este);

}

A variável especial somente leitura chamada this é usada para se referir explicitamente ao nosso objeto (o contexto do objeto “atual”) na chamada para addMouseMotionListener(). Um método pode usar isso para se referir à instância do objeto que o contém. As duas instruções a seguir são, portanto, formas equivalentes de atribuir o valor à variável de instância Message:

aMensagem = mensagem;

ou:

this.theMessage = mensagem;

Normalmente usaremos a forma mais curta e implícita para nos referirmos a variáveis de instância, mas precisaremos disso quando tivermos que passar explicitamente uma referência ao nosso objeto para um método em outra classe. Frequentemente passamos tais referências para que métodos em outras classes possam invocar nossos métodos públicos ou usar nossas variáveis públicas.

Eventos

Os dois últimos métodos de HelloComponent2, mouseDragged() e mouseMoved(), dizem ao Java para transmitir qualquer informação que possa obter do mouse. Cada vez que o usuário executa uma ação, como pressionar uma tecla no teclado, mover o mouse ou talvez bater a cabeça em uma tela sensível ao toque, o Java gera um evento.

Um evento representa uma ação que ocorreu; contém informações sobre a ação, como horário e local. A maioria dos eventos está associada a um componente GUI específico em um aplicativo. Um pressionamento de tecla, por exemplo, pode corresponder a um caractere digitado em um campo de entrada de texto específico. Clicar em um botão do mouse pode ativar um botão específico na tela. Até mesmo mover o mouse dentro de uma determinada área da tela pode desencadear efeitos como realçar texto ou alterar a forma do cursor.

Para trabalhar com esses eventos, importamos um novo pacote, java.awt.event, que fornece objetos Event específicos que usamos para obter informações do usuário.

(Observe que importar java.awt.* não importa automaticamente o pacote de eventos.

As importações não são recursivas. Os pacotes não contêm realmente outros pacotes, mesmo que o esquema de nomenclatura hierárquica implique que sim.)

Existem dezenas de classes de eventos, incluindo MouseEvent, KeyEvent e Action Event. Na maior parte, o significado destes eventos é bastante intuitivo. Um MouseEvent ocorre quando o usuário faz algo com o mouse, um KeyEvent ocorre quando o usuário pressiona

ou solta uma tecla e assim por diante. ActionEvent é um pouco especial; veremos isso em açã[o emCapítulo 12. Por](#) enquanto, vamos nos concentrar em lidar com MouseEvents.

Os componentes GUI em Java geram eventos para tipos específicos de ações do usuário. Por exemplo, se você clicar com o mouse dentro de um componente, o componente gerará um evento de mouse. Os objetos podem solicitar o recebimento de eventos de um ou mais componentes registrando um ouvinte na fonte do evento. Por exemplo, para declarar que um ouvinte deseja receber eventos de movimento do mouse de um componente, você invoca o método addMouseMotionListener() desse componente, especificando o objeto ouvinte como um argumento. É isso que nosso exemplo está fazendo em seu construtor. Nesse caso, o componente está chamando seu próprio método addMouseMotionListener(), com o argumento this, que significa

“Quero receber meus próprios eventos de movimento do mouse”.

É assim que nos registramos para receber eventos. Mas como podemos realmente obtê-los? É para isso que servem os dois métodos relacionados ao mouse em nossa classe. O método mouseDragged() é chamado automaticamente em um ouvinte para receber os eventos gerados quando o usuário arrasta o mouse – ou seja, move o mouse com qualquer botão clicado. O método mouseMoved() é chamado sempre que o usuário move o mouse sobre a área sem clicar em um botão.

Nesse caso, colocamos esses métodos em nossa classe HelloComponent2 e fizemos com que ela se registrasse

como ouvinte. Isso é totalmente apropriado para nosso novo componente de arrastar texto. De modo mais geral, um bom design geralmente

determina que os ouvintes de eventos sejam implementados como classes adaptadoras que fornecem melhor separação entre GUI e “lógica de negócios”. Uma classe adaptadora é uma classe intermediária conveniente que implementa todos os métodos de uma interface com algum comportamento padrão. Discutiremos eventos, ouvintes e adaptadores em detalhes emCapítulo 12.

Nosso método mouseMoved() é chato: ele não faz nada. Ignoramos movimentos simples do mouse e reservamos nossa atenção para arrastar. Mas temos que fornecer algum tipo de implementação – mesmo que vazia – já que a interface

MouseMotionListener a inclui. Nosso método mouseDragged(), por outro lado, tem alguma importância. Este método é chamado repetidamente pelo sistema de janelas para nos fornecer atualizações sobre a posição do mouse conforme o usuário o arrasta. Aqui está:

```
public void mouseDragged(MouseEvent e) {

mensagemX = e.getX();

mensagemY = e.getY();

repintar();

}
```

O único parâmetro para mouseDragged() é um objeto MouseEvent, e, que contém todas as informações que precisamos saber sobre este evento. Pedimos ao MouseEvent que nos informe as coordenadas xey da posição atual do mouse chamando seus métodos getX() e getY(). Nós os salvamos nas variáveis de instância messageX e messageY para uso em outro lugar.

A beleza do modelo de evento é que você precisa lidar apenas com os tipos de eventos que deseja. Se você não se importa com eventos de teclado, simplesmente não registra um ouvinte para eles; o usuário pode digitar tudo o que quiser e você não será incomodado. Se não houver ouvintes para um determinado tipo de evento, o Java nem mesmo o gerará. O resultado é que o tratamento de eventos é bastante eficiente.2

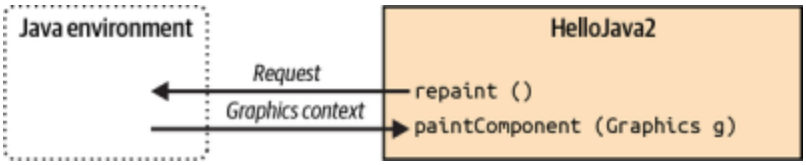
Enquanto discutimos eventos, devemos mencionar outra pequena adição que inserimos no HelloJava2:

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

Esta linha informa ao quadro para sair do aplicativo quando o botão Fechar for clicado. É chamada de operação de fechamento “padrão” porque esta operação, como quase todas as outras interações da GUI, é governada por eventos. Poderíamos registrar um ouvinte de janela para ser notificado quando o usuário clicar no botão Fechar e realizar qualquer ação que desejarmos, mas esse método conveniente lida com os casos comuns.

Finalmente, dançamos em torno de algumas outras questões aqui. Como o sistema sabe que nossa classe contém os métodos mouseDragged() e mouseMoved() necessários? De onde vêm esses nomes? E por que

temos que fornecer um método mouseMoved() que não faz nada? A resposta a essas perguntas tem a ver com



interfaces. Abordaremos as interfaces depois de esclarecer alguns assuntos inacabados com repaint().

O método repaint()

Como alteramos as coordenadas da mensagem quando arrastamos o mouse,

gostaríamos que HelloComponent2 se redesenhasse. Fazemos isso chamando repaint(), que pede ao sistema para redesenhar a tela posteriormente. Não podemos chamar paintComponent() diretamente, mesmo que quiséssemos, porque não temos um contexto gráfico para passar para ele.

Podemos usar o método repaint() da classe JComponent para solicitar que nosso componente seja redesenhado. repaint() faz com que o sistema de janelas Java agende uma chamada para nosso método paintComponent() no próximo horário possível; Java fornece o objeto Graphics necessário, conforme mostrado emFigura 2-20.

Figura 2-20. Invocando o método repaint()

Este modo de operação não é apenas um inconveniente causado por não ter o contexto gráfico correto à mão. Sua maior vantagem é que o comportamento de repintura é controlado por outra pessoa, enquanto somos livres para cuidar de nossos negócios. O sistema Java

possui um thread de execução separado e dedicado que lida com todas as solicitações repaint(). Ele pode agendar e consolidar solicitações repaint() conforme necessário, o que ajuda a evitar que o sistema de janelas fique sobrecarregado durante situações de pintura intensa, como rolagem. Outra vantagem é que toda a funcionalidade de pintura deve ser encapsulada através do nosso método paintComponent(); não ficamos tentados a espalhá-lo por todo o aplicativo (o que pode dificultar a manutenção).

Interfaces

Agora é hora de abordar algumas das questões que evitamos anteriormente: como o sistema sabe como chamar mouseDragged() quando ocorre um evento de mouse? É

simplesmente uma questão de saber que mouseDragged() é algum nome mágico que nosso método de manipulação de eventos deve ter? Não exatamente; a resposta aborda a discussão sobre interfaces, que são um dos recursos mais importantes da linguagem Java.

O primeiro sinal de uma interface vem na linha de código que apresenta a classe HelloComponent2. Dizemos que a classe implementa a interface

MouseMotionListener:

class HelloComponent2 estende JComponent implementa MouseMotionListener {

// ...

public void mouseMoved(MouseEvent e) {

```
// Sua própria lógica vai aqui

}

public void mouseDragged(MouseEvent e) {

// Sua própria lógica vai aqui

}

}
```

Essencialmente, uma interface é uma lista de métodos que a classe deve ter; esta interface específica requer que nossa classe tenha métodos chamados

mouseDragged() e mouseMoved(). A interface não diz o que esses métodos devem fazer; na verdade, nosso mouseMoved() não faz absolutamente nada. Diz que os métodos devem receber um MouseEvent como argumento e não retornar nenhum valor (é isso que void significa).

Uma interface é um contrato entre você, o desenvolvedor do código e o compilador.

Ao dizer que sua classe implementa a interface MouseMotionListener, você está dizendo que esses métodos estarão disponíveis para chamadas de outras partes do sistema. Se você não os fornecer, ocorrerá um erro de compilação. É por isso que precisamos de um método mouseMoved(); mesmo que o que fornecemos não faça nada, a interface MouseMotionListener diz que devemos ter um.

A distribuição Java vem com muitas interfaces que definem o que as classes devem fazer. Essa ideia de

contrato entre o compilador e uma classe é muito importante.

Existem muitas situações como a que acabamos de ver em que você não se importa com a classe de algo; você apenas se importa que ele tenha alguma capacidade, como ouvir eventos do mouse. As interfaces nos dão uma maneira de agir sobre objetos com base em suas capacidades, sem saber ou nos preocupar com seu tipo real. Eles são um conceito tremendamente importante na forma como usamos Java como uma

linguagem orientada a objetos. Falaremos sobre eles em detalhes emcapítulo 5.

capítulo 5também discute como as interfaces fornecem uma espécie de cláusula de escape para a regra Java de que qualquer nova classe pode estender apenas uma única classe (“herança única”). Uma classe em Java pode estender apenas uma classe, mas pode implementar quantas interfaces desejar. As interfaces podem ser usadas como tipos de dados, podem estender outras interfaces (mas não classes) e podem ser herdadas por classes (se a classe A implementa a interface B, as subclasses de A também implementam B). A diferença crucial é que as classes não herdam realmente métodos de interfaces; as interfaces apenas especificam os métodos que a classe deve ter.

Adeus e Olá de novo

Bem, é hora de dizer adeus ao HelloJava. Esperamos que você tenha desenvolvido uma noção de alguns dos recursos da linguagem Java e dos fundamentos para escrever e executar um programa Java. Esta breve introdução deve ajudá-lo a explorar os detalhes da

programação com Java. Se você está um pouco confuso com parte do material apresentado aqui, anime-se. Estaremos cobrindo todos os principais tópicos

apresentados aqui novamente em seus próprios capítulos ao longo do livro. Este tutorial foi concebido para ser uma espécie de “prova de fogo” para colocar conceitos e terminologia importantes em seu cérebro, para que na próxima vez que você os ouvir, você tenha uma vantagem inicial.

Conheceremos melhor as ferramentas do mundo Java no próximo capítulo. Veremos detalhes sobre os comandos que já introduzimos, como javac, bem como abordaremos outros utilitários importantes. Continue lendo para dizer olá a vários de seus novos melhores amigos como desenvolvedor Java!

Perguntas de revisão

Aqui estão algumas perguntas de revisão para garantir que você captou os principais tópicos deste capítulo:

1. Qual comando você usa para compilar um arquivo fonte Java?

2. Como a JVM sabe por onde começar quando você executa uma classe Java?

3. Você pode estender mais de uma classe ao criar uma nova classe?

4. Você pode implementar mais de uma interface ao criar uma nova classe?

5. Qual classe representa a janela principal de uma aplicação gráfica?

Exercícios de código

E para seu primeiro exercício de programação,3crie uma classe GoodbyeJava que funcione exatamente como o primeiro programa HelloJava de“Executando o Projeto”,

mas exibe a mensagem “Adeus, Java!” em vez de. Experimente a versão de linha de comando ou a versão gráfica – ou ambas! Sinta-se à vontade para copiar o programa original o quanto desejar. Certifique-se de compilar e executar sua classe GoodbyeJava para ajudar a praticar o processo de execução de um aplicativo Java. Você certamente adquirirá mais prática ao longo do caminho, mas adquirir um pouco mais de familiaridade com seu IDE ou com os comandos javac e java agora o ajudará ao ler os próximos capítulos.

1Os termos parâmetro e argumento geralmente são usados de forma intercambiável.

Tudo bem, mas tecnicamente você fornece parâmetros com um tipo e um nome ao definir métodos ou construtores. Você fornece argumentos para preencher esses parâmetros ao chamar o método ou construtor.

2A manipulação de eventos em Java 1.0 era uma história muito diferente. No início, Java não tinha noção de ouvintes de eventos e todo o tratamento de eventos acontecia substituindo métodos em classes GUI básicas. Isto foi ineficiente e levou a um design deficiente, com uma proliferação de componentes altamente especializados.

3Você pode encontrar as soluções para os desafios de programação de cada capítulo na pasta de exercícios do código-fonte.Apêndice Acontém detalhes sobre como baixar

e usar o código-fonte.Apêndice Bconté m respostas às perguntas do final do capítulo, bem como dicas sobre soluções de código para cada capítulo.

Capítulo 3. Ferramentas do Comércio

Embora você quase certamente faça a maior parte do seu desenvolvimento Java em um IDE como VS Code ou IntelliJ IDEA, todas as ferramentas básicas necessárias para construir aplicativos Java estão incluídas no JDK que você baixou em“Instalando o

JDK”. Quando escrevemos código-fonte Java, é o compilador Java – javac – que transforma nossa fonte em bytecode utilizável. Quando queremos testar esse bytecode, é o próprio comando Java – java – que usamos para executar nossos programas. Quando todas as nossas classes estão compiladas e funcionando juntas, é a ferramenta de arquivamento Java — jar — que nos permite agrupar essas classes para distribuição. Neste capítulo, discutiremos algumas dessas ferramentas de linha de comando que você pode usar para compilar, executar e empacotar aplicativos Java.

Existem muitas ferramentas de desenvolvedor adicionais incluídas no JDK, como jshell para trabalho interativo ou javap para descompilar arquivos de classe. Não teremos tempo para discutir todos eles neste livro, mas em qualquer lugar que outra ferramenta possa ser útil, nós a mencionaremos. (E com certeza estaremos olhando para o jshell. É ótimo para testar rapidamente uma nova classe ou método.) Queremos que você se sinta confortável com essas ferramentas de linha de comando, mesmo que normalmente não trabalhe em um terminal ou janela de comando. Alguns recursos dessas ferramentas não são facilmente acessíveis por meio de

IDEs. Você também pode encontrar momentos em que um IDE seja impraticável ou totalmente indisponível. Administradores de sistemas e engenheiros de DevOps, por exemplo, muitas vezes têm apenas conexões limitadas baseadas em texto com seus servidores executados em data centers sofisticados. Se você precisar corrigir um problema de Java nessa conexão, essas ferramentas de linha de comando serão essenciais.

Ambiente JDK

Depois de instalar o JDK, o comando core java runtime geralmente aparece em seu caminho (disponível para execução) automaticamente, embora nem sempre. Além disso, muitos dos outros comandos fornecidos com o JDK podem não estar disponíveis, a menos que você adicione o diretório bin Java ao caminho de execução.

Para garantir que você tenha acesso a todas as ferramentas, independentemente da sua configuração, os comandos a seguir mostram como configurar corretamente seu ambiente de desenvolvimento no Linux, macOS e Windows. Você define uma nova variável de ambiente para a localização do Java e anexa essa pasta bin à variável de caminho existente. (Os sistemas operacionais usam variáveis de ambiente para armazenar informações que os aplicativos podem usar e potencialmente compartilhar

à medida que são executados.) É claro que você terá que alterar os caminhos em nossos exemplos para corresponder à versão do Java instalada:

#Linux

exportar JAVA_HOME=/usr/lib/jvm/jdk-21-ea14

exportar PATH=$PATH:$JAVA_HOME/bin

Mac OS X

exportar JAVA_HOME=/Users/marc/jdks/jdk-21-ea14/Contents/Home

exportar PATH=$PATH:$JAVA_HOME/bin

Janelas

defina JAVA_HOME=c:\Arquivos de Programas\Java\jdk21

definir PATH=%PATH%;%JAVA_HOME%\bin

No macOS, a situação pode ser mais confusa porque as versões recentes do sistema operacional vêm com “stubs” para os comandos Java instalados. A Apple não fornece mais sua própria implementação de Java, portanto, se você tentar executar um desses comandos, o sistema operacional solicitará que você baixe o Java naquele momento.

Em caso de dúvida, seu teste para determinar se o Java está instalado e qual versão das ferramentas você está usando é usar o sinalizador -version nos comandos java e javac:

% versão java

versão openjdk "21-ea" 19/09/2023

Ambiente de execução OpenJDK (compilação 21-ea+14-1161)

VM de servidor OpenJDK de 64 bits (compilação 21-ea+14-1161, modo misto, compartilhamento)

```
% javac -versão

javac 21-ea
```

O ea na saída da nossa versão indica que esta é uma versão de “acesso antecipado”.

(Java 21 ainda está sendo testado enquanto escrevemos esta edição.)

A máquina virtual Java

Uma máquina virtual Java (VM) é um software que implementa o sistema de tempo de execução Java e executa aplicativos Java. Pode ser um aplicativo independente, como o comando java que vem com o JDK, ou integrado em um aplicativo maior, como um navegador da web. Normalmente o próprio interpretador é uma aplicação nativa, fornecida para cada plataforma, que então inicializa outras ferramentas escritas na linguagem Java. Ferramentas como compiladores Java e IDEs são frequentemente implementadas diretamente em Java para maximizar sua portabilidade e

extensibilidade. Eclipse, por exemplo, é um aplicativo Java puro.

O Java VM executa todas as atividades de tempo de execução do Java. Ele carrega arquivos de classe Java, verifica classes de fontes não confiáveis e executa o bytecode compilado. Ele gerencia a memória e os recursos do sistema. Boas implementações também realizam otimização dinâmica, compilando bytecode Java em instruções de máquina nativas.

Executando aplicativos Java

Uma aplicação Java independente deve ter pelo menos uma classe contendo um método chamado main(), que é o primeiro código a ser executado na inicialização.

Para executar o aplicativo, inicie a VM, especificando essa classe como argumento.

Você também pode especificar opções para o interpretador, bem como argumentos a serem passados para a aplicação:

% java [opções do interpretador] class_name [argumentos do programa]

A classe deve ser especificada como um nome de classe completo, incluindo o nome do pacote, se houver. Observe, entretanto, que você não inclui a extensão de arquivo

.class. Aqui estão alguns exemplos que você pode experimentar no terminal na pasta ch03/examples:

% cd ch03/exemplos

% java animais.birds.BigBird

% java MeuTeste

O intérprete procura a classe no classpath, uma lista de diretórios e arquivos onde as classes são armazenadas. Você pode especificar o caminho de classe por uma variável de ambiente semelhante a JAVA_HOME acima ou com a opção de linha de comando -

classpath. Se ambos estiverem presentes, Java usará a opção de linha de comando.

Discutiremos o caminho de classe em detalhes na próxima seção.

Você também pode usar o comando java para iniciar um arquivo Java ARchive (JAR)

“executável”:

% java -jar spaceblaster.jar

Nesse caso, o arquivo JAR inclui metadados com o nome da classe de inicialização que contém o método main(), e o caminho de classe se torna o próprio arquivo JAR.

Veremos mais de perto os arquivos JAR em“Arquivos JAR”.

Se você estiver trabalhando principalmente em um IDE, lembre-se de que ainda pode tentar os comandos anteriores usando as opções de terminal integradas que mencionamos em“Executando o Projeto”.

Após carregar a primeira classe e executar seu método main(), a aplicação pode referenciar outras classes, iniciar threads adicionais e criar sua interface de usuário ou outras estruturas, conforme mostrado emFigura 3-1.

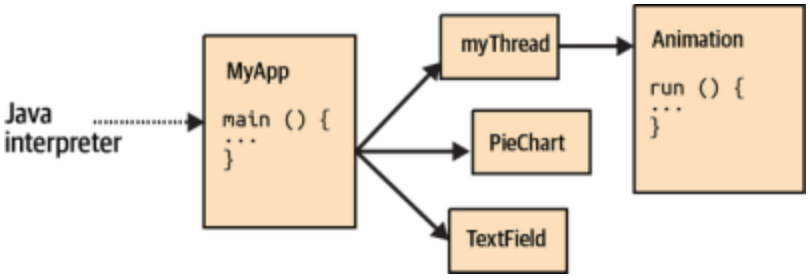


Figura 3-1. Iniciando um aplicativo Java

O método main() deve ter a assinatura de método correta. Uma assinatura de método é o conjunto de

informações que define o método. Inclui o nome do método, argumentos e tipo de retorno, bem como modificadores de tipo e visibilidade. O

método main() deve ser um método público e estático que recebe um array de objetos String como argumento e não retorna nenhum valor (void):

público estático void principal (String [] myArgs)

O fato de main() ser um método público e estático significa simplesmente que é globalmente acessível e pode ser chamado diretamente pelo nome. Discutiremos as implicações dos modificadores de visibilidade, como público, e o significado de estático emCapítulo 4ecapítulo 5.

O único argumento do método main(), o array de objetos String, contém os argumentos da linha de comando passados para o aplicativo. O nome do parâmetro não importa; apenas o tipo é importante. Em Java, o conteúdo de myArgs é um array.

(Mais sobre matrizes emCapítulo 4.) Em Java, os arrays sabem quantos elementos eles contêm e podem fornecer essa informação com prazer:

int numArgs = meusArgs.length;

meusArgs[0]é o primeiro argumento da linha de comando e assim por diante.

O interpretador Java continua a ser executado até que o método main() do arquivo de classe inicial retorne e até que qualquer thread iniciado também saia. (Mais sobre tópicos emCapítulo 9.) Threads especiais designados

como threads daemon são encerrados automaticamente quando o restante do aplicativo é concluído.

Propriedades do sistema

Embora seja possível ler variáveis de ambiente de host em Java, a Oracle desencoraja seu uso para configuração de aplicativos. Em vez disso, Java permite passar qualquer número de valores de propriedades do sistema para o aplicativo quando a VM é iniciada. As propriedades do sistema são simplesmente pares de strings nome-valor que estão disponíveis para o aplicativo por meio do método estático

System.getProperty(). Você pode usar essas propriedades como uma alternativa mais estruturada e portátil para argumentos de linha de comando e variáveis de ambiente para fornecer informações gerais de configuração para seu aplicativo na inicialização.

Você passa cada propriedade do sistema para o interpretador na linha de comando usando a opção -D seguida de nome=valor. Por exemplo:

% java -Dstreet=gergelim -Dscene=beco animais.birds.BigBird

Você pode então acessar o valor da propriedade da rua dentro do seu programa desta forma:

String rua = System.getProperty("rua");

Um aplicativo pode obter sua configuração de inúmeras outras maneiras, é claro, inclusive por meio de arquivos ou pela rede em tempo de execução.

O caminho de classe

O conceito de caminho deve ser familiar para qualquer pessoa que tenha trabalhado em uma plataforma DOS ou Unix. É uma variável de ambiente que fornece ao aplicativo uma lista de locais para procurar algum recurso. O exemplo mais comum é um caminho para programas executáveis. Em um shell Unix, a variável de ambiente PATH é uma lista separada por dois pontos de diretórios que são pesquisados, em ordem, quando o usuário digita o nome de um comando. A variável de ambiente Java CLASSPATH, da mesma forma, é uma lista de locais onde tanto o interpretador quanto o compilador procurarão pacotes e classes Java.

Um elemento do classpath pode ser um diretório ou um arquivo JAR. JARs são arquivos simples que incluem arquivos extras (metadados) que descrevem o conteúdo de cada arquivo. Os arquivos JAR são criados com o utilitário jar do JDK.

Muitas ferramentas para criar arquivos ZIP estão disponíveis publicamente e também podem ser usadas para inspecionar ou criar arquivos JAR.1O formato de arquivo permite que grandes grupos de classes e seus recursos sejam distribuídos em um único arquivo compacto; o tempo de execução Java extrai automaticamente arquivos de classe individuais do arquivo, conforme necessário. Veremos mais de perto os JARs e o comando jar em“O utilitário jar”.

Os meios e formatos precisos para definir o caminho de classe variam de sistema para sistema. Vamos dar uma olhada em como fazer isso.

CLASSPATH no Unix e macOS

Em um sistema Unix (incluindo macOS), você define a variável de ambiente CLASSPATH com uma lista de diretórios e arquivos de classe separados por dois pontos:

% exportar CLASSPATH=/home/vicky/Java/classes:/home/josh/lib/foo.jar:.

Este exemplo especifica um caminho de classe com três locais: um diretório na página inicial do usuário, um arquivo JAR no diretório de outro usuário e o diretório atual, que é sempre especificado com um ponto (.). O último componente do classpath, o diretório atual, é útil quando você está mexendo nas classes.

CLASSPATH no Windows

Em um sistema Windows, a variável de ambiente CLASSPATH é definida com uma lista de diretórios e arquivos de classe separados por ponto e vírgula: C:\> set CLASSPATH=C:\home\vicky\Java\classes;C:\home\josh\lib\foo.jar;.

O iniciador Java e as outras ferramentas de linha de comando sabem como localizar as classes principais, que são as classes incluídas em cada instalação Java. As classes nos pacotes java.lang, java.io, java.net e javax.swing, por exemplo, são todas classes principais,

portanto você não precisa incluir uma biblioteca ou diretório para essas classes em seu classpath.

Curingas CLASSPATH

A variável de ambiente CLASSPATH também pode incluir caracteres curinga “*” que correspondem a todos os arquivos JAR em um diretório. Por exemplo:

% exportar CLASSPATH=/home/sarah/libs/*

Para encontrar outras classes, o interpretador Java pesquisa os elementos do classpath na ordem em que estão listados. A procura combina a localização do caminho e os componentes do nome completo da classe. Por exemplo, considere uma pesquisa pela classe Animals.birds.BigBird, conforme mostrado emFigura 3-2.

Pesquisar o diretório classpath /usr/lib/java significa que o interpretador procura um arquivo de classe individual em /usr/lib/java/animals/birds/BigBird.class. Pesquisar um arquivo ZIP ou JAR no caminho de classe, digamos /home/sarah/zoo.jar, significa que o intérprete procura o arquivo Animals/birds/BigBird.class dentro desse arquivo.

Figura 3-2. Encontrando um nome totalmente qualificado no caminho de classe Para o tempo de execução Java, java, e o compilador Java, javac, o caminho de classe também pode ser especificado com a opção -classpath. Em uma máquina Linux ou macOS, por exemplo:

% javac -classpath /home/pat/classes:/utils/utils.jar:. Foo.java

É essencialmente o mesmo no Windows, mas você deve seguir o separador de caminho do sistema (ponto e

vírgula) e usar letras de unidade para iniciar caminhos absolutos.

Se você não especificar a variável de ambiente CLASSPATH ou a opção de linha de comando, o caminho de classe será padronizado para o diretório atual (.); isso significa que os arquivos em seu diretório atual estão normalmente disponíveis. Se

você alterar o classpath e não incluir o diretório atual, esses arquivos não estarão mais acessíveis.

Suspeitamos que muitos dos problemas que os recém-chegados enfrentam quando aprendem Java pela primeira vez estão relacionados ao caminho de classe. Preste atenção especial à configuração e verificação do caminho de classe ao começar. Se você estiver trabalhando dentro de um IDE, isso poderá remover parte ou toda a carga de gerenciamento do caminho de classe. Em última análise, porém, compreender o caminho de classe e saber exatamente o que ele contém quando seu aplicativo é executado é muito importante para sua sanidade a longo prazo.

Módulos

Java 9 introduziu a abordagem de módulos para aplicativos Java. Os módulos permitem implantações de aplicativos mais refinadas e com melhor desempenho, mesmo quando o aplicativo em questão é (muito) grande. (Módulos não são necessários, mesmo para aplicativos grandes. Você pode continuar usando a abordagem clássica do caminho de classe se ela atender às suas necessidades.) O uso de módulos requer configuração extra, por isso não iremos abordá-los neste

livro, mas aplicativos maiores, distribuídos comercialmente, podem ser baseado em módulo.

Você pode conferir *Modularidade Java 9* por Paul Bakker e Sander.

O compilador Java

O utilitário de linha de comando javac é o compilador no JDK. O compilador é escrito inteiramente em Java, portanto está disponível para qualquer plataforma que suporte o sistema de tempo de execução Java. javac transforma o código-fonte Java em uma classe compilada que contém bytecode Java. Por convenção, os arquivos de origem são nomeados com uma extensão .java; os arquivos de classe resultantes têm uma extensão .class. Cada arquivo de código-fonte é considerado uma única unidade de compilação. (Como você verá em capítulo 5, as classes em uma determinada unidade de compilação compartilham certos recursos, como instruções package e import.) *javac* permite uma classe pública por arquivo e insiste que o arquivo deve ter o mesmo nome da classe. Se o nome do arquivo e o nome da classe não corresponderem, o javac emitirá um erro de compilação. Um único arquivo pode conter diversas classes, desde que apenas uma das classes seja pública e tenha o nome do arquivo. Evite agrupar muitas classes em um único arquivo de origem. Empacotar classes em um arquivo

.java as associa apenas superficialmente.

Vá em frente e crie um novo arquivo chamado Bluebird.java na pasta

ch03/examples/animals/birds. Você pode usar seu IDE para esta etapa ou simplesmente abrir qualquer editor

de texto antigo e criar um novo arquivo. Depois de criar o arquivo, coloque o seguinte código-fonte no arquivo:

pacote animais.pássaros;

classe pública Bluebird {

}

Em seguida, compile-o com:

% cd ch03/exemplos

% javac animais/pássaros/Bluebird.java

Nosso pequeno arquivo ainda não faz nada, mas compilá-lo deve funcionar bem. Você não deverá ver nenhum erro.

Ao contrário do interpretador Java, que usa apenas um nome de classe como argumento, javac precisa de um nome de arquivo (incluindo a extensão .java) para ser processado. O comando anterior produz o arquivo de classe Bluebird.class no mesmo diretório do arquivo de origem. Embora seja bom ver o arquivo de classe no mesmo diretório da fonte deste exemplo, para a maioria dos aplicativos reais, você precisa armazenar o arquivo de classe em um local apropriado no caminho de classe.

Você pode usar a opção -d com javac para especificar um diretório alternativo para armazenar os arquivos de classe gerados por javac. O diretório especificado é usado como raiz da hierarquia de classes, portanto, os arquivos .class são colocados nesse diretório ou em um subdiretório, dependendo se a classe está contida em um pacote.

(O compilador cria subdiretórios intermediários automaticamente, se necessário.) Por exemplo, poderíamos usar o seguinte comando para criar o arquivo Bluebird.class em

/home/vicky/Java/classes/animals/birds/Bluebird.class:

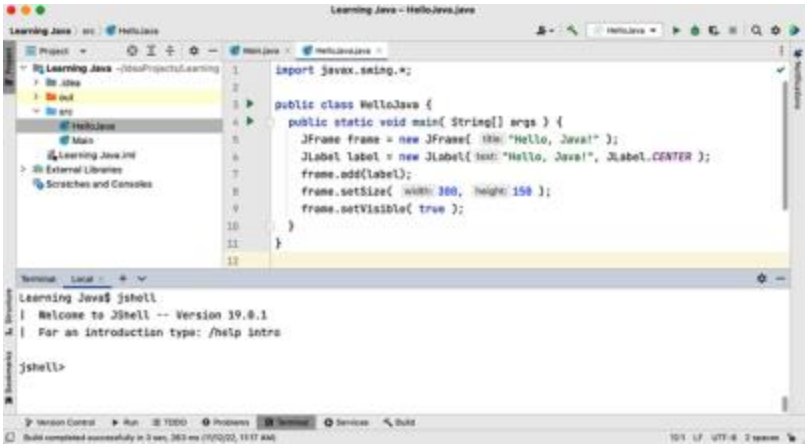
% javac -d /home/vicky/Java/classes Bluebird.java

Você pode especificar vários arquivos .java em um único comando javac; o compilador cria um arquivo de classe para cada arquivo de origem fornecido. Mas você não precisa listar as outras classes que sua classe faz referência, desde que elas estejam no caminho de classe na forma fonte ou compilada. Durante a compilação, Java resolve todas as outras referências de classe usando o classpath.

O compilador Java é mais inteligente que o compilador comum. Por exemplo, javac compara os tempos de modificação dos arquivos de origem e de classe para todas as classes e os recompila conforme necessário. Uma classe Java compilada lembra o arquivo fonte a partir do qual foi compilada e, desde que o arquivo fonte esteja disponível, javac pode recompilá-lo, se necessário. Se, no exemplo anterior, a classe BigBird faz referência a outra classe, digamos, Animals.furry.Grover, javac procura o arquivo fonte Grover.java em um pacote Animals.furry e recompila o arquivo, se necessário, para trazer o Grover.class arquivo de classe atualizado.

Por padrão, entretanto, javac verifica apenas os arquivos de origem referenciados diretamente de outros arquivos de origem. Isso significa que se você tiver um arquivo de classe desatualizado referenciado apenas por um arquivo de classe atualizado, ele poderá não ser notado e

recompilado. Por essa e muitas outras razões, a maioria dos projetos usa um utilitário de construção real, comoGradlepara gerenciar compilações, empacotamentos e muito mais.



Finalmente, é importante notar que javac pode compilar uma aplicação mesmo que apenas as versões compiladas (binárias) de algumas classes estejam disponíveis. Você não precisa de código-fonte para todos os seus objetos. Os arquivos de classe Java contêm todas as informações de tipo de dados e assinatura de método que os arquivos de origem contêm, portanto, compilar em arquivos de classe binários é tão bom quanto compilar com código-fonte Java. (Se precisar fazer alterações, é claro, você ainda precisará dos arquivos de origem.)

Experimentando Java

O Java 9 introduziu um utilitário chamado jshell, que permite testar pedaços de código Java e ver os resultados imediatamente. jshell é um REPL – um loop de leitura-avaliação-impressão. Muitas linguagens os possuem e, antes do Java 9, havia muitas variações de terceiros disponíveis, mas nada foi incorporado ao

próprio JDK. Vejamos com um pouco mais de cuidado suas capacidades.

Você pode usar um terminal ou janela de comando do seu sistema operacional ou pode abrir uma guia de terminal no IntelliJ IDEA, conforme mostrado emFigura 3-3.

Basta digitar jshell no prompt de comando e você verá algumas informações sobre a versão junto com um rápido lembrete sobre como visualizar a ajuda no REPL.

Figura 3-3. Iniciando jshell dentro do IDEA

Vamos em frente e tentar esse comando de ajuda agora:

| Bem-vindo ao JShell - Versão 19.0.1

| Para uma introdução digite: /help intro

jshell> /ajuda introdução

|

| introdução

| =====

|

| A ferramenta jshell permite executar código Java, obtendo resultados im ediatos.

| Você pode inserir uma definição Java (variável, método, classe, etc),

| como: int x = 8

| ou uma expressão Java, como: x + x

| ou uma instrução ou importação Java.

| Esses pequenos pedaços de código Java são chamados de 'snippets'.

|

| Existem também os comandos da ferramenta jshell que permitem entender e

| controle o que você está fazendo, como: /list

|

| Para obter uma lista de comandos: /help

shell é bastante poderoso e não usaremos todos os seus recursos neste livro. No entanto, certamente o usaremos para testar o código Java aqui e na maior parte dos capítulos restantes. Pense no nosso exemplo HelloJava2,“HelloJava2: A Sequela”. Você pode criar elementos de UI como esse JFrame diretamente no REPL e depois manipulá-los – tudo isso enquanto obtém feedback imediato! Não há necessidade de salvar, compilar, executar, editar, salvar, compilar, executar, etc. Vamos tentar: jshell> quadro JFrame = new JFrame("HelloJava2")

| Erro:

| não consigo encontrar o símbolo

| símbolo: classe JFrame

| Quadro JFrame = new JFrame("HelloJava2");

| ^----^

| Erro:

| não consigo encontrar o símbolo

| símbolo: classe JFrame

| Quadro JFrame = new JFrame("HelloJava2");

| ^----^

Ops! jshell é inteligente e rico em recursos, mas também é bastante literal. Lembre-se que se quiser usar uma classe não incluída no pacote padrão, você deverá importá-la.

Isso é verdade em arquivos de origem Java e ao usar jshell. Vamos tentar de novo: jshell> importar javax.swing.*

jshell> quadro JFrame = new JFrame("HelloJava2")

frame ==>
javax.swing.JFrame[frame0,0,23,0x0,inválido,oculto ...
led=true

]

Isso é melhor. Um pouco estranho, provavelmente, mas melhor. Nosso objeto frame foi criado. Essas informações extras após a seta ==> são apenas detalhes sobre nosso JFrame, como tamanho (0x0) e posição na tela (0,23). Outros tipos de objetos mostrarão outros detalhes. Vamos dar ao nosso quadro alguma largura e altura como fizemos antes e colocá-lo na tela onde possamos vê-lo:

```
jshell> frame.setSize(300.200)

jshell> frame.setLocation(400.400)

jshell> frame.setVisible (verdadeiro)
```

Você deverá ver uma janela aparecer bem diante de seus olhos! Será resplandecente em elegância moderna, como mostrado emFigura 3-4.

Figura 3-4. Mostrando um JFrame do jshell

Aliás, não se preocupe em cometer erros no REPL. Você verá uma mensagem de erro, mas poderá simplesmente corrigir o que estava errado e continuar. Como um exemplo rápido, imagine cometer um erro de digitação ao tentar alterar o tamanho do quadro: jshell> frame.setsize(300.100)

| Erro:

| não consigo encontrar o símbolo

| símbolo: método setsize(int,int)

| quadro.setsize(300.100)

| ^-----------^

Java diferencia maiúsculas de minúsculas, então setSize() não é o mesmo que setsize(). jshell fornece o mesmo tipo de informação de erro que o compilador Java forneceria, mas as apresenta in-line. Corrija esse erro e observe o quadro ficar um pouco menor (Figura 3-5)!

Figura 3-5. Alterando o tamanho do nosso quadro

Incrível! Bem, tudo bem, talvez não seja nada útil, mas estamos apenas começando.

Vamos adicionar algum texto usando a classe JLabel:

jshell> rótulo JLabel = new JLabel("Olá jshell!")

rótulo ==> javax.swing.JLabel[,0,0,0x0, ...rticalTextPosition=CENTRO]

jshell> frame.add(rótulo)

$8 ==> javax.swing.JLabel[,0,0,0x0, ...text=Olá, ...]

Legal, mas por que nossa gravadora não apareceu no quadro? Entraremos em muito mais detalhes sobre isso emCapítulo 11, mas o Java permite que algumas alterações gráficas sejam realizadas antes de realizá-las na tela. Esse pode ser um truque

extremamente eficiente, mas às vezes pode pegar você desprevenido. Vamos forçar o quadro a se redesenhar (Figura 3-6):

jshell> frame.revalidate()

jshell> frame.repaint()

Figura 3-6. Adicionando um JLabel ao nosso quadro

Agora podemos ver nosso rótulo. Algumas ações acionarão automaticamente uma chamada para revalidate() ou repaint(). Qualquer componente já adicionado ao nosso quadro antes de torná-lo visível, por exemplo, apareceria imediatamente quando mostrássemos o quadro. Ou podemos remover o rótulo da mesma forma que o adicionamos. Observe novamente para ver o que acontece quando alteramos o tamanho do quadro imediatamente após remover nosso rótulo (Figura 3-7):

```
jshell> frame.remove(label) // assim como add(), as coisas não mudam imed iatamente

jshell> frame.setSize(400.150)
```

Figura 3-7. Removendo um rótulo e redimensionando nosso quadro

Ver? Temos uma janela nova e mais fina e sem etiqueta – tudo sem repintura forçada.

Trabalharemos mais com elementos de UI em capítulos posteriores, mas vamos tentar mais um ajuste em nosso rótulo apenas para mostrar como é fácil experimentar novas ideias ou métodos que você pesquisou na documentação. Podemos centralizar o texto da etiqueta, por exemplo, resultando em algo comoFigura 3-8:

```
jshell> frame.add(rótulo)

$ 45 ==> javax.swing.JLabel[,0,0,300x278,...,text=Olá jshell!,...]

jshell> frame.revalidate()

jshell> frame.repaint()
```


```
jshell> label.setHorizontalAlignment(JLabel.CENTER)
```

Figura 3-8. Centralizando o texto em nossa etiqueta

Sabemos que este foi outro passeio rápido com vários pedaços de código que talvez ainda não façam sentido. Por que CENTER está em letras maiúsculas? Por que o nome da classe JLabel é usado antes do nosso alinhamento central? Não podemos responder a todas as perguntas agora, mas esperamos que digitar, provavelmente cometer alguns pequenos erros, corrigi-los e ver os resultados faça você querer saber mais.

Queremos ter certeza de que você tem as ferramentas para continuar jogando ao longo do restante deste livro. Como tantas outras habilidades, a programação se beneficia da prática, além da leitura!

Arquivos JAR

Os arquivos Java ARchive (JAR) são as malas do Java. Eles são a maneira padrão e portátil de agrupar todas as partes do seu aplicativo Java em um pacote compacto para distribuição ou instalação. Você pode colocar o que quiser em um arquivo JAR: arquivos de classe Java,

objetos serializados, arquivos de dados, imagens, áudio, etc.

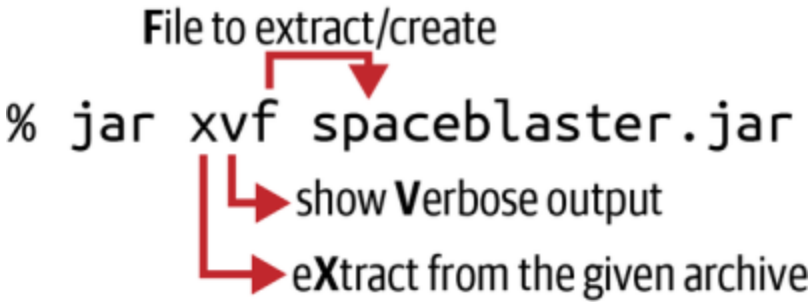
Um arquivo JAR também pode conter uma ou mais assinaturas digitais que atestam sua integridade e autenticidade, anexadas ao arquivo como um todo ou para itens individuais no arquivo.

O sistema de tempo de execução Java pode carregar arquivos de classe diretamente de um arquivo na variável de ambiente CLASSPATH, conforme descrito anteriormente.

Arquivos que não são de classe (dados, imagens, etc.) contidos em seu arquivo JAR

também podem ser recuperados do caminho de classe pelo seu aplicativo usando o método getResource(). Usando esse recurso, seu código não precisa saber se algum recurso está em um arquivo simples ou se é membro de um arquivo JAR. Quer uma determinada classe ou arquivo de dados seja um item em um arquivo JAR ou um arquivo individual no caminho de classe, você sempre pode consultá-lo de maneira padrão e deixar o carregador de classes do Java resolver a localização.

Os itens armazenados em arquivos JAR são compactados com a compactação de arquivo ZIP padrão.2A compactação torna o download de aulas em uma rede muito mais rápido. Uma rápida pesquisa na distribuição Java padrão mostra que um arquivo de classe típico diminui cerca de 40% quando é compactado. Arquivos de texto contendo palavras em inglês, como HTML ou ASCII, geralmente são compactados para um décimo do tamanho original ou menos. (Por outro lado, os arquivos de imagem



normalmente não ficam menores quando compactados, pois os formatos de imagem mais comuns são formatos de compactação.)

O utilitário jar

O utilitário jar fornecido com o JDK é uma ferramenta simples para criar e ler arquivos JAR. Sua interface de usuário não é particularmente amigável. Ele imita o comando de arquivamento de fita Unix, tar. Se você estiver familiarizado com o tar, reconhecerá os seguintes encantamentos, todos compartilhando a forma apresentada emFigura 3-9:

jar -cvf jarCaminho do arquivo [caminho] [...]

Crie jarFile contendo caminho(s).

jar -tvf jarArquivo [caminho] [...]

Liste o conteúdo de jarFile, mostrando opcionalmente apenas o(s) caminho(s).

jar -xvf jarArquivo [caminho] [...]

Extraia o conteúdo de jarFile, extraindo opcionalmente apenas o(s) caminho(s).

Figura 3-9. Elementos importantes do utilitário de linha de comando jar Nestes comandos, as letras de

sinalização c, t e x informam ao jar se ele está criando um arquivo, listando o conteúdo de um arquivo ou extraindo arquivos de um arquivo.

O sinalizador f significa que o próximo argumento é o nome do arquivo JAR no qual operar.

Dica

O sinalizador v opcional informa ao comando jar para ser detalhado ao exibir informações sobre arquivos. No modo detalhado, você obtém informações sobre tamanhos de arquivos, tempos de modificação e taxas de compactação.

Os itens subsequentes na linha de comando (praticamente qualquer coisa além das letras que dizem ao jar o que fazer e o arquivo em que o jar deve operar) são considerados nomes de itens de arquivo. Se você estiver criando um arquivo, os arquivos e diretórios listados serão colocados nele. Se você estiver extraindo, apenas os nomes de arquivos listados serão extraídos do arquivo. (Se você não listar nenhum arquivo, jar extrai tudo do arquivo.)

Por exemplo, digamos que acabamos de completar nosso novo jogo, Space Blaster.

Todos os arquivos associados ao jogo estão em três diretórios. As próprias classes Java estão no diretório spaceblaster/game, spaceblaster/images contém as imagens do jogo e spaceblaster/docs contém dados de jogo associados. Podemos empacotar tudo isso em um arquivo com este comando:

% jar -cvf spaceblaster.jar spaceblaster/

Como solicitamos uma saída detalhada, jar nos diz o que está fazendo: manifesto adicionado

adicionando: spaceblaster/(in = 0) (out= 0)(armazenado 0%)

adicionando: spaceblaster/docs/(in = 0) (out= 0) (armazenado 0%)

adicionando: spaceblaster/docs/help1.html(in = 502) (out= 327)(deflaciona do 34%)

adicionando: spaceblaster/docs/help2.html(in = 562) (out= 360)(deflaciona do 35%)

adicionando: spaceblaster/game/(in = 0) (out= 0) (armazenado 0%)

adicionando: spaceblaster/game/Game.class(in = 362) (out= 270)(deflaciona do 25%)

adicionando: spaceblaster/game/Planetoid.class(in = 606) (out= 418)(defla cionado 31%)

adicionando: spaceblaster/game/SpaceShip.class(in = 1084) (out= 629)(defl acionado 41%)

adicionando: spaceblaster/images/(in = 0) (out= 0) (armazenado 0%)

adicionando: spaceblaster/images/planetoid.png(in = 3434) (out= 3439)(def lacionado 0%)

adicionando: spaceblaster/images/spaceship.png(in = 2760) (out= 2765)(def lacionado 0%)

jarra cria o arquivo spaceblaster.jar e adiciona o diretório spaceblaster, adicionando os diretórios e arquivos dentro

do spaceblaster ao arquivo. No modo detalhado, jar relata a economia obtida ao compactar os arquivos no arquivo.

Podemos descompactar o arquivo com este comando:

% jar -xvf spaceblaster.jar

Descompactar um arquivo JAR é como descompactar um arquivo ZIP. As pastas são criadas onde você emitiu o comando e os arquivos são colocados na hierarquia correta. Também podemos extrair um arquivo ou diretório individual fornecendo mais um argumento de linha de comando:

% jar -xvf spaceblaster.jar spaceblaster/docs/help2.html

Isso extrairá o arquivo help2.html, mas ele será colocado na pasta spaceblaster/docs –

ambos podem ser criados se necessário. É claro que normalmente não é necessário descompactar um arquivo JAR para usar seu conteúdo; As ferramentas Java sabem como extrair arquivos de arquivos automaticamente. Se você quiser apenas ver o que há dentro de um arquivo JAR, você pode listar o conteúdo do nosso JAR com o comando:

% jar -tvf spaceblaster.jar

Aqui está a saída. Ele lista todos os arquivos, seus tamanhos e horários de criação:

0 Ter, 07 de fevereiro 18:33:20 EST 2023 META-INF/

63 Ter, 07 de fevereiro, 18:33:20 EST 2023 META-INF/MANIFEST.MF

0 seg, 06 de fevereiro 19:21:24 EST 2023 spaceblaster/

0 seg, 06 de fevereiro 19:31:30 EST 2023 spaceblaster/docs/

502 Seg, 06 de fevereiro 19:31:30 EST 2023 spaceblaster/docs/help1.html 562 Seg, 06 de fevereiro 19:30:52 EST 2023 spaceblaster/docs/help2.html 0 seg, 06 de fevereiro 19:41:14 EST 2023 spaceblaster/jogo/

362 Seg, 06 de fevereiro 19:40:22 EST 2023 spaceblaster/game/Game.class 606 Seg, 06 de fevereiro 19:40:22 EST 2023 spaceblaster/game/Planetoid.cl ass

1084 Seg, 06 de fevereiro 19:40:22 EST 2023 spaceblaster/game/SpaceShip.c lass

0 seg, 06 de fevereiro 16:30:06 EST 2023 spaceblaster/imagens/

3434 Seg, 06 de fevereiro 16:30:06 EST 2023 spaceblaster/images/planetoid

.png

2760 Seg, 06 de fevereiro 16:27:26 EST 2023 spaceblaster/images/spaceship

.png

Se você deixar de fora o sinalizador detalhado para as ações de extração ou criação, não verá nenhuma saída (a menos que algo dê errado). Deixar de fora o sinalizador detalhado para a ação do índice simplesmente imprime o caminho e o nome de cada arquivo ou diretório sem qualquer informação extra.

Manifestos JAR

Observe que o comando jar adiciona automaticamente um diretório chamado METAINF ao nosso arquivo. O diretório META-INF contém arquivos que descrevem o conteúdo do arquivo JAR. Sempre contém pelo menos um arquivo: MANIFEST.MF. O

arquivo MANIFEST.MF geralmente contém uma “lista de embalagem” nomeando arquivos importantes no arquivo, junto com um conjunto de atributos definíveis pelo usuário para cada entrada.

O manifesto é um arquivo de texto contendo um conjunto de linhas no formato palavra-chave: valor. O manifesto é, por padrão, quase vazio e contém apenas informações de versão do arquivo JAR:

Versão do manifesto: 1.0

Criado por: 1.7.0_07 (Oracle Corporation)

Também é possível assinar arquivos JAR com assinatura digital. Ao fazer isso, informações de resumo (soma de verificação) são adicionadas ao manifesto para cada item arquivado (conforme mostrado a seguir), e o diretório META-INF contém arquivos de assinatura digital para itens no arquivo:

Nome: com/oreilly/Test.class

SHA1-Digest: dF2GZt8G11dXY2p4olzzlc5RjP3=

...

Você pode adicionar suas próprias informações às descrições do manifesto especificando seu próprio

arquivo de manifesto complementar ao criar o arquivo. Este

é um local possível para armazenar outros tipos simples de informações de atributos sobre os arquivos no arquivo, talvez informações de versão ou autoria.

Por exemplo, podemos criar um arquivo com a seguinte palavra-chave: linhas de valor:

Nome: spaceblaster/images/planetoid.gif

Número de revisão: 42,7

Temperamento do Artista: temperamental

Para adicionar essas informações ao manifesto em nosso arquivo, coloque-as em um arquivo chamado myManifest.mf3em seu diretório atual e dê o seguinte comando jar:

% jar -cvmf meuManifesto.mf spaceblaster.jar spaceblaster

Observe que incluímos uma opção adicional na lista compacta de sinalizadores, m, que especifica que jar deve ler informações adicionais do manifesto do arquivo fornecido na linha de comando. Como o jar sabe qual arquivo é qual? Como m é anterior a f, ele espera encontrar as informações do nome do arquivo de manifesto antes do nome do arquivo JAR que criará. Se você acha isso estranho, você está certo; coloque os nomes na ordem errada e jar faça a coisa errada. Felizmente, é fácil corrigir: basta excluir o arquivo incorreto e criar um novo com os nomes na ordem correta.

Se você estiver curioso, um aplicativo pode ler suas próprias informações de manifesto de um arquivo JAR usando a classe java.util.jar.Manifest. Seus detalhes estão além do escopo do que precisamos para este livro, mas sinta-se à vontade para verificar o pacote java.util.jar na documentação. Os aplicativos Java podem fazer bastante com o conteúdo dos arquivos JAR.

Tornando um arquivo JAR executável

Agora, de volta ao nosso novo arquivo de manifesto. Além dos atributos, você pode colocar alguns valores especiais no arquivo de manifesto. Uma delas, Main-Class, permite especificar uma classe contendo o método main() primário para uma aplicação contida no JAR:

Classe principal: spaceblaster.game.Game

capítulo 5tem mais informações sobre nomes de pacotes. Se você adicionar isso ao manifesto do arquivo JAR (usando a opção m descrita anteriormente), poderá executar o aplicativo diretamente do JAR:

% java -jar spaceblaster.jar

Infelizmente, a maioria dos sistemas operacionais abandonou a capacidade de clicar duas vezes em um aplicativo JAR em seus navegadores de arquivos. Atualmente, os aplicativos de desktop profissionais escritos em Java geralmente têm um wrapper executável (como um arquivo .bat no Windows ou um arquivo .sh no Linux ou macOS) para melhor compatibilidade.

Conclusão da ferramenta

Obviamente, existem algumas ferramentas no ecossistema Java – elas acertaram o nome com o

agrupamento inicial de tudo no “Kit” de desenvolvimento Java. Você não usará todas as ferramentas mencionadas acima imediatamente, então não se preocupe se a lista de utilitários parecer um pouco complicada. Vamos nos concentrar no uso do compilador javac e do utilitário interativo jshell à medida que você avança nas águas do Java. Nosso objetivo neste capítulo é garantir que você saiba quais ferramentas existem para que possa voltar para obter detalhes quando precisar delas.

Perguntas de revisão

1. Qual instrução dá acesso aos componentes Swing em seu aplicativo?

2. Qual variável de ambiente determina onde o Java procurará os arquivos de classe ao compilar ou executar?

3. Quais opções você usa para visualizar o conteúdo de um arquivo JAR sem descompactá-lo?

4. Qual entrada é necessária no arquivo MANIFEST.MF para tornar um arquivo JAR executável?

5. Qual ferramenta permite que você experimente o código Java de forma interativa?

Exercícios de código

Seus desafios de programação para este capítulo não requerem nenhuma

programação. Em vez disso, queremos analisar a criação e execução de arquivos JAR.

Este exercício permite praticar a inicialização de um aplicativo Java a partir de um arquivo JAR. Primeiro,

localize o aplicativo de revisão interativa, lj6review.jar, na pasta do questionário onde você instalou os exemplos. Use o sinalizador -jar com o comando java (Java 17 ou superior) para iniciar o aplicativo de revisão:

% teste de cd

% java -jar lj6review.jar

Assim que começar, você poderá testar sua memória e suas novas habilidades respondendo às perguntas de revisão de todos os capítulos deste livro. Não de uma vez, é claro! Mas você pode continuar voltando ao aplicativo de revisão enquanto lê mais. O aplicativo apresenta as mesmas questões encontradas no final de cada capítulo em formato de múltipla escolha. Se você errar uma resposta, incluímos algumas breves explicações que o ajudarão a apontar a direção certa.

O código-fonte deste pequeno aplicativo de revisão está incluído na pasta quiz/src se você quiser dar uma olhada nos bastidores.

Exercícios avançados de código

Para um desafio extra, crie um arquivo JAR executável. Compile HelloJar.java e inclua os arquivos de classe resultantes (deve haver dois) junto com o arquivo manifest.mf em seu arquivo. Nomeie o arquivo JAR como hello.jar. Você precisa fazer uma modificação: você terá que atualizar o arquivo manifest.mf para indicar a classe principal. Neste aplicativo, a classe HelloJar contém o método main() necessário para iniciar. Quando isso for concluído, você poderá executar o seguinte comando em uma janela de terminal ou na guia do terminal em seu IDE:4

```
% java -jar olá.jar
```

Uma saudação gráfica amigável semelhante ao nosso exemplo HelloComponent

de“OláComponente”deve aparecer na sua tela. Não trapaceie! Usamos alguns dos métodos mencionados em“Manifestos JAR”para ler o conteúdo do arquivo de manifesto. Se você simplesmente compilar e executar o aplicativo sem criar um arquivo JAR, sua saudação não será tão congratulatória.

Por fim, veja o código-fonte do programa, se desejar. Inclui alguns novos elementos de Java que abordaremos no próximo capítulo.

1Os arquivos JAR são, em sua maioria, arquivos ZIP convencionais com metadados extras. Como tal, Java também suporta arquivos no formato ZIP convencional, mas raramente é usado.

2Você pode até usar utilitários ZIP padrão para inspecionar ou descompactar arquivos JAR.

3O nome real depende inteiramente de você, mas a extensão do arquivo .mf é comum.

4Se você tiver algum problema para construir este arquivo JAR, as soluções de exercícios emApêndice Bconté m etapas mais detalhadas para ajudar.

Capítulo 4. A Linguagem Java

Como humanos, aprendemos as sutilezas da linguagem falada por meio de tentativa e erro. Aprendemos onde colocar o sujeito em relação ao verbo e como lidar com coisas como tempos e plurais. Certamente aprendemos

regras linguísticas avançadas na escola, mas mesmo os alunos mais jovens podem fazer perguntas inteligíveis aos professores. As linguagens de computador têm características semelhantes: existem

“classes gramaticais” que funcionam como blocos de construção combináveis. Existem maneiras de declarar fatos e fazer perguntas. Neste capítulo, examinamos essas unidades de programação fundamentais em Java. Tentativa e erro continuam sendo um ótimo professor, então também veremos como brincar com essas novas unidades e praticar suas habilidades.

Como a sintaxe do Java é derivada de C, fazemos algumas comparações com recursos dessa linguagem, mas nenhum conhecimento prévio de C é necessário.capítulo 5se

baseia neste capítulo falando sobre o lado orientado a objetos do Java e completando a discussão sobre a linguagem central.Capítulo 7discute genéricos e registros, recursos que aprimoram o modo como os tipos funcionam na linguagem Java, permitindo escrever determinados tipos de classes com mais flexibilidade e segurança.

Depois disso, mergulhamos nas APIs Java e vemos o que podemos fazer com a linguagem. O restante deste livro está repleto de breves exemplos que fazem coisas úteis em diversas áreas. Se você tiver alguma dúvida após estes capítulos introdutórios, esperamos que ela seja respondida à medida que você analisa o código.

Sempre há mais para aprender, é claro! Tentaremos apontar outros recursos ao longo do caminho que podem

beneficiar as pessoas que desejam continuar sua jornada Java além dos tópicos que abordamos.

Para leitores que estão começando sua jornada de programação, a web provavelmente será uma companheira constante. Muitos, muitos sites, artigos da Wikipédia, postagens de blogs e, bem, a totalidade deEstouro de pilhapode ajudá-lo a se aprofundar em tópicos específicos ou a responder pequenas perguntas que possam surgir. Por exemplo, embora este livro cubra a linguagem Java e como começar a escrever programas úteis com Java e suas ferramentas, não abordamos tópicos básicos de programação, comoalgoritmoscom muito detalhe. Esses fundamentos de programação aparecerão naturalmente em nossas discussões e exemplos de código, mas você pode aproveitar algumas tangentes de hiperlink para ajudar a consolidar certas ideias ou preencher as lacunas que necessariamente devemos deixar.

Como mencionamos antes, muitos termos neste capítulo serão desconhecidos. Não se preocupe se você ocasionalmente ficar um pouco confuso. A grande amplitude do Java significa que temos que deixar de fora explicações ou detalhes básicos de vez em quando. À medida que você avança, esperamos que você tenha a oportunidade de revisitar alguns desses primeiros capítulos. Novas informações podem funcionar um pouco como um quebra-cabeça. É mais fácil encaixar uma nova peça se você já tiver outras peças relacionadas conectadas. Quando você passar algum tempo escrevendo código e este livro se tornar mais uma referência para você e menos um guia, você descobrirá que os tópicos desses primeiros capítulos farão mais sentido.

Codificação de texto

Java é uma linguagem para a internet. Como os usuários individuais falam e escrevem em muitas linguagens humanas diferentes, Java também deve ser capaz de lidar com um grande número de linguagens. Ele lida com a internacionalização através do conjunto de caracteres Unicode, um padrão mundial que suporta scripts da maioria dos idiomas.1A versão mais recente do Java baseia seus dados de caracteres e strings no padrão Unicode 14.0, que usa pelo menos dois bytes para representar cada símbolo internamente. Como você deve se lembrar de"O Passado: Java 1.0 – Java 20", a Oracle

se esforça para acompanhar os novos lançamentos do padrão Unicode. Sua versão do Java pode incluir uma versão mais recente do Unicode.

O código-fonte Java pode ser escrito usando Unicode e armazenado em qualquer número de codificações de caracteres. Isso torna o Java uma linguagem bastante amigável para incluir conteúdo que não seja em inglês. Os programadores podem usar o rico conjunto de caracteres do Unicode não apenas para exibir informações ao usuário, mas também em seus próprios nomes de classes, métodos e variáveis.

O tipo char Java e a classe String suportam nativamente valores Unicode.

Internamente, o texto é armazenado usando matrizes de caracteres ou bytes; entretanto, a linguagem Java e as APIs tornam isso transparente para você e geralmente você não terá que pensar nisso. Unicode também é muito compatível com ASCII (ASCII é a codificação de caracteres mais comum em inglês). Os primeiros 256

caracteres são definidos para serem idênticos aos primeiros 256 caracteres do conjunto de caracteres ISO 8859-1 (Latin-1), portanto, o Unicode é efetivamente compatível com versões anteriores dos conjuntos de caracteres ingleses mais comuns.

Além disso, uma das codificações de arquivo mais comuns para Unicode, chamada UTF-8, preserva os valores ASCII em seu formato de byte único. Essa codificação é usada por padrão em arquivos de classe Java compilados, portanto o armazenamento permanece compacto para texto em inglês.

A maioria das plataformas não pode exibir todos os caracteres Unicode definidos atualmente. Como solução alternativa, os programas Java podem ser escritos com sequências de escape Unicode especiais. Um caractere Unicode pode ser representado com esta sequência de escape:

\uxxxx

xxxx é uma sequência de um a quatro dígitos hexadecimais. A sequência de escape indica um caractere Unicode codificado em ASCII. Esta também é a forma que Java usa para gerar (imprimir) caracteres Unicode em um ambiente que de outra forma não os suporta. Java vem com classes para ler e escrever fluxos de caracteres Unicode em codificações específicas, incluindo UTF-8.

Tal como acontece com muitos padrões de longa duração no mundo da tecnologia, o Unicode foi originalmente projetado com tanto espaço extra que nenhuma codificação de caracteres concebível poderia exigir mais de 64K caracteres. Suspirar.

Naturalmente ultrapassamos esse limite e algumas codificações UTF-32 estão em circulação popular. Mais notavelmente, os caracteres emoji espalhados pelos aplicativos de mensagens são codificados além da faixa padrão de caracteres Unicode.

(Por exemplo, o emoji canônico do smiley tem o valor Unicode 1F600.) Java suporta sequências de escape UTF-16 multibyte para tais caracteres. Nem toda plataforma que suporta Java suportará saída de emoji, mas você pode iniciar o jshell para descobrir se seu ambiente pode mostrar caracteres emoji (consulteFigura 4-1).

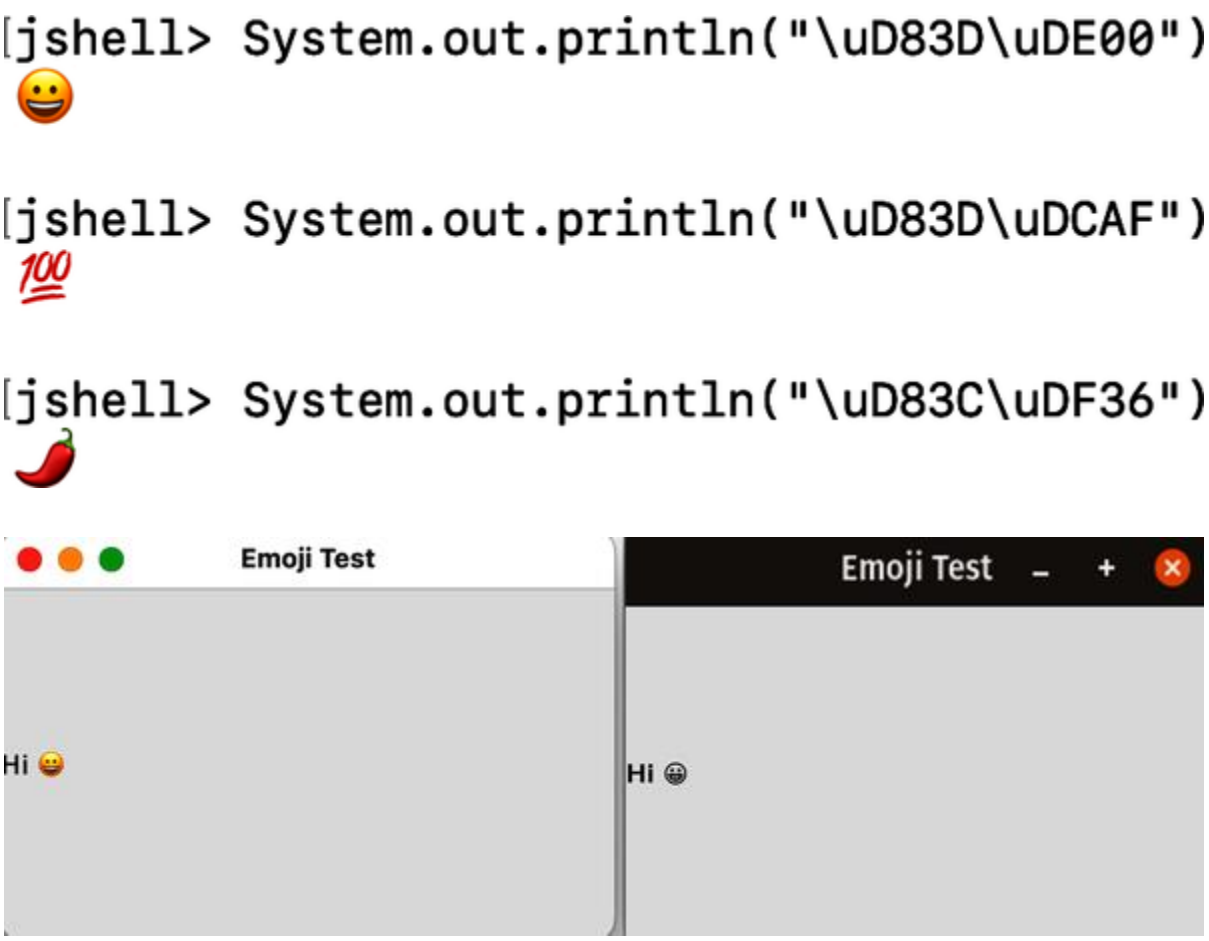


Figura 4-1. Imprimindo emojis no aplicativo macOS Terminal

Porém, tenha cuidado ao usar esses caracteres. Tivemos que usar uma captura de tela para garantir que você pudesse ver as pequenas fofuras do jshell rodando em um Mac.

Você pode usar jshell para testar seu próprio sistema. Você pode criar um aplicativo gráfico mínimo semelhante à nossa classe HelloJava em“OláJava”. Crie um JFrame, adicione um JLabel e torne o quadro visível:

```
jshell> importar javax.swing.*

jshell> JFrame f = new JFrame("Teste de Emoji")

f ==> javax.swing.JFrame[frame0 ...=true]

jshell> f.add(new JLabel("Olá \uD83D\uDE00"))

$ 12 ==> javax.swing.JLabel[ ...=CENTRO]

jshell> f.setSize(300.200)

jshell> f.setVisible(verdadeiro)
```

Esperamos que você veja o smiley, mas isso dependerá do seu sistema.Figura 4-2mostra os resultados que obtivemos ao fazer esse teste exato no macOS e no Linux.

Figura 4-2. Testando a apresentação de emojis em vários sistemas

Não é que você não possa usar ou oferecer suporte a emojis em seus aplicativos, você apenas precisa estar ciente das diferenças nos recursos de saída. Certifique-se de que seus usuários tenham uma boa experiência onde quer que estejam executando seu código.

Aviso

Ao importar os componentes gráficos do pacote Swing, tome cuidado para usar o prefixo javax correto em vez do prefixo java padrão. Mais sobre todas as coisas Swing

inCapítulo 12.

Comentários

Agora que sabemos como o texto dos nossos programas é armazenado, podemos nos concentrar no que armazenar! Os programadores geralmente incluem comentários em seu código para ajudar a explicar partes complexas da lógica ou para fornecer um guia de leitura do código para outros programadores. (Muitas vezes o "outro programador" é você mesmo vários meses ou anos depois.) O texto de um comentário é completamente ignorado pelo compilador. Os comentários não têm impacto no desempenho ou na funcionalidade do seu aplicativo. Como tal, somos grandes fãs de escrever bons comentários. Java suporta comentários de bloco no estilo C que podem abranger várias linhas delimitadas por /* e */ e comentários de linha no estilo C++

indicados por //:

/* Isto é um

multilinha

Comente. */

// Este é um comentário de linha única

// e então // é isso

Os comentários em bloco têm uma sequência inicial e final e podem cobrir grandes intervalos de texto. No entanto, eles não podem ser “aninhados”, o que significa que você não pode colocar um comentário de bloco dentro de outro comentário de bloco sem entrar em conflito com o compilador. Os comentários de linha única possuem apenas uma sequência inicial e são delimitados pelo final de uma linha; // indicadores extras dentro de uma única linha não têm efeito. Os comentários de linha são úteis para comentários curtos dentro dos métodos; eles não entram em conflito com comentários de bloqueio. Você ainda pode agrupar pedaços maiores de código nos quais os comentários de linha única aparecem com um comentário em bloco. Isso geralmente é chamado de comentar uma seção de código — um truque comum para depurar aplicativos grandes. Como o compilador ignora todos os comentários, você pode colocar comentários em linhas ou em torno de blocos de código para ver como um programa se comporta quando esse código é removido.2

Comentários Javadoc

Um comentário de bloco especial começando com /** indica um comentário de documento. Um comentário de documento foi projetado para ser extraído por geradores automatizados de documentação, como o próprio programa javadoc do JDK

ou as dicas de ferramentas sensíveis ao contexto em muitos IDEs. Um comentário de documento é finalizado pelo próximo */, assim como um comentário de bloco normal.

Dentro do comentário do documento, as linhas que começam com @ são

interpretadas como instruções especiais para o gerador de documentação, fornecendo informações sobre o código-fonte. Por convenção, cada linha de um comentário de documento começa com um *, conforme mostrado no exemplo a seguir, mas isso é opcional. Qualquer espaçamento inicial e o * em cada linha são ignorados:

```
/**

*Eu acho que essa aula é possivelmente a coisa mais incrível que você irá

*já viu. Deixe-me contar sobre minha visão pessoal e

* motivação em criá-lo.

* <p>

* Tudo começou quando eu era criança, crescendo no

* ruas de Idaho. As batatas estavam na moda e a vida era boa...

*

* @ver Descascador de Batata

* @ver Masher de Batata

* @autor John 'Spuds' Smith

* @versão 1.00, 19 de novembro de 2022

*/

classe Batata {...}
```

A ferramenta de linha de comando javadoc cria documentação HTML para classes lendo o código-fonte e extraindo os comentários incorporados e as tags @. Neste exemplo, as tags criam informações de autor e versão na documentação da classe. As tags @see produzem links de hipertexto para a documentação da classe relacionada.

O compilador também analisa os comentários do documento; em particular, está interessado na tag @deprecated, o que significa que o método foi declarado obsoleto e deve ser evitado em novos programas. A classe compilada inclui informações sobre quaisquer métodos obsoletos para que o compilador possa avisá-lo sempre que você usar um recurso obsoleto em seu código (mesmo que a fonte não esteja disponível).

Os comentários do documento podem aparecer acima das definições de classes, métodos e variáveis, mas algumas tags podem não se aplicar a todas elas. Por exemplo, a tag @exception só pode ser aplicada a métodos.Tabela 4-1resume as tags usadas nos comentários do documento.

Tabela 4-1. Tags de comentários de documentos

Marcação

Descrição

Aplica-se a

@ver

Nome da classe associada

Classe, método ou

variável

@código

Conteúdo do código-fonte

Classe, método ou

variável

@link

URL associado

Classe, método ou

variável

@autor

Nome do autor

Aula

@versão

Cadeia de versão

Aula

@param

Nome e descrição do parâmetro

Método

@retornar

Descrição do valor de retorno

Método

Marcação

Descrição

Aplica-se a

@exceção

Nome e descrição da exceção

Método

@descontinuada Declara um item como obsoleto

Classe, método ou

variável

@desde

Versão da API do Notes quando o item foi Variável

adicionado

As tags Javadoc nos comentários do documento representam metadados sobre o código-fonte; isto é, adicionam informações descritivas sobre a estrutura ou conteúdo do código que não faz, estritamente falando, parte da aplicação. Algumas ferramentas adicionais estendem o conceito de tags estilo Javadoc para incluir outros tipos de metadados sobre programas Java que são transportados com o código compilado e podem ser usados mais facilmente pelo aplicativo para afetar sua compilação ou comportamento em tempo de execução. O recurso de anotações Java fornece uma maneira mais formal e extensível de incluir metadados em classes,

métodos e variáveis Java. Esses metadados também estão disponíveis em tempo de execução.

Anotações

O prefixo @ desempenha outra função em Java que pode ser semelhante a tags. Java suporta a noção de anotações como meio de marcar determinado conteúdo para tratamento especial. Você aplica anotações ao código fora dos comentários. A anotação pode fornecer informações úteis para o compilador ou para o seu IDE. Por exemplo, a anotação @SuppressWarnings faz com que o compilador (e muitas vezes também o seu IDE) oculte avisos sobre possíveis problemas, como código inacessível.

À medida que você cria aulas mais interessantes em“Design de Classe Avançado”, você poderá ver seu IDE adicionar anotações @Overrides ao seu código. Esta anotação informa ao compilador para realizar algumas verificações extras; essas verificações têm como objetivo ajudá-lo a escrever código válido e detectar erros antes que você (ou seus usuários) execute seu programa.

Você pode até criar anotações personalizadas para trabalhar com outras ferramentas ou estruturas. Embora uma discussão mais profunda sobre anotações esteja além do escopo deste livro, gostaríamos que você as conhecesse, pois tags como @Overrides aparecerão tanto em nosso código quanto em exemplos ou postagens de blog que você possa encontrar on-line.

Variáveis e Constantes

Embora adicionar comentários ao seu código seja fundamental para produzir arquivos legíveis e de fácil manutenção, em algum momento você terá que começar

a escrever algum conteúdo compilável. Programar é a arte de manipular esse conteúdo. Em quase todas as linguagens, essas informações são armazenadas em variáveis e constantes para facilitar o uso pelo programador. Java tem ambos. As variáveis

armazenam informações que você planeja alterar e reutilizar ao longo do tempo (ou informações que você não conhece antecipadamente, como o endereço de e-mail de um usuário). As constantes armazenam informações que são, bem, constantes. Vimos exemplos de ambos os elementos mesmo em nossos pequenos programas iniciais.

Lembre-se de nosso rótulo gráfico simples de“OláJava”:

```
importar javax.swing.*;

classe pública HelloJava {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Olá, Java!");

Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

frame.add(rótulo);

frame.setSize(300, 300);

frame.setVisible (verdadeiro);

}

}
```

Neste trecho, frame é uma variável. Carregamos na linha 5 com uma nova instância da classe JFrame. Então

podemos reutilizar a mesma instância na linha 7 para adicionar nosso rótulo. Reutilizamos a variável novamente para definir o tamanho do nosso quadro na linha 8 e torná-lo visível na linha 9. Toda essa reutilização é exatamente onde as variáveis brilham.

A linha 6 contém uma constante: JLabel.CENTER. As constantes contêm um valor específico que nunca muda durante o programa. Informações que não mudam podem parecer algo estranho de se armazenar – por que não usar sempre as informações em si? As constantes podem ser mais simples de usar do que seus dados; Math.PI é provavelmente mais fácil de lembrar do que o valor 3,141592653589793 que ele representa. E como você pode selecionar o nome das constantes em seu próprio código, outro benefício é que você pode descrever as informações de uma forma útil.

JLabel.CENTER ainda pode parecer um pouco opaco, mas a palavra CENTER pelo menos dá uma dica sobre o que está acontecendo.

O uso de constantes nomeadas também permite alterações mais simples no futuro. Se você codificar algo como o número máximo de algum recurso que você usa, alterar esse limite será muito mais fácil se tudo o que você precisar fazer for alterar o valor inicial dado à constante. Se você usar um número literal como 5, toda vez que seu código precisar verificar esse máximo, você terá que procurar em todos os seus arquivos Java para rastrear cada ocorrência de 5 e alterá-la também - se esse 5

específico estiver em fato referente ao limite de recursos. Esse tipo de pesquisa e substituição manual está sujeito a erros, além de ser tedioso.

Veremos mais detalhes sobre os tipos e valores iniciais de variáveis e constantes na próxima seção. Como sempre, fique à vontade para usar o jshell para explorar e descobrir alguns desses detalhes por conta própria! Devido às limitações do interpretador, você não pode declarar suas próprias constantes de nível superior no

jshell. Você ainda pode usar constantes definidas para classes como JLabel.CENTER ou defini-las em suas próprias classes.

Tente digitar as seguintes instruções em jshell para calcular e armazenar a área de um círculo em uma variável usando Math.PI. Este exercício também prova que reatribuir constantes não funcionará. (E, novamente, temos que introduzir alguns conceitos novos, como atribuição – colocar um valor em uma variável – e o operador de multiplicação *. Se esses comandos ainda parecerem estranhos, continue lendo.

Examinaremos todos os novos elementos em mais detalhes. detalhes ao longo do restante deste capítulo.)

jshell> raio duplo = 42,0;

DADOS DE ODINRIGHT

Sobre a obra:

A presente obra é disponibilizada pela equipe ***eLivros*** e seus diversos parceiros, com o objetivo de oferecer conteúdo para uso parcial em pesquisas e estudos acadêmicos, bem como o simples teste da qualidade da obra, com o fim exclusivo de compra futura.

É expressamente proibida e totalmente repudíavel a venda, aluguel, ou quaisquer uso comercial do presente conteúdo.

Sobre nós:

O ***eLivros*** e seus parceiros disponibilizam conteúdo de dominio publico e propriedade intelectual de forma totalmente gratuita, por acreditar que o conhecimento e a educação devem ser acessíveis e livres a toda e qualquer pessoa. Você pode encontrar mais obras em nosso site: ***eLivros***.

Como posso contribuir?

Você pode ajudar contribuindo de várias maneiras, enviando livros para gente postar ***Envie um livro*** ;)

Ou ainda podendo ajudar financeiramente a pagar custo de servidores e obras que compramos para postar, ***faça uma doação aqui*** :)

"Quando o mundo estiver unido na busca do conhecimento, e não mais lutando por dinheiro e

poder, então nossa sociedade poderá enfim evoluir a um novo nível."

```
raio ==> 42,0

jshell> Math.PI

$ 2 ==> 3,141592653589793

jshell> Math.PI = 3;

| Erro:

| não é possível atribuir um valor à variável final PI

| Matemática.PI = 3;

| ^-----^

jshell> área dupla = Math.PI * raio * raio;

área ==> 5541.769440932396

jshell> raio = 6;

raio ==> 6,0

jshell> área = Math.PI * raio * raio;

área ==> 113.09733552923255

jshell> área

área ==> 113.0973352923255
```

Observe o erro do compilador quando tentamos definir Math.PI como 3. Você pode alterar o raio e até mesmo a área depois de declará-los e inicializá-los. Mas as variáveis contêm apenas um valor por vez, portanto o

cálculo mais recente é a única coisa que permanece na área da variável.

Tipos

O sistema de tipos de uma linguagem de programação descreve como seus elementos de dados (as variáveis e constantes que acabamos de mencionar) estão associados ao armazenamento na memória e como estão relacionados entre si. Em uma linguagem de tipo estaticamente, como C ou C++, o tipo de um elemento de dados é um atributo simples e imutável que geralmente corresponde diretamente a algum fenômeno de

hardware subjacente, como um registro ou um valor de ponteiro. Em uma linguagem de tipo dinâmico, como Smalltalk ou Lisp, variáveis podem receber elementos arbitrários e podem efetivamente alterar seu tipo ao longo de sua vida útil. Uma quantidade considerável de sobrecarga é necessária para validar o que acontece nessas linguagens em tempo de execução. Linguagens de script, como Perl, são fáceis de usar, fornecendo sistemas de tipos drasticamente simplificados, nos quais apenas determinados elementos de dados podem ser armazenados em variáveis e os valores são unificados em uma representação comum, como strings.

Java combina muitos dos melhores recursos de linguagens de tipagem estática e dinâmica. Como em uma linguagem de tipo estaticamente, cada variável e elemento de programação em Java possui um tipo que é conhecido em tempo de compilação, portanto, o sistema de tempo de execução normalmente não precisa verificar a validade das atribuições entre os tipos enquanto o código está em execução. Ao contrário do C ou C++

tradicional, Java também mantém informações de tempo de execução sobre objetos e usa isso para permitir um comportamento verdadeiramente dinâmico. O código Java pode carregar novos tipos em tempo de execução e usá-los de maneira totalmente orientada a objetos, permitindo conversão (conversão entre tipos) e polimorfismo completo (combinação de recursos de vários tipos). O código Java também pode “refletir” ou examinar seus próprios tipos em tempo de execução, permitindo tipos avançados de comportamento do aplicativo, como interpretadores que podem interagir dinamicamente com programas compilados.

Os tipos de dados Java se enquadram em duas categorias. Os tipos primitivos representam valores simples que possuem funcionalidade integrada na linguagem; eles representam números, valores booleanos (verdadeiros ou falsos) e caracteres. Os tipos de referência (ou tipos de classe) incluem objetos e matrizes; eles são chamados de tipos de referência porque “se referem” a um grande tipo de dados que é passado

“por referência”, como explicaremos em breve. Genéricos são tipos de referência que refinam um tipo existente enquanto ainda fornecem segurança de tipo em tempo de compilação. Por exemplo, Java possui uma classe List que pode armazenar uma série de itens. Usando genéricos, você pode criar um List<String> que é uma lista que só pode conter Strings. Ou poderíamos criar uma lista de objetos JLabel com List<JLabel>. Veremos muito mais gen[éricos emCapítulo 7.]

Tipos Primitivos

Números, caracteres e valores booleanos são elementos fundamentais em Java. Ao contrário de algumas outras linguagens orientadas a objetos (talvez mais puras), elas não são objetos. Para aquelas situações em que é desejável tratar um valor primitivo como um objeto, Java fornece classes “wrapper”. (Mais sobre isso mais tarde.) A principal vantagem de tratar valores primitivos como especiais é que o compilador Java e o tempo de execução podem otimizar mais prontamente sua implementação.

Valores primitivos e cálculos ainda podem ser mapeados para hardware, como sempre foram em linguagens de nível inferior.

Um importante recurso de portabilidade do Java é que os tipos primitivos são definidos com precisão. Por exemplo, você nunca precisa se preocupar com o tamanho de um int em uma plataforma específica; é sempre um número assinado de 32 bits. O

“tamanho” de um tipo numérico determina o tamanho (ou precisão) do valor que você pode armazenar. Por exemplo, o tipo byte é um valor assinado de 8 bits para armazenar números pequenos, de -128 a 127.3O tipo int mencionado acima pode lidar com a maioria das necessidades numéricas, armazenando valores entre

(aproximadamente) +/- dois bilhões.Tabela 4-2resume os tipos primitivos de Java e suas capacidades.

Tabela 4-2. Tipos de dados primitivos Java

Alcance ou precisão

Tipo

Definição

aproximado

boleano

Valor lógico

verdadeiroou falso

Caracteres Caractere Unicode de 16 bits

64 mil caracteres

byte

Inteiro assinado de 8 bits

-128 a 127

curto

Inteiro assinado de 16 bits

-32.768 a 32.767

interno

Inteiro assinado de 32 bits

-2.1e9 a 2.1e9

longo

Inteiro assinado de 64 bits

-9.2e18 a 9.2e18

flutuador

32 bits, IEEE 754, valor de ponto

6-7 casas decimais

flutuante

significativas

dobro

64 bits, IEEE 754

15 casas decimais

significativas

Observação

Aqueles de vocês com experiência em C podem notar que os tipos primitivos parecem uma idealização dos tipos escalares C em uma máquina de 32 bits, e você está absolutamente certo. É assim que eles deveriam parecer. Os designers de Java fizeram algumas mudanças, como suporte a caracteres de 16 bits para Unicode e eliminação de ponteiros ad hoc. Mas no geral, a sintaxe e a semântica dos tipos primitivos Java derivam de C.

Mas por que ter tamanhos? Novamente, isso remete à eficiência e otimização. O

número de gols em uma partida de futebol raramente ultrapassa um dígito – eles caberiam em uma variável de byte. O número de torcedores assistindo aquela partida, porém, precisaria de algo maior. O valor total gasto por todos os torcedores em todas as partidas de futebol em todos os países da Copa do Mundo precisaria de algo ainda maior. Ao escolher o tamanho certo, você dá ao compilador a melhor chance de otimizar seu código,

fazendo com que seu aplicativo seja executado mais rapidamente, consuma menos recursos do sistema ou ambos.

Algumas aplicações científicas ou criptográficas exigem que você armazene e manipule números muito grandes (ou muito pequenos) e valorize a precisão em detrimento do desempenho. Se precisar de números maiores do que os tipos primitivos oferecem, você pode verificar as classes BigInteger e BigDecimal no pacote java.math. Essas classes oferecem tamanho ou precisão quase infinito. (Se você quiser ver esses grandes números em ação, usamos BigInteger para calcular valores fatoriais

em“Criando um redutor personalizado”.)

Precisão de ponto flutuante

As operações de ponto flutuante em Java seguem o IEEE 754especificação internacional, o que significa que o resultado dos cálculos de ponto flutuante é normalmente o mesmo em diferentes plataformas Java. No entanto, Java permite precisão estendida em plataformas que o suportam. Isso pode introduzir diferenças misteriosas e de valor extremamente pequeno nos resultados de operações de alta precisão. A maioria dos aplicativos nunca perceberia isso, mas se você quiser garantir que seu aplicativo produza exatamente os mesmos resultados em plataformas diferentes, você pode usar a palavra-chave especial strictfp como um modificador de classe na classe que contém a manipulação de ponto flutuante (cobrimos as classes

emcapítulo 5). O compilador então proíbe essas otimizações específicas da plataforma.

Declaração e inicialização de variáveis

Você declara variáveis dentro de métodos e classes com um nome de tipo, seguido por um ou mais nomes de variáveis separados por vírgula. Por exemplo:

int foo;

duplo d1, d2;

booleano isFun;

Opcionalmente, você pode inicializar uma variável com uma expressão do tipo apropriado ao declará-la:

int foo = 42;

duplo d1 = 3,14, d2 = 2 * 3,14;

booleano isFun = verdadeiro;

Variáveis declaradas como membros de uma classe serão definidas com valores padrão se não forem inicializadas (consultecapítulo 5). Nesse caso, os tipos numéricos assumem como padrão o sabor apropriado de zero, os caracteres são definidos como o caractere nulo (\0) e as variáveis booleanas têm o valor falso. (Os tipos de referência também recebem um valor padrão, nulo, mas falaremos mais sobre isso em breve“Tipos de referência”.)

Variáveis locais, que são declaradas dentro de um método e permanecem vivas apenas durante uma chamada de método, por outro lado, devem ser inicializadas explicitamente antes de poderem ser usadas. Como veremos, o compilador impõe esta regra, portanto não há perigo de esquecimento.

Literais inteiros

Literais inteiros podem ser especificados em binário (base 2), octal (base 8), decimal (base 10) ou hexadecimal (base 16). Bases binárias, octais e hexadecimais são usadas principalmente ao lidar com arquivos de baixo nível ou dados de rede. Eles representam agrupamentos úteis de bits individuais: 1, 3 e 4 bits, respectivamente. Os valores decimais não possuem esse mapeamento, mas são muito mais amigáveis para a maioria das informações numéricas. Um número inteiro decimal é especificado por uma sequência de dígitos começando com um dos caracteres de 1 a 9:

```
int i = 1230;
```

Um número binário é denotado pelos caracteres iniciais 0b ou 0B (zero “b”), seguidos por uma combinação de zeros e uns:

```
int i = 0b01001011; // i = 75 decimais
```

Os números octais são diferenciados dos números decimais por um simples zero à esquerda:

```
int i = 01230; // i = 664 decimal
```

Um número hexadecimal é denotado pelos caracteres iniciais 0x ou 0X (zero “x”), seguidos por uma combinação de dígitos e os caracteres a – f ou A – F, que representam os valores decimais 10–15:

```
int i = 0xFFFF; //i = 65535 decimal
```

Literais inteiros são do tipo int, a menos que sejam sufixados com um L, denotando que devem ser produzidos como um valor longo:

longo l = 13L;

longo eu = 13; // equivalente: 13 é convertido do tipo int

longo eu = 40123456789L;

longo eu = 40123456789; //erro: muito grande para um int sem conversão (A letra minúscula l também funciona, mas deve ser evitada porque muitas vezes se parece com o número 1.)

Quando um tipo numérico é usado em uma atribuição ou expressão envolvendo um tipo “maior” com um intervalo maior, ele pode ser promovido para o tipo maior. Na segunda linha do exemplo anterior, o número 13 tem o tipo padrão int, mas é promovido para o tipo long para atribuição à variável long.

Certas outras operações numéricas e de comparação também causam esse tipo de promoção aritmética, assim como as expressões matemáticas que envolvem mais de um tipo. Por exemplo, ao multiplicar um valor de byte por um valor int, o compilador primeiro promove o byte para um int:

byte b = 42;

int i = 43;

resultado interno = b * i; // b é promovido para int antes da multiplicaç ão

Você nunca pode seguir o outro caminho e atribuir um valor numérico a um tipo com um intervalo menor sem uma conversão explícita, uma sintaxe especial que você pode usar para informar ao compilador exatamente qual tipo você precisa:

int i = 13;

byte b = eu; // Erro em tempo de compilação, conversão explícita necessár ia

byte b = (byte) i; // OK

A conversão na terceira linha é a frase (byte) antes de nossa variável i. As conversões de tipos de ponto flutuante para inteiros sempre exigem uma conversão explícita devido à potencial perda de precisão.

Por último e talvez menos importante, você pode adicionar um pouco de formatação aos seus literais numéricos utilizando o caractere “_” (sublinhado) entre os dígitos. Se você tiver sequências de dígitos particularmente grandes, poderá separá-las como nos exemplos a seguir:

int RICHARD_NIXONS_SSN = 567_68_0515;

int for_no_reason = 1___2___3;

int JAVA_ID = 0xCAFE_BABE;

grandeTotal longo = 40_123_456_789L;

Os sublinhados só podem aparecer entre dígitos, não no início ou no final de um número ou próximo ao significante inteiro longo L. Experimente alguns números grandes em jshell. Observe que se você tentar armazenar um valor longo sem o significante L, receberá um erro. Você pode ver como realmente é a formatação apenas para sua conveniência. Não é armazenado; apenas o valor real é mantido na sua variável ou constante:

```
jshell> longo m = 41234567890;

| Erro:

| número inteiro muito grande

| longo m = 41234567890;

| ^

jshell> longo m = 40123456789L;

m ==> 40123456789

jshell> longo grandTotal = 40_123_456_789L;

grandeTotal ==> 40123456789
```

Experimente alguns outros exemplos. Pode ser útil ter uma ideia do que você considera legível. Também pode ajudá-lo a aprender os tipos de promoções e castings disponíveis ou necessários. Nada como um feedback imediato para esclarecer essas sutilezas!

Literais de ponto flutuante

Os valores de ponto flutuante podem ser especificados em notação decimal ou científica. Literais de ponto flutuante são do tipo double, a menos que sejam sufixados com f ou F, denotando que são um valor flutuante de menor precisão. E assim como acontece com literais inteiros, você pode usar o caractere sublinhado para formatar números de ponto flutuante – mas, novamente, apenas entre dígitos. Você não pode colocá-los no início, no final, próximo à vírgula decimal ou próximo ao significante F

do número:

duplo d = 8,31;

duplo e = 3,00e+8;

flutuar f = 8,31F;

flutuar g = 3,00e+8F;

flutuar pi = 3,1415_9265F;

Literais de caracteres

Um valor de caractere literal pode ser especificado como um caractere entre aspas simples ou uma sequência ASCII ou Unicode de escape, também entre aspas simples: char a = 'a';

char nova linha = '\n';

char sorridente = '\u263a';

Na maioria das vezes você lidará com caracteres coletados em uma String, mas ainda há lugares onde caracteres individuais são úteis. Por exemplo, se você manipula a entrada do teclado em seu aplicativo, pode ser necessário processar pressionamentos de teclas individuais, um caractere por vez.

Tipos de referência

Em uma linguagem orientada a objetos como Java, você cria tipos de dados novos e complexos a partir de primitivos simples criando uma classe. Cada classe serve então como um novo tipo na linguagem. Por exemplo, se criarmos uma nova classe chamada Car em Java, também estaremos criando implicitamente um novo tipo chamado Car. O

tipo de item determina como ele é usado e onde pode ser atribuído. Tal como acontece com os primitivos, um item do tipo Car pode, em geral, ser atribuído a uma variável do tipo Car ou passado como argumento para um método que aceita um valor Car.

Um tipo não é apenas um simples atributo. As classes podem ter relacionamentos com outras classes, assim como os tipos que elas representam. Todas as classes em Java existem em uma hierarquia pai-filho, onde uma classe ou subclasse filha é um tipo especializado de sua classe pai. Os tipos correspondentes possuem o mesmo relacionamento, onde o tipo da classe filha é considerado um subtipo da classe pai.

Como as classes filhas herdam toda a funcionalidade de suas classes pai, um objeto do tipo filho é, em certo sentido, equivalente ou uma extensão do tipo pai. Um objeto do tipo filho pode ser usado no lugar de um objeto do tipo pai.

Por exemplo, se você criar uma nova classe, Dog, que estenda Animal, o novo tipo, Dog, será considerado um subtipo de Animal. Objetos do tipo Dog podem então ser usados em qualquer lugar onde um objeto do tipo Animal possa ser usado; Diz-se que um objeto do tipo Dog pode ser atribuído a uma variável do tipo Animal. Isso é chamado de polimorfismo de subtipo e é um dos principais recursos de uma linguagem orientada a objetos. Veremos mais de perto as classes e os objet[os emcapítulo 5.]

Tipos primitivos em Java são usados e passados “por valor”. Isso significa que quando um valor primitivo como um int é atribuído a uma variável ou passado como argumento para um método, seu valor é copiado. Os

tipos de referência (tipos de classe), por outro lado, são sempre acessados “por referência”. Uma referência é um identificador ou nome de um objeto. O que uma variável de um tipo de referência contém é um “ponteiro” para um objeto de seu tipo (ou de um subtipo, conforme descrito anteriormente). Quando você atribui a referência a uma variável ou a passa para um método, apenas a referência é copiada, não o objeto para o qual ela está apontando. Uma referência é como um ponteiro em C ou C++, exceto que seu tipo é estritamente aplicado. O valor de referência em si não pode ser criado ou alterado explicitamente. Você deve atribuir um objeto apropriado para fornecer um valor de referência a uma variável de tipo de referência.

Vejamos um exemplo. Declaramos uma variável do tipo Car, chamada myCar, e atribuímos a ela um objeto apropriado:4

```
Carro meuCarro = novo Carro();

Carro outroCarro = meuCarro;
```

meu carroé uma variável do tipo referência que contém uma referência ao objeto Car recém-construído. (Por enquanto, não se preocupe com os detalhes da criação de um objeto; novamente, abordaremos isso emcapítulo 5.) Declaramos uma segunda variável do tipo Car, anotherCar, e a atribuímos ao mesmo objeto. Existem agora duas referências idênticas: myCar e anotherCar, mas apenas uma instância real do objeto Car. Se mudarmos as coisas no estado do próprio objeto Car, veremos o mesmo efeito olhando para ele com qualquer referência. Podemos ver um pouco dos bastidores tentando isso com jshell:

```
jshell> classe Carro {}

| classe criada Carro

jshell> Carro meuCarro = novo Carro()

meuCarro ==> Carro@21213b92

jshell> Carro outroCarro = meuCarro

outroCarro ==> Carro@21213b92

jshell> Carro notMyCar = novo Carro()

nãoMeuCarro ==> Carro@66480dd7
```

Observe o resultado da criação e atribuições. Aqui você pode ver que os tipos de referência Java vêm com um valor de ponteiro (21213b92, o lado direito do @) e seu

tipo (Car, o lado esquerdo do @). Quando criamos um novo objeto Car, notMyCar, obtemos um valor de ponteiro diferente. myCar e anotherCar apontam para o mesmo objeto; notMyCar aponta para um segundo objeto separado.

Inferindo tipos

Versões modernas de Java melhoraram continuamente a capacidade de inferir tipos de variáveis em muitas situações. A partir do Java 10, você pode usar a palavra-chave var em conjunto com a declaração e o início de uma variável e permitir que o compilador infira o tipo correto:

```
jshell> classe Car2 {}

| classe criada Car2
```

```
jshell> Car2 meuCar2 = new Car2()

meuCarro2 ==> Carro2@728938a9

jshell> var meuCarro3 = new Car2()

meuCarro3 ==> Carro2@6433a2
```

Observe a saída (reconhecidamente feia) ao criar myCar3 em jshell. Embora não tenhamos fornecido explicitamente o tipo como fizemos para myCar2, o compilador pode entender facilmente o tipo correto a ser usado e, de fato, obtemos um objeto Car2.

Passando referências

As referências de objetos são passadas para métodos da mesma maneira. Neste caso, myCar ou anotherCar serviriam como argumentos equivalentes para algum método hipotético, chamado myMethod(), em nossa classe hipotética:

```
meuMetodo(meuCarro);
```

Uma distinção importante, mas às vezes confusa, é que a referência em si é um valor (um endereço de memória). Esse valor é copiado quando você o atribui a uma variável ou o passa em uma chamada de método. Dado o nosso exemplo anterior, o argumento passado para um método (uma variável local do ponto de vista do método) é na verdade uma terceira referência ao objeto Car, além de myCar e anotherCar.

O método pode alterar o estado do objeto Car através dessa referência, chamando os métodos do objeto Car ou alterando suas variáveis. No entanto, myMethod() não pode alterar a noção do chamador sobre a referência a

myCar: ou seja, o método não pode alterar o myCar do chamador para apontar para um objeto Car diferente; ele pode alterar apenas sua própria referência. Isso ficará mais óbvio quando falarmos sobre métodos posteriormente.

Os tipos de referência sempre apontam para objetos (ou nulos) e os objetos são sempre definidos por classes. Semelhante aos tipos nativos, se você não inicializar uma instância ou variável de classe ao declará-la, o compilador atribuirá a ela o valor padrão nulo. Além disso, como os tipos nativos, as variáveis locais que possuem um

tipo de referência não são inicializadas por padrão, portanto, você deve definir seu próprio valor antes de usá-las. Entretanto, dois tipos especiais de tipos de referência –

arrays e interfaces – especificam o tipo de objeto para o qual apontam de uma maneira um pouco diferente.

Matrizes em Java são um tipo interessante de objeto criado automaticamente para conter uma coleção de algum outro tipo de objeto, conhecido como tipo base. Um elemento individual na matriz terá esse tipo base. (Portanto, um elemento de um array do tipo int[] será um int, e um elemento de um array do tipo String[] será um String.) Declarar um array cria implicitamente o novo tipo de classe projetado como um contêiner para sua base tipo, como você verá mais adiante neste capítulo.

Interfaces são um pouco mais sorrateiros. Uma interface define um conjunto de métodos e fornece a esse conjunto um tipo correspondente. Um objeto que implementa os métodos da interface pode ser referido

por esse tipo de interface, bem como por seu próprio tipo. Variáveis e argumentos de métodos podem ser declarados como sendo de tipos de interface, assim como outros tipos de classe, e qualquer objeto que implemente a interface pode ser atribuído a eles. Isso adiciona flexibilidade ao sistema de tipos e permite que Java cruze os limites da hierarquia de classes e crie objetos que efetivamente tenham muitos tipos. Abordaremos interfaces emcapítulo

5também.

Tipos genéricos ou tipos parametrizados, como mencionamos anteriormente, são uma extensão da sintaxe da classe Java que permite abstração adicional na maneira como as classes funcionam com outros tipos Java. Os genéricos permitem que o programador especialize uma classe sem alterar nenhum código dessa classe.

Cobrimos os genéricos em detalhes emCapítulo 7.

Uma palavra sobre cordas

Strings em Java são objetos; eles são, portanto, um tipo de referência. Os objetos String, entretanto, contam com alguma ajuda especial do compilador Java que os faz parecer mais com tipos primitivos. Valores de strings literais no código-fonte Java, uma série de caracteres ou sequências de escape entre aspas duplas, são transformados em objetos String pelo compilador. Você pode usar um literal String diretamente, passá-lo como argumento para métodos ou atribuí-lo a uma variável do tipo String:

```
System.out.println("Olá, Mundo...");
```

String s = "Eu sou a morsa...";

String t = "John disse: \"Eu sou a morsa...\"";

O símbolo + em Java está sobrecarregado para funcionar com strings e também com números regulares. Sobrecarga é um termo usado em linguagens que permitem usar o mesmo nome de método ou símbolo de operador ao trabalhar com tipos de dados distintos. Com números, + realiza adição. Com strings, + realiza concatenação, que é o que os programadores chamam de unir duas strings. Embora Java permita sobrecarga

arbitrária de métodos (mais em“Sobrecarga de método”), + é um dos poucos operadores sobrecarregados em Java:

String quote = "Fourscore e " + "sete anos atrás,"; String mais = citação + "nosso" + "pais" + "trouxe...";

// a citação agora é "Oitenta e sete anos atrás,"

//mais agora é "nossos pais trouxeram..."

Java constrói um único objeto String a partir dos literais de string concatenados e o fornece como resultado da expressão. (Mais sobre todas as coisas sobre String

emCapítulo 8.)

Declarações e Expressões

As instruções Java aparecem dentro de métodos e classes. Eles descrevem todas as atividades de um programa Java. Declarações e atribuições de variáveis, como as da seção anterior, são declarações, assim como estruturas básicas de linguagem, como condicionais if/then e loops. (Mais sobre essas estruturas

posteriormente neste capítulo.) Aqui estão algumas instruções em Java:

```
tamanho interno = 5;

se (tamanho > 10)

faça alguma coisa();

for (int x = 0; x < tamanho; x++) {

doSomethingElse();

doMoreThings();

}
```

Expressões produzir valores; Java avalia uma expressão para produzir um resultado.

Esse resultado pode então ser usado como parte de outra expressão ou em uma instrução. Chamadas de métodos, alocações de objetos e, claro, expressões matemáticas são exemplos de expressões:

```
// Estas são todas expressões Java válidas

novo objeto()

Matemática.pecado(3.1415)

42*64
```

Um dos princípios do Java é manter as coisas simples e consistentes. Para isso, quando não há outras restrições, as avaliações e inicializações em Java sempre ocorrem na ordem em que aparecem no código – da esquerda para a direita, de cima para baixo.

Você verá esta regra usada na avaliação de expressões de atribuição, chamadas de métodos e índices de array, para citar alguns casos. Em algumas outras línguas, a ordem de avaliação é mais complicada ou mesmo dependente da implementação. Java remove esse elemento de perigo ao definir de forma precisa e simples como o código é avaliado.

Isso não significa que você deva começar a escrever declarações obscuras e complicadas. Confiar na ordem de avaliação de expressões de maneiras complexas é um mau hábito de programação, mesmo quando funciona. Ele produz código difícil de ler e mais difícil de modificar.

Declarações

Em qualquer programa, as instruções fazem a verdadeira mágica. As declarações nos ajudam a implementar os algoritmos que mencionamos no início deste capítulo. Na verdade, eles não apenas ajudam, são justamente o ingrediente de programação que usamos; cada etapa de um algoritmo corresponderá a uma ou mais declarações. As declarações geralmente fazem uma de quatro coisas:

•

Reúna informações para atribuir a uma variável

•

Escreva a saída (para o seu terminal, para um JLabel, etc.)

•

Tomar uma decisão sobre quais instruções executar

•

Repita uma ou mais outras declarações

Instruções e expressões em Java aparecem dentro de um bloco de código. Um bloco de código contém uma série de instruções cercadas por uma chave aberta ({) e uma chave fechada (}). As instruções em um bloco de código podem incluir declarações de variáveis e a maioria dos outros tipos de instruções e expressões que mencionamos anteriormente:

```
{

tamanho interno = 5;

setNome("Máx");

// mais declarações poderiam seguir...

}
```

De certa forma, os métodos são apenas blocos de código que recebem parâmetros e podem ser chamados por seus nomes – por exemplo, um método hipotético setUpDog() pode começar assim:

```
setUpDog(Nome da string) {

tamanho interno = 5;

setNome(nome);

//faça qualquer outro trabalho de configuração...

}
```

As declarações de variáveis têm escopo definido em Java. Eles estão limitados ao bloco de código envolvente - ou seja, você não pode ver ou usar uma variável fora do conjunto de colchetes mais próximo:

```
{

// Os escopos são como Vegas...

// O que é declarado em um escopo permanece nesse escopo

int eu = 5;

}

eu = 6; // Erro em tempo de compilação, essa variável não existe i
```

Dessa forma, você pode usar blocos de código para agrupar instruções e variáveis arbitrariamente. O uso mais comum de blocos de código, entretanto, é definir um grupo de instruções para uso em uma instrução condicional ou iterativa.

condicionais if/else

Um dos conceitos-chave da programação é a noção de tomar uma decisão. “Se este arquivo existir” ou “Se o usuário tiver uma conexão WiFi” são exemplos de decisões que programas e aplicativos de computador tomam o tempo todo. Java usa a popular instrução if/else para muitos desses tipos de decisões.5Java define uma cláusula if/else da seguinte forma:

```
se (condição)
```

declaração1;

outro

declaração2;

Em inglês, você pode ler a instrução if/else como “se a condição for verdadeira, execute a instrução1. Caso contrário, execute a instrução2.”

A condição é uma expressão booleana e deve ser colocada entre parênteses. Uma expressão booleana, por sua vez, é um valor booleano (verdadeiro ou falso) ou uma expressão avaliada como um desses valores.6Por exemplo, i == 0 é uma expressão booleana que testa se o inteiro i contém o valor 0:

//nome do arquivo: ch04/examples/IfDemo.java

int eu = 0;

// você pode usar i agora para fazer outro trabalho e depois

// podemos testar para ver se alguma coisa mudou

se (eu == 0)

System.out.println("ainda é zero");

outro

System.out.println("i definitivamente não é zero");

Todo o exemplo anterior é em si uma instrução e pode ser aninhado em outra cláusula if/else. A cláusula if tem a funcionalidade comum de assumir duas formas diferentes: uma “linha única” ou um bloco. Veremos esse

mesmo padrão com outras instruções, como os loops discutidos na próxima seção. Se você tiver apenas uma instrução para executar (como as chamadas println() simples no trecho anterior), poderá colocar essa instrução solitária após o teste if ou após a palavra-chave else. Se precisar executar mais de uma instrução, você usa um bloco. O formulário do bloco é assim: se (condição) {

//condição era verdadeira, execute este bloco

declaração;

declaração;

// e assim por diante...

} outro {

// condição era falsa, execute este bloco

declaração;

declaração;

// e assim por diante...

}

Aqui, todas as instruções incluídas no bloco são executadas para qualquer ramificação utilizada. Podemos usar este formulário quando precisarmos fazer mais do que apenas imprimir uma mensagem. Por exemplo, poderíamos garantir que outra variável, talvez j, não seja negativa:

//nome do arquivo: ch04/examples/IfDemo.java

```
int j = 0;

// você pode usar j agora para fazer o trabalho como eu antes,

// então certifique-se de que o trabalho não caiu

// valor de j abaixo de zero

se (j < 0) {

System.out.println("j é menor que 0! Redefinindo.");

j = 0;

} outro {

System.out.println("j é positivo ou 0. Continuando.");

}
```

Observe que usamos chaves para a cláusula if, com duas instruções, e para a cláusula else, que ainda possui uma única chamada println(). Você sempre pode usar um bloco, se quiser. Mas se você tiver apenas uma instrução, o bloco com colchetes é opcional.

instruções de troca

Muitas linguagens suportam uma condicional “uma de muitas”, comumente conhecida como instrução switch ou case. Dada uma variável ou expressão, uma instrução switch fornece diversas opções que podem corresponder. E nós queremos dizer poder. Um valor não precisa corresponder a nenhuma das opções de opção; nesse caso nada acontece. Se a expressão corresponder a um caso, essa ramificação será executada. Se mais de um caso corresponder, a primeira correspondência vence.

A forma mais comum da instrução switch Java pega um número inteiro (ou um argumento de tipo numérico que pode ser automaticamente promovido para um tipo inteiro) ou uma string e seleciona entre uma série de ramificações case constantes alternativas:7

mudar (expressão) {

caso ExpressãoConstante:

declaração;

[caso expressãoconstante:

declaração;]

// ...

[padrão :

declaração;]

}

A expressão case para cada ramificação deve ser avaliada como um número inteiro constante diferente ou valor de string em tempo de compilação. Strings são comparadas usando o método String equals(), que discutiremos com mais detalhes

emCapítulo 8.

Você pode especificar um caso padrão opcional para capturar condições incompatíveis. Quando executado, o switch simplesmente encontra o ramo que corresponde à sua expressão condicional (ou ao ramo padrão) e executa a instrução correspondente. Mas esse não é o fim da história. Talvez de forma contra-intuitiva, a instrução

switch continua executando ramificações após a ramificação

correspondente até atingir o final da opção ou uma instrução especial chamada break.

Aqui estão alguns exemplos:

```
//nome do arquivo: ch04/examples/SwitchDemo.java

valor interno = 2;

mudar(valor) {

caso 1:

System.out.println(1);

caso 2:

System.out.println(2);

caso 3:

System.out.println(3);

}

// imprime 2 e 3
```

Usar break para encerrar cada branch é mais comum:

```
//nome do arquivo: ch04/examples/SwitchDemo.java

valor interno = BOM;

mudar (valor) {

caso BOM:
```

// algo bom

System.out.println("Bom");

quebrar;

caso RUIM:

// algo ruim

System.out.println("Ruim");

quebrar;

padrão:

// nenhum dos dois

System.out.println("Não tenho certeza");

quebrar;

}

// imprime apenas "Bom"

Neste exemplo, apenas uma ramificação – GOOD, BAD ou o padrão – é executada. O

comportamento de “continuar” do switch é justificado quando você deseja cobrir vários valores de caso possíveis com a(s) mesma(s) instrução(ões) sem precisar duplicar um monte de código:

//nome do arquivo: ch04/examples/SwitchDemo.java

valor int = MINÚSCULO;

```
Tamanho da string = "Desconhecido";

mudar(valor) {

caso MINISCULO:

caso TEENYWEENY:

caso PEQUENO:

tamanho = "Pequeno";

quebrar;

caso MÉDIO:

tamanho = "Médio";

quebrar;

caso GRANDE:

caso EXTRALARGE:

tamanho = "Grande";

quebrar;

}

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

Este exemplo agrupa efetivamente os seis valores possíveis em três casos. E esse recurso de agrupamento agora pode aparecer diretamente nas expressões. O Java

ofereceu expressões switch como um recurso de visualização que foi aprimorado e tornado permanente com o Java 14.

Por exemplo, em vez de imprimir os nomes dos tamanhos no exemplo acima, poderíamos atribuir nosso rótulo de tamanho diretamente a uma variável:

//nome do arquivo: ch04/examples/SwitchDemo.java

valor interno = EXTRALARGE;

Tamanho da string = switch(valor) {

case MINISCULE, TEENYWEENY, SMALL -> "Pequeno";

case MÉDIO -> "Médio";

caso GRANDE, EXTRALARGE -> "Grande";

padrão -> "Desconhecido";

}; // observe o ponto e vírgula! Ele completa a instrução switch

System.out.println("Seu tamanho é: " + tamanho);

// imprime "Seu tamanho é grande"

Observe a nova sintaxe da “seta” (um hífen seguido pelo símbolo de maior que). Você ainda usa entradas de caso separadas, mas com essa sintaxe de expressão, os valores

de caso são fornecidos em uma lista separada por vírgula, em vez de entradas em cascata separadas. Você então usa -> entre a lista e o valor a ser retornado. Este

formulário pode tornar a expressão switch um pouco mais compacta e (espero) mais legível.

fazer/enquanto loops

O outro conceito importante para controlar qual instrução será executada em seguida (fluxo de controle ou fluxo de controle em programês) é a repetição. Os computadores são realmente bons em fazer coisas repetidamente. A repetição de um bloco de código é feita com um loop. Existem várias instruções de loop diferentes em Java. Cada tipo de loop tem vantagens e desvantagens. Vejamos esses diferentes tipos agora.

As instruções iterativas do e while são executadas enquanto uma expressão booleana (geralmente chamada de condição do loop) retornar um valor verdadeiro. A estrutura básica desses loops é simples:

enquanto (condição)

declaração; //ou bloquear

fazer

declaração; //ou bloquear

enquanto (condição);

Um loop while é perfeito para aguardar alguma condição externa, como receber um novo email:

enquanto(mailQueue.isEmpty())

espere();

É claro que esse método hipotético wait() precisa ter um limite (normalmente um limite de tempo, como esperar

um segundo) para que ele termine e dê ao loop outra chance de ser executado. Mas depois de receber algum e-mail, você também deseja processar todas as mensagens que chegaram, não apenas uma. Novamente, um loop while é perfeito. Você pode usar um bloco de instruções entre chaves se precisar executar mais de uma instrução em seu loop. Considere uma impressora simples de contagem regressiva:

```
//nome do arquivo: ch04/examples/WhileDemo.java

contagem interna = 10;

enquanto(contagem > 0) {

System.out.println("Contagem regressiva: " +
contagem);

// talvez faça outras coisas úteis

// e decrementa nossa contagem

contagem = contagem - 1;

}

System.out.println("Concluído");
```

Neste exemplo, usamos o operador de comparação > para monitorar nossa variável de contagem. Queremos continuar trabalhando enquanto a contagem regressiva é positiva. Dentro do corpo do loop, imprimimos o valor atual de count e depois o reduzimos em um antes de repetir. Quando eventualmente reduzirmos a contagem para 0, o loop será interrompido porque a comparação retornará falso.

Ao contrário dos loops while que testam suas condições primeiro, um loop do-while (ou mais frequentemente apenas um loop do) sempre executa seu corpo de instrução pelo menos uma vez. Um exemplo clássico é a validação da entrada de um usuário.

Você sabe que precisa obter algumas informações, então solicita essas informações no corpo do loop. A condição do loop pode testar erros. Se houver algum problema, o loop será reiniciado e solicitará as informações novamente. Esse processo pode se repetir até que sua solicitação retorne sem erros e você saiba que tem boas informações.

```
fazer {

System.out.println("Digite um e-mail válido: ");

String email = askUserForEmail();

} enquanto (email.hasErrors());
```

Novamente, o corpo de um loop do é executado pelo menos uma vez. Se o usuário nos fornecer um endereço de e-mail válido na primeira vez, simplesmente não repetiremos o ciclo.

O loop for

Outra instrução de loop popular é o loop for. É excelente em contar. A forma mais geral do loop for também é um resquício da linguagem C. Pode parecer um pouco confuso, mas representa de forma compacta um pouco de lógica:

```
for (inicialização; condição; incrementador)
```

declaração; //ou bloquear

A seção de inicialização de variáveis pode declarar ou inicializar variáveis que são limitadas ao escopo do corpo for. O loop for então inicia uma possível série de rodadas nas quais a condição é verificada primeiro e, se verdadeira, a instrução do corpo (ou bloco) é executada. Após cada execução do corpo, as expressões incrementais são avaliadas para lhes dar a chance de atualizar variáveis antes do início da próxima rodada. Considere um loop de contagem clássico:

//nome do arquivo: ch04/examples/ForDemo.java

for (int i = 0; i < 100; i++) {

System.out.println(i);

int j = eu;

//faz qualquer outro trabalho necessário

}

Este loop será executado 100 vezes, imprimindo valores de 0 a 99. Declaramos e inicializamos uma variável, i, com zero. Usamos a cláusula de condição para ver se i é

menor que 100. Se for, o Java executa o corpo do loop. Na cláusula de incremento, aumentamos i em um. (Veremos mais sobre os operadores de comparação como < e >, bem como o atalho de incremento ++ na próxima seção,"Expressões".) Depois que i é incrementado, o loop volta para verificar a condição. Java continua repetindo essas etapas (condição, corpo, incremento) até chegar a 100.

Lembre-se de que a variável j é local para o bloco (visível apenas para instruções dentro dele) e não estará acessível ao código após o loop for. Se a condição de um loop for retornar falso na primeira verificação (por exemplo, se definirmos i como 1.000 na cláusula de inicialização), o corpo e a seção do incrementador nunca serão executados.

Você pode usar várias expressões separadas por vírgula nas seções de inicialização e incrementação do loop for. Por exemplo:

```
//nome do arquivo: ch04/examples/ForDemo.java

//gera algumas coordenadas

for (int x = 0, y = 10; x < y; x++, y--) {

System.out.println(x + ", " + y);

// fazemos outras coisas com nosso novo (x, y)...

}
```

Você também pode inicializar variáveis existentes fora do escopo do loop for dentro do bloco inicializador. Você pode fazer isso se quiser usar o valor final da variável de loop em outro lugar. Essa prática geralmente é desaprovada: é propensa a erros e pode dificultar o raciocínio do seu código. No entanto, é legal e você pode se deparar com uma situação em que esse comportamento faça mais sentido para você: interno x;

```
for(x = 0; x < algumHaltingValue; x++) {

System.out.print(x + ": ");
```

```
//faça qualquer trabalho que você precisar...

}

// x ainda é válido e disponível

System.out.println("Após o loop, x é: " + x);
```

Na verdade, você pode deixar de lado completamente a etapa de inicialização se quiser trabalhar com uma variável que já possui um bom valor inicial: int x = 1;

```
for(; x < algumHaltingValue; x++) {

System.out.print(x + ": ");

//faça qualquer trabalho que você precisar...

}
```

Observe que você ainda precisa do ponto e vírgula que normalmente separa a etapa de inicialização da condição.

O loop for aprimorado

O auspiciosamente apelidado de “loop for aprimorado” do Java atua como a instrução foreach em algumas outras linguagens, iterando sobre uma série de valores em um array ou outro tipo de coleção:

```
for (varDeclaration: iterável)

declaração_ou_bloco;
```

O loop for aprimorado pode ser usado para fazer loop em matrizes de qualquer tipo, bem como em qualquer tipo de objeto Java que implemente a interface

java.lang.Iterable. (Teremos mais a dizer sobre arrays, classes e interfaces emcapítulo

5.) Isso inclui a maioria das classes da API Java Collections (consulteCapítulo 7). Aqui estão alguns exemplos:

```
//nome do arquivo:
ch04/examples/EnhancedForDemo.java

int [] arrayOfInts = new int [] { 1, 2, 3, 4 };

total interno = 0;

for(int i: arrayOfInts) {

System.out.println(i);

total = total + eu;

}

System.out.println("Total: " + total);

// ArrayList é uma classe de coleção popular

ArrayList<String> lista = new ArrayList<String>();

lista.add("foo");

lista.add("barra");

for (String s: lista)

System.out.println(s);
```

Novamente, não discutimos arrays ou a classe ArrayList e sua sintaxe especial neste exemplo. O que estamos

mostrando aqui é a sintaxe do loop for aprimorado iterando sobre um array e uma lista de valores de string. A brevidade deste formulário o torna popular sempre que você precisa trabalhar com uma coleção de itens.

interromper/continuar

A instrução Java break e seu amigo continue também podem ser usados para encurtar um loop ou instrução condicional saltando para fora dela. Uma pausa faz com que o Java pare a instrução do loop atual (ou switch) e ignore o resto do corpo. Java começa a executar o código que vem após o loop. No exemplo a seguir, o loop while continua indefinidamente até que o método watchForErrors() retorne verdadeiro, acionando uma instrução break que interrompe o loop e prossegue no ponto marcado “após o loop while”:

```
enquanto(verdadeiro) {

if (watchForErrors())

quebrar;

// Nenhum erro ainda, então faça algum trabalho...

}

// O "break" fará com que a execução

// retome aqui, após o loop while
```

Uma instrução continue faz com que os loops for e while passem para a próxima iteração, retornando ao ponto onde verificam sua condição. O exemplo a seguir imprime os números de 0 a 9, ignorando o número 5:

```
//nome do arquivo: ch04/examples/ForDemo.java

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

se (eu == 5)

continuar;

System.out.println(i);

}
```

As instruções break e continue se parecem com as da linguagem C, mas os formulários Java têm a capacidade adicional de usar um rótulo como argumento e saltar vários níveis para o escopo do ponto rotulado no código. Esse uso não é muito comum na codificação Java diária, mas pode ser importante em casos especiais. Aqui está o que parece:

```
rótuloUm:

enquanto (condição1) {

// ...

rótuloDois:

enquanto (condição2) {

// ...

if (pequeno problema)

quebrar; // Sairá apenas deste loop

se (grande problema)
```

quebrar rótuloUm; // Irá sair de ambos os loops

}

//depois do rótuloTwo

}

//depois do rótuloOne

Instruções envolventes, como blocos de código, condicionais e loops, podem ser rotuladas com identificadores como labelOne e labelTwo. Neste exemplo, um break ou continue sem argumento tem o mesmo efeito dos exemplos anteriores. Uma pausa faz com que o processamento seja retomado no ponto denominado “após rótuloTwo”; um continue imediatamente faz com que o loop labelTwo retorne ao seu teste de condição.

Poderíamos usar a instrução break labelTwo na instrução smallProblem. Teria o mesmo efeito que um break comum, mas o break labelOne, como visto na instrução bigProblem, rompe ambos os níveis e retoma no ponto rotulado “após labelOne”. Da mesma forma, continue labelTwo serviria como um continue normal, mas continue labelOne retornaria ao teste do loop labelOne. As instruções break e continue multiníveis removem a justificativa principal para a tão difamada instrução goto em C/C++.8

Existem algumas instruções Java que não discutiremos agora. As instruções try, catch e finalmente são usadas no tratamento de exceções, como discutiremos emCapítulo 6.

A instrução sincronizada em Java é usada para coordenar o acesso a instruções entre vários threads de execução; verCapítulo 9para uma discussão sobre sincronização de threads.

Declarações inacessíveis

Numa nota final, devemos mencionar que o compilador Java sinaliza instruções inacessíveis como erros em tempo de compilação. Uma instrução inacessível é aquela que o compilador determina que nunca será chamada. É claro que muitos métodos ou pedaços de código podem nunca ser chamados em seu programa, mas o compilador detecta apenas aqueles que ele pode “provar” que nunca foram chamados com alguma verificação inteligente em tempo de compilação. Por exemplo, um método com uma instrução de retorno incondicional no meio causa um erro em tempo de compilação, assim como um método com uma condicional que o compilador pode dizer que nunca será cumprida:

```
se (1 <2) {

// Este branch sempre roda e o compilador sabe disso

System.out.println("1 é, na verdade, menor que 2");

retornar;

} outro {

// instruções inacessíveis, este branch nunca é executado

System.out.println("Olha só, parece que erramos em \"matemática\".");
```

```
}
```

Você deve corrigir os erros inacessíveis antes de concluir a compilação. Felizmente, a maioria dos casos desse erro são apenas erros de digitação que podem ser facilmente corrigidos. Nas raras ocasiões em que esta verificação do compilador revela uma falha na sua lógica e não na sua sintaxe, você sempre pode reorganizar ou excluir o código que não pode ser executado.

Expressões

Uma expressão produz um resultado, ou valor, quando é avaliada. O valor de uma expressão pode ser do tipo numérico, como em uma expressão aritmética; um tipo de referência, como em uma alocação de objetos; ou o tipo especial, void, que é o tipo declarado de um método que não retorna um valor. No último caso, a expressão é avaliada apenas pelos seus efeitos colaterais; isto é, o trabalho que realiza além de

produzir um valor. O compilador conhece o tipo de uma expressão. O valor produzido em tempo de execução terá este tipo ou, no caso de um tipo de referência, um subtipo compatível (atribuível). (Mais sobre essa compatibilidade emcapítulo 5.)

Já vimos diversas expressões em nossos programas de exemplo e trechos de código.

Também veremos muitos outros exemplos de expressões na seção"Atribuição".

Operadores

Os operadores ajudam você a combinar ou alterar expressões de diversas maneiras.

Eles “operam” expressões. Java suporta quase todos os operadores padrão da linguagem C. Esses operadores também têm a mesma precedência em Java e em C, conforme mostrado emTabela 4-3.9

Tabela 4-3. Operadores Java

Tipo de

Precedência Operador

operando

Descrição

1

++, -

Aritmética

Incremento e decremento

1

+, -

Aritmética

Unário mais e menos

1

~

Integrante

Complemento bit a bit

1

!

boleano

Complemento lógico

1

(tipo)

Qualquer

Elenco

2

*, /,%

Aritmética

Multiplicação, divisão, resto

3

+, -

Aritmética

Adição e subtração

3

+

Corda

Concatenação de strings

4

<<

Integrante

Desvio à esquerda

4

>>

Integrante

Deslocamento para a direita com

extensão de sinal

4

>>>

Integrante

Deslocamento para a direita sem

extensão

5

<, <=, >, >=

Aritmética

Comparação numérica

5

instancia

Objeto

Comparação de tipos

de

6

==,!=

Primitivo

Igualdade e desigualdade de valor

6

==,!=

Objeto

Igualdade e desigualdade de

referência

Tipo de

Precedência Operador

operando

Descrição

7

&

Integrante

E bit a bit

7

&

boleano

Booleano E

8

^

Integrante

XOR bit a bit

8

^

boleano

XOR booleano

9

|

Integrante

OU bit a bit

9

|

boleano

Booleano OU

10

&&

boleano

Condicional E

11

||

boleano

OU condicional

12

?:

N / D

Operador ternário condicional

13

=

Qualquer

Atribuição

Devemos também notar que o operador percentagem (%) não é estritamente um módulo, mas um resto, e pode ter um valor negativo. Tente brincar com alguns desses operadores no jshell para ter uma noção melhor de seus efeitos. Se você é novo em programação, é particularmente útil se familiarizar com os operadores e

sua ordem de precedência. Você encontrará expressões e operadores regularmente, mesmo ao executar tarefas comuns em seu código:

```
jshell> int x = 5

x ==> 5

jshell> int y = 12

y ==> 12

jshell> int sumOfSquares = x * x + y * y

somaDeQuadrados ==> 169

jshell> int ordemexplícita = (((x * x) + y) * y)

ordemexplícita ==> 444

jshell> sumOfSquares% 5

US$ 7 ==> 4
```

Java também adiciona alguns novos operadores. Como vimos, você pode usar o operador + com valores String para realizar a concatenação de strings. Como todos os tipos inteiros em Java são valores assinados, você pode usar o operador >> para executar uma operação de deslocamento aritmético para a direita com extensão de sinal. O operador >>> trata o operando como um número sem sinal10e executa uma

mudança aritmética para a direita sem extensão de sinal. Como programadores, não precisamos manipular os bits individuais em nossas variáveis tanto quanto costumávamos fazer, então você provavelmente não verá esses operadores de deslocamento com muita

frequência. Se eles surgirem em exemplos de codificação ou análise de dados binários que você lê on-line, sinta-se à vontade para acessar o jshell para ver como eles funcionam. Esse tipo de jogo é um dos nossos usos favoritos do jshell!

Atribuição

Embora declarar e inicializar uma variável seja considerado uma instrução sem valor resultante, a atribuição de variável por si só é, na verdade, uma expressão: int eu, j; //instrução sem valor resultante

int k = 6; // também uma instrução sem resultado

eu = 5; // uma instrução e uma expressão

Normalmente, confiamos na atribuição apenas pelos seus efeitos colaterais, como nas duas primeiras linhas acima, mas uma atribuição pode ser usada como um valor em outra parte de uma expressão. Alguns programadores usarão esse fato para atribuir um determinado valor a múltiplas variáveis de uma só vez:

j = (eu = 5);

// j e eu agora temos 5 anos

Confiar extensivamente na ordem de avaliação (neste caso, usando atribuições compostas) pode tornar o código obscuro e difícil de ler. Não recomendamos isso, mas esse tipo de inicialização aparece em exemplos online.

O valor nulo

A expressão null pode ser atribuída a qualquer tipo de referência. Significa “sem referência”. Uma referência nula não pode ser usada para fazer referência a nada e tentar fazer isso gera uma NullPointerException em tempo de execução. Lembrar

de“Tipos de referência”esse null é o valor padrão atribuído a variáveis de classe e instância não inicializadas; certifique-se de realizar suas inicializações antes de usar variáveis de tipo de referência para evitar essa exceção.

Acesso variável

O operador ponto (.) é usado para selecionar membros de uma classe ou instância de objeto. (Falaremos sobre membros em detalhes nos capítulos seguintes.) Ele pode recuperar o valor de uma variável de instância (de um objeto) ou de uma variável estática (de uma classe). Também pode especificar um método a ser invocado em um objeto ou classe:

```
int i = meuObjeto.comprimento;

String s = meuObjeto.nome;

meuObject.someMethod();
```

Uma expressão do tipo referência pode ser usada em avaliações compostas (múltiplos usos da operação de ponto em uma expressão) selecionando outras variáveis ou métodos no resultado:

```
int len = meuObjeto.nome.comprimento();

int inicialLen = meuObjeto.nome.substring(5,
10).comprimento();
```

A primeira linha encontra o comprimento de nossa variável de nome invocando o método length() do objeto String. No segundo caso, damos um passo intermediário e solicitamos uma substring da string do nome. O método substring da classe String também retorna uma referência String, para a qual perguntamos o comprimento.

Operações compostas como essa também são chamadas de chamadas de métodos de encadeamento. Uma operação de seleção encadeada que já usamos muito é chamar o método println() na variável fora da classe System:

System.out.println("chamando println para fora");

Invocação de método

Métodos são funções que residem dentro de uma classe e podem ser acessíveis através da classe ou de suas instâncias, dependendo do tipo de método. Invocar um método significa executar suas instruções corporais, passando quaisquer variáveis de parâmetro necessárias e possivelmente obtendo um valor em troca. Uma invocação de método é uma expressão que resulta em um valor. O tipo do valor é o tipo de retorno do método:

System.out.println("Olá, Mundo...");

int meuComprimento = minhaString.comprimento();

Aqui, invocamos os métodos println() e length() em objetos diferentes. O método length() retornou um valor inteiro; o tipo de retorno de println() é void (sem valor).

Vale ressaltar que println() produz saída, mas nenhum valor. Não podemos atribuir esse método a uma variável

como fizemos acima com length():

jshell> String minhaString = "Olá!"

minhaString ==> "Olá!"

jshell> int meuComprimento =
minhaString.comprimento()

meuComprimento ==> 9

jshell> int erro = System.out.println("Isso é um erro.")

| Erro:

| tipos incompatíveis: void não pode ser convertido em
int

| erro interno = System.out.println("Isso é um erro.");

| ^--^

Os métodos constituem a maior parte de um programa Java. Embora você possa escrever alguns aplicativos triviais que existem inteiramente dentro de um único método main() de uma classe, você descobrirá rapidamente que precisa dividir as coisas. Os métodos não apenas tornam seu aplicativo mais legível, mas também abrem

as portas para aplicativos complexos, interessantes e úteis que simplesmente não são possíveis sem eles. Na verdade, veja nossos aplicativos gráficos Hello World

em“OláJava”. Usamos vários métodos definidos para a classe JFrame.

Estes são exemplos simples, mas emcapítulo 5você verá que fica um pouco mais complexo quando existem métodos com o mesmo nome, mas com tipos de parâmetros diferentes na mesma classe, ou quando um método é redefinido em uma subclasse.

Declarações, expressões e algoritmos

Vamos reunir uma coleção de declarações e expressões desses diferentes tipos para atingir um objetivo real. Em outras palavras, vamos escrever algum código Java para implementar um algoritmo. Um exemplo clássico de algoritmo é o processo de Euclides para encontrar o máximo denominador comum (MDC) de dois números. Ele usa um processo simples (embora tedioso) de subtração repetida. Podemos usar o loop while do Java, uma condicional if/else e algumas atribuições para realizar o trabalho:

```
//nome do arquivo: ch04/examples/EuclidGCD.java

intuma = 2701;

interno b = 222;

enquanto (b! = 0) {

se (a > b) {

uma = uma - b;

} outro {

b=b - uma;

}

}
```

System.out.println("GCD é " + a);

Não é sofisticado, mas funciona – e é exatamente o tipo de tarefa que os programas de computador executam perfeitamente. É para isso que você está aqui! Bem, você provavelmente não está aqui pelo maior denominador comum de 2701 e 222 (37, a propósito), mas está aqui para começar a formular as soluções para problemas como algoritmos e traduzir esses algoritmos em código Java executável.

Esperamos que mais algumas peças do quebra-cabeça da programação estejam começando a se encaixar. Mas não se preocupe se essas ideias ainda estiverem confusas. Todo esse processo de codificação exige muita prática. Para um dos exercícios de codificação deste capítulo, queremos que você tente colocar o bloco de código acima em uma classe Java real dentro do método main(). Tente alterar os valores de a e b. EmCapítulo 8veremos como converter strings em números, para que você possa encontrar o GCD simplesmente executando o programa novamente, passando dois números como parâmetros para o método main(), conforme mostrado

emFigura 2-10, sem recompilar.

Criação de objeto

Objetos em Java são alocados com o operador new:

Objeto o = new Objeto();

O argumento para new é o construtor da classe. O construtor é um método que sempre possui o mesmo nome da classe. O construtor especifica quaisquer parâmetros necessários para criar uma instância do objeto. O valor da nova expressão é uma referência do

tipo do objeto criado. Os objetos sempre têm um ou mais construtores, embora nem sempre estejam acessíveis para você.

Vemos a criação de objetos em detalhes emcapítulo 5. Por enquanto, observe apenas que a criação de objetos também é um tipo de expressão e que o resultado é uma referência de objeto. Uma pequena estranheza é que a ligação de new é “mais rígida”

do que a do seletor de ponto (.). Um efeito colateral popular desse detalhe é que você pode criar um novo objeto e invocar um método nele sem atribuir o objeto a uma variável de tipo de referência. Por exemplo, você pode precisar da hora atual do dia, mas não do restante das informações encontradas em um objeto Date. Você não precisa manter uma referência à data recém-criada, basta pegar o atributo necessário por meio do encadeamento:

```
jshell> int horas = new Date().getHours()

horas ==> 13
```

A classe Date é uma classe utilitária que representa a data e hora atuais. Aqui criamos uma nova instância de Date com o operador new e chamamos seu método getHours() para recuperar a hora atual como um valor inteiro. A referência do objeto Date dura o suficiente para atender a chamada do método getHours() e é então liberada e eventualmente coletada como lixo (consulte"Coleta de lixo").

Chamar métodos a partir de uma nova referência de objeto dessa maneira é uma questão de estilo. Certamente seria mais claro alocar uma variável intermediária do tipo Date para armazenar o novo objeto

e então chamar seu método getHours(). No entanto, é comum combinar operações como fizemos para obter as horas acima. À

medida que você aprende Java e se sente confortável com suas classes e tipos, provavelmente adotará alguns desses padrões. Até então, porém, não se preocupe em ser “detalhado” em seu código. Clareza e legibilidade são mais importantes do que floreios estilísticos à medida que você trabalha neste livro.

O operador instanceof

Você usa o operador instanceof para determinar o tipo de um objeto em tempo de execução. Ele testa se um objeto é do mesmo tipo ou um subtipo do tipo de destino.

(Novamente, falaremos mais sobre essa hierarquia de classes!) Isso é o mesmo que perguntar se o objeto pode ser atribuído a uma variável do tipo de destino. O tipo de destino pode ser uma classe, interface ou tipo de array. instanceof retorna um valor booleano que indica se o objeto corresponde ao tipo. Vamos tentar em jshell:

jshell> booleano b

b ==> falso

jshell> String str = "alguma coisa"

str ==> "alguma coisa"

jshell> b = (str instância de String)

b ==> verdadeiro

jshell> b = (str instância do objeto)

b ==> verdadeiro

jshell> b = (str instância de data)

| Erro:

| tipos incompatíveis: java.lang.String não pode ser convertido em java.u til.Date

| b = (str instância de data)

| ^-^

Observe que a instância final do teste retorna um erro. Com seu forte senso de tipos, Java muitas vezes pode capturar combinações impossíveis em tempo de compilação.

Semelhante ao código inacessível, o compilador não permitirá que você prossiga até corrigir o problema.

O operador instanceof também informa corretamente se o objeto é do tipo de um array:

if (minhaVariável instância de byte[]) {

// agora temos certeza que myVariable é um array de bytes

// vá em frente com seu trabalho de array aqui...

}

Também é importante observar que o valor null não é considerado uma instância de nenhuma classe. O teste a seguir retorna falso, independentemente do tipo atribuído à variável:

jshell> String s = nulo

s ==> nulo

jshell> Data d = nulo

d ==> nulo

jshell>s instância de String

$7 ==> falso

jshell> d instância de Data

$8 ==> falso

jshell> d instância de String

| Erro:

| tipos incompatíveis: java.util.Date não pode ser convertido em java.lan g.String

| d instância de String

| ^

Portanto, null nunca é uma “instância de” nenhuma classe, mas Java ainda rastreia os tipos de suas variáveis e não permitirá testar (ou converter) entre tipos incompatíveis.

Matrizes

Uma matriz é um tipo especial de objeto que pode conter uma coleção ordenada de elementos. O tipo dos elementos do array é chamado de tipo base do array; o número de elementos que ele contém é um atributo fixo

denominado comprimento. Java suporta arrays de todos os tipos primitivos, bem como tipos de referência. Para criar um array com um tipo base de byte, por exemplo, você poderia usar o tipo byte[]. Da mesma forma, você pode criar um array com o tipo base String com String[].

Se você fez alguma programação em C ou C++, a sintaxe básica dos arrays Java deve parecer familiar. Você cria uma matriz de comprimento especificado e acessa os elementos com o operador de índice, []. Ao contrário dessas linguagens, entretanto, os arrays em Java são objetos verdadeiros e de primeira classe. Um array é uma instância de uma classe especial de array Java e possui um tipo correspondente no sistema de tipos. Isso significa que para usar um array, como acontece com qualquer outro objeto, primeiro você declara uma variável do tipo apropriado e depois usa o operador new para criar uma instância dela.

Os objetos array diferem de outros objetos em Java em três aspectos:

•

Java cria implicitamente um tipo de classe Array especial para nós sempre que declaramos um novo tipo de array. Não é estritamente necessário conhecer esse processo para usar arrays, mas ajudará a entender sua estrutura e seu relacionamento com outros objetos em Java posteriormente.

•

Java nos permite usar o operador [] para acessar e atribuir elementos de array para que os arrays tenham a aparência que muitos programadores experientes esperam. Poderíamos implementar nossas próprias

classes que agem como arrays, mas teríamos que nos contentar em ter métodos como get() e set() em vez de usar a notação especial [].

•

Java fornece uma forma especial correspondente do operador new que nos permite construir uma instância de um array com um comprimento

especificado com a notação [] ou inicializá-lo diretamente a partir de uma lista estruturada de valores.

As matrizes facilitam o trabalho com blocos de informações relacionadas, como as linhas de texto em um arquivo ou as palavras em uma dessas linhas. Nós os usamos

frequentemente em exemplos ao longo do livro; você verá muitos exemplos de criação e manipulação de arrays com a notação [] neste e nos próximos capítulos.

Tipos de matriz

Uma variável de matriz é denotada por um tipo base seguido por colchetes vazios, [].

Alternativamente, Java aceita uma declaração no estilo C com colchetes colocados após o nome do array.

As seguintes declarações são equivalentes:

int[] arrayOfInts; // preferido

int [] arrayOfInts; //espaçamento é opcional

int arrayOfInts[]; // estilo C, permitido

Em cada caso, declaramos arrayOfInts como um array de inteiros. O tamanho do array ainda não é um problema porque estamos apenas declarando uma variável do tipo array. Ainda não criamos uma instância real da classe array, nem seu armazenamento associado. Nem é possível especificar o comprimento de um array ao declarar uma variável do tipo array. O tamanho é estritamente uma função do próprio objeto array, não da referência a ele.

Matrizes de tipos de referência podem ser criadas da mesma maneira:

String[] algumasStrings;

JLabel alguns rótulos[];

Criação e inicialização de array

Você usa o operador new para criar uma instância de um array. Após o operador new, especificamos o tipo base do array e seu comprimento com uma expressão inteira entre colchetes. Podemos usar esta sintaxe para criar instâncias de array com armazenamento real para nossas variáveis declaradas recentemente. Como expressões são permitidas, podemos até fazer alguns cálculos entre colchetes: número interno = 10;

arrayOfInts = new int[42];

algumasStrings = new String[número + 2];

Também podemos combinar as etapas de declaração e alocação do array:

double[] algunsNúmeros = new double[20];

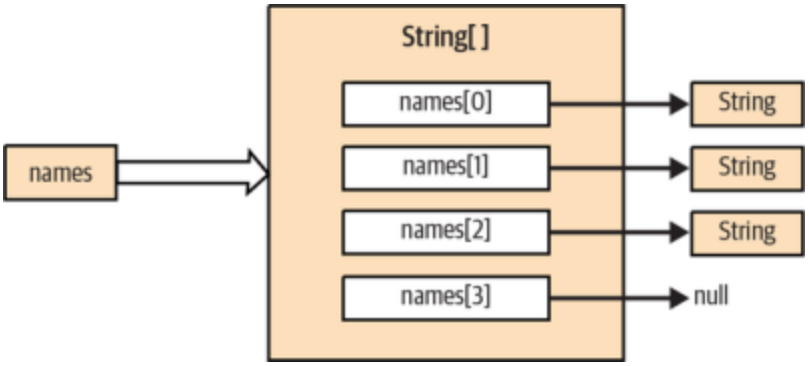
Componente[] widgets = novo Componente[12];

Os índices da matriz começam com zero. Assim, o primeiro elemento de

someNumbers[] possui índice 0 e o último elemento possui índice 19. Após a criação, os próprios elementos do array são inicializados com os valores padrão para seu tipo.

Para tipos numéricos, isso significa que os elementos são inicialmente zero: int[] notas = new int[30];

// notas do primeiro elemento[0] == 0



// ...

// notas do último elemento[19] == 0

Os elementos de um array de objetos são referências aos objetos — assim como as variáveis individuais para as quais eles apontam — mas na verdade não contêm instâncias dos objetos. O valor padrão de cada elemento é, portanto, nulo até atribuirmos instâncias de objetos apropriados:

Nomes de string[] = new String[42];

//nomes[0] == nulo

```
// nomes[1] == nulo

// ...
```

Esta é uma distinção importante que pode causar confusão. Em muitas outras linguagens, o ato de criar um array é o mesmo que alocar armazenamento para seus elementos. Em Java, um array de objetos recém-alocado contém, na verdade, apenas variáveis de referência, cada uma com o valor nulo.11Isso não quer dizer que não haja memória associada a um array vazio; é necessária memória para armazenar essas referências (os “slots” vazios no array).Figura 4-3ilustra a matriz de nomes do exemplo anterior.

Figura 4-3. Uma matriz Java

Construímos nossa variável de nomes como um array de strings (String[]). Este objeto String[] específico contém quatro variáveis do tipo String. Atribuímos objetos String aos três primeiros elementos do array. O quarto tem o valor padrão nulo.

Java suporta a construção de chaves {} no estilo C para criar um array e inicializar seus elementos:

```
jshell> int[] primos = { 2, 3, 5, 7, 7+4 };

primos ==> int[5] { 2, 3, 5, 7, 11 }

jshell> primos[2]

$ 12 ==> 5

jshell> primos[4]

$ 13 ==> 11
```

Um objeto de matriz do tipo e comprimento adequados é criado implicitamente e os valores da lista de expressões separadas por vírgula são atribuídos aos seus elementos. Observe que não usamos a palavra-chave new ou o tipo de array aqui. Java infere o uso de new da atribuição.

Também podemos usar a sintaxe {} com uma matriz de objetos. Neste caso, cada expressão deve ser avaliada como um objeto que pode ser atribuído a uma variável do tipo base do array ou ao valor nulo. aqui estão alguns exemplos:

jshell> String[] verbos = { "correr", "pular", "ocultar" }

verbos ==> String[3] { "correr", "pular", "ocultar" }

jshell> importar javax.swing.JLabel

jshell> JLabel simLabel = new JLabel("Sim")

simLabel ==> javax.swing.JLabel...

jshell> JLabel noLabel = new JLabel("Não")

noLabel ==> javax.swing.JLabel...

jshell> JLabel[] escolhas={ simLabel, noLabel,

...> new JLabel("Talvez") }

escolhas ==> JLabel[3] { javax.swing.JLabel ...
ition=CENTER] }

jshell> Object[] qualquer coisa = { "run", yesLabel, new Date() }

qualquer coisa ==> Object[3] { "run", javax.swing.JLabe
... 2023 }

As seguintes declarações e instruções de inicialização são equivalentes: JLabel[] trêsLabels = new JLabel[3];

JLabel[] trêsLabels = { null, null, null };

Obviamente, o primeiro exemplo é melhor quando você tem um grande número de coisas para guardar. A maioria dos programadores usa a inicialização de chaves apenas quando tem objetos reais prontos para serem armazenados no array.

Usando matrizes

O tamanho de um objeto array está disponível na variável pública length: jshell> char[] alfabeto = novo char[26]

alfabeto ==> char[26] { '\000', '\000' ... , '\000' }

jshell> String[] mosqueteiros = { "um", "dois", "três" }

mosqueteiros ==> String[3] { "um", "dois", "três" }

jshell> alfabeto.comprimento

$ 24 ==> 26

jshell> mosqueteiros.length

US$ 25 ==> 3

comprimentoé o único campo acessível de um array; é uma variável, não um método como em muitas outras linguagens. Felizmente, o compilador avisa quando você

acidentalmente usa parênteses como alfabeto.length(), como todo mundo faz de vez em quando.

O acesso ao array em Java é exatamente como o acesso ao array em muitas outras linguagens; você acessa um elemento colocando uma expressão com valor inteiro entre colchetes após o nome do array. Esta sintaxe funciona tanto para acessar elementos individuais existentes quanto para atribuir novos elementos. Podemos conseguir nosso segundo mosqueteiro assim:

// lembre-se que o primeiro índice é 0!

jshell> System.out.println(mosqueteiros[1])

dois

O exemplo a seguir cria uma matriz de objetos JButton chamada keyPad. Em seguida, ele preenche o array com botões, usando nossos colchetes e a variável loop como índice:

JButton[] keyPad = new JButton[10];

for (int i=0; i < keyPad.length; i++)

keyPad[i] = new JButton("Botão " + i);

Lembre-se de que também podemos usar o loop for aprimorado para iterar sobre valores de array. Aqui vamos usá-lo para imprimir todos os valores que acabamos de atribuir:

para (JButton b: teclado)

System.out.println(b);

A tentativa de acessar um elemento que está fora do intervalo do array gera uma ArrayIndexOutOfBoundsException. Este é um tipo de

RuntimeException, então você pode capturá-lo e tratá-lo sozinho, se realmente espera, ou ignorá-lo, como discutiremos emCapítulo 6. Aqui está uma amostra da sintaxe try/catch que o Java usa para agrupar esse código potencialmente problemático:

```
String [] estados = new String [50];

tentar {

estados[0] = "Alabama";

estados[1] = "Alasca";

// Mais 48...

estados[50] = "Terra do McDonald's"; // Erro: array fora dos limites

} catch (ArrayIndexOutOfBoundsException err) {

System.out.println("Erro tratado: " + err.getMessage());

}
```

É uma tarefa comum copiar uma série de elementos de um array para outro. Uma maneira de copiar arrays é usar o método arraycopy() de baixo nível da classe System: System.arraycopy(fonte, sourceStart, destino, destStart, comprimento); O exemplo a seguir dobra o tamanho da matriz de nomes de um exemplo anterior:

```
String[] tmpVar = new String [2 * nomes.comprimento];
System.arraycopy(nomes, 0, tmpVar, 0, nomes.comprimento);

nomes = tmpVar;
```

Aqui alocamos e atribuímos uma variável temporária, tmpVar, como um novo array, com o dobro do tamanho dos nomes. Usamos arraycopy() para copiar os elementos dos nomes para o novo array. Finalmente, atribuímos um array temporário aos nomes. Se não houver referências restantes ao array antigo de nomes após atribuir o novo array aos nomes, o array antigo será coletado como lixo na próxima passagem.

Talvez uma maneira mais fácil de realizar a mesma tarefa seja usar os métodos copyOf() ou copy OfRange() da classe java.util.Arrays:

```
jshell> byte[] barra = novo byte[] { 1, 2, 3, 4, 5 }

barra ==> byte[5] { 1, 2, 3, 4, 5 }

jshell> byte[] barCopy = Arrays.copyOf(bar, bar.length)

barraCópia ==> byte[5] { 1, 2, 3, 4, 5 }

jshell> byte[] expandido = Arrays.copyOf(bar,
bar.length+2)

expandido ==> byte[7] { 1, 2, 3, 4, 5, 0, 0 }

jshell> byte[] primeiroTrês = Arrays.copyOfRange(bar, 0,
3)

primeirosTrês ==> byte[3] { 1, 2, 3 }

jshell> byte[] lastThree = Arrays.copyOfRange(bar, 2,
bar.length)

últimosTrês ==> byte[3] { 3, 4, 5 }

jshell> byte[] plusTwo = Arrays.copyOfRange(bar, 2,
bar.length+2)
```

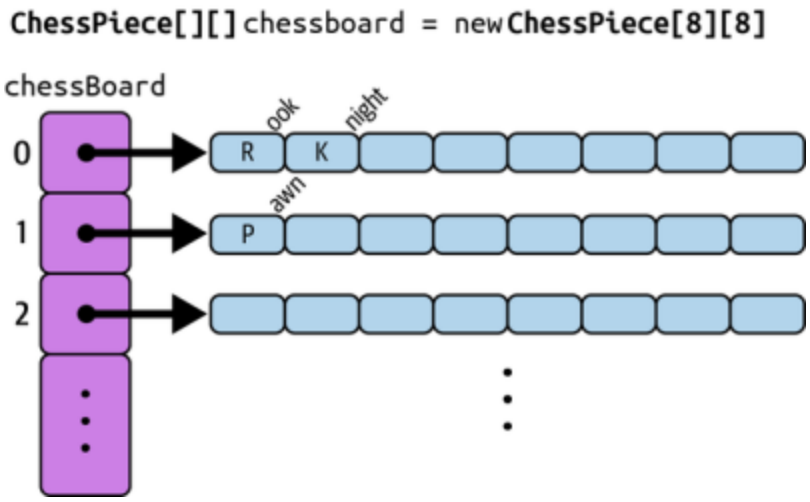
maisDois ==> byte[5] { 3, 4, 5, 0, 0 }

O método copyOf() pega o array original e um comprimento alvo. Se o comprimento alvo for maior que o comprimento original da matriz, a nova matriz será preenchida (com zeros ou nulos) até o comprimento desejado. O copyOfRange() leva um índice inicial (inclusivo) e um índice final (exclusivo) e um comprimento desejado, que também será preenchido, se necessário.

Matrizes anônimas

Muitas vezes é conveniente criar arrays descartáveis: arrays que são usados em um lugar e nunca referenciados em nenhum outro lugar. Esses arrays não precisam de nomes porque você nunca mais os consulta nesse contexto. Por exemplo, você pode querer criar uma coleção de objetos para passar como argumento para algum método.

É bastante fácil criar um array normal nomeado, mas se você realmente não trabalhar com o array (se usar o array apenas como suporte para alguma coleção), não precisará nomear esse suporte temporário. Java facilita a criação de arrays “anônimos” (sem nome).



Digamos que você precise chamar um método chamado setPets(), que recebe um array de objetos Animal como argumentos. Desde que Cat e Dog sejam subclasses de Animal, veja como chamar setPets() usando um array anônimo:

Cachorro pete = novo Cachorro ("dourado");

Cachorro mj = novo Cachorro ("preto e branco");

Esconderijo de gato = novo gato ("laranja");

setPets (new Animal[] { pete, mj, stash });

A sintaxe é semelhante à inicialização de um array em uma declaração de variável.

Definimos implicitamente o tamanho do array e preenchemos seus elementos usando a notação de chaves. No entanto, como esta não é uma declaração de variável, temos que usar explicitamente o operador new e o tipo de array para criar o objeto array.

Matrizes multidimensionais

Java oferece suporte a arrays multidimensionais na forma de arrays de outros arrays.

Você cria uma matriz multidimensional com sintaxe semelhante à C, usando vários pares de colchetes, um para cada dimensão. Você também usa essa sintaxe para acessar elementos em várias posições dentro do array. Aqui está um exemplo de array multidimensional que representa um tabuleiro de xadrez hipotético:

```
ChessPiece[][] tabuleiro de xadrez;

tabuleiro de xadrez = nova Peça de Xadrez[8][8];

tabuleiro de xadrez[0][0] = novo ChessPiece.Rook;

tabuleiro de xadrez[1][0] = novo ChessPiece.Pawn;

chessBoard[0][1] = novo ChessPiece.Knight;

//configura as peças restantes
```

Figura 4-4ilustra a matriz de matrizes que criamos.

Figura 4-4. Uma série de matrizes de peças de xadrez

Aqui, chessBoard é declarado como uma variável do tipo ChessPiece[][] (um array de arrays ChessPiece). Esta declaração cria implicitamente o tipo ChessPiece[] também. O

exemplo ilustra a forma especial do operador new usado para criar uma matriz multidimensional. Ele cria um array de objetos ChessPiece[] e então, por sua vez, transforma cada elemento em um array de objetos ChessPiece. Em seguida, indexamos chessBoard para especificar valores para elementos ChessPiece específicos.

Claro, você pode criar arrays com mais de duas dimensões. Aqui está um exemplo um pouco impraticável:

Cor [][][] rgb = nova Cor [256][256][256];

rgb[0][0][0] = Cor.PRETO;

rgb[255][255][0] = Cor.AMARELO;

rgb[128][128][128] = Cor.CINZA;

// Faltam apenas 16 milhões!

Podemos especificar um índice parcial de um array multidimensional para obter um subarray de objetos do tipo array com menos dimensões. Em nosso exemplo, a variável chessBoard é do tipo ChessPiece[][]. A expressão chessBoard[0] é válida e refere-se ao primeiro elemento do chessBoard, que, em Java, é do tipo ChessPiece[].

Por exemplo, podemos preencher nosso tabuleiro de xadrez uma linha por vez: Peça de Xadrez[] homeRow = {

nova Peça de Xadrez("Torre"), nova Peça de Xadrez("Cavaleiro"), nova Peça de Xadrez("Bispo"), nova Peça de Xadrez("Rei"), nova Peça de Xadrez("Rainha"), nova Peça de Xadrez("Bispo"), nova Peça de Xadrez("Cavaleiro"), nova Peça de Xadrez("Torre")

};

tabuleiro de xadrez[0] = homeRow;

Não precisamos necessariamente especificar os tamanhos das dimensões de um array multidimensional

com uma única nova operação. A sintaxe do operador new permite deixar os tamanhos de algumas dimensões não especificados. O tamanho de pelo menos a primeira dimensão (a dimensão mais significativa da matriz) deve ser especificado, mas os tamanhos de qualquer número de dimensões finais da matriz menos significativas podem ser deixados indefinidos. Podemos atribuir valores apropriados do tipo array posteriormente.

Podemos criar um quadro simplificado de valores booleanos que poderia hipoteticamente rastrear o status ocupado de um determinado quadrado usando esta técnica:

booleano [][] checkerBoard = novo booleano [8][];

Aqui, checkerBoard é declarado e criado, mas seus elementos, os oito objetos boolean[] do próximo nível, são deixados vazios. Com este tipo de inicialização, checkerBoard[0] é nulo até que criemos explicitamente um array e o atribuamos, como segue:

checkerBoard[0] = novo booleano [8];

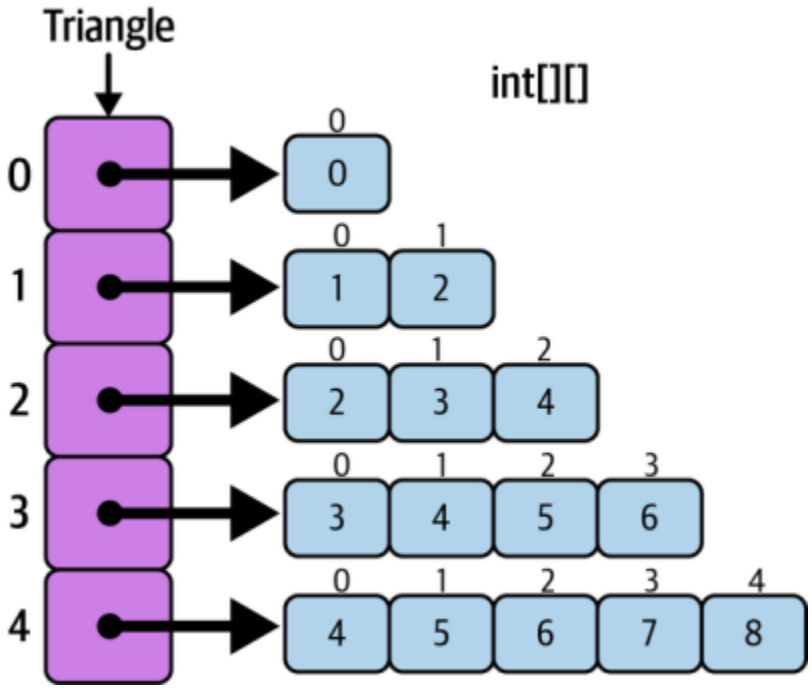
checkerBoard[1] = novo booleano [8];

// ...

checkerBoard[7] = novo booleano [8];

O código dos dois trechos anteriores é equivalente a:

booleano [][] checkerBoard = novo booleano [8][8];



Um motivo pelo qual você pode querer deixar as dimensões de um array não especificadas é para que você possa armazenar os arrays que nos forem fornecidos posteriormente.

Observe que, como o comprimento do array não faz parte de seu tipo, os arrays no tabuleiro de damas não precisam necessariamente ter o mesmo comprimento; isto é, matrizes multidimensionais não precisam ser retangulares. Considere a matriz

“triangular” de inteiros mostrada emFigura 4-5onde a linha um possui uma coluna, a linha dois possui duas colunas e assim por diante.

Figura 4-5. Uma matriz triangular de matrizes

Os exercícios no final do capítulo lhe dão a oportunidade de configurar esse array e inicializá-lo você mesmo!

Tipos, classes e matrizes, meu Deus!

Java possui uma ampla variedade de tipos de armazenamento de informações, cada um com sua própria maneira de representar bits literais dessas informações. Com o tempo, você ganhará familiaridade e conforto com ints, doubles, chars e Strings. Mas não se apresse – esses blocos de construção fundamentais são exatamente o tipo de coisa que o jshell foi projetado para ajudá-lo a explorar. Sempre vale a pena verificar sua compreensão do que uma variável pode armazenar. Os arrays, em particular, podem se beneficiar com um pouco de experimentação. Você pode experimentar as diferentes técnicas de declaração e confirmar se sabe como acessar os elementos individuais dentro de estruturas unidimensionais e multidimensionais.

Você também pode brincar com instruções simples de fluxo de controle no jshell, como nossas instruções if branching e while looping. É necessário um pouco de paciência para digitar trechos de múltiplas linhas ocasionais, mas não podemos exagerar o quão úteis são o jogo e a prática à medida que você carrega mais e mais detalhes de Java em seu cérebro. As linguagens de programação certamente não são tão complexas quanto as linguagens humanas, mas ainda têm muitas semelhanças.

Você pode aprender Java da mesma forma que aprendeu em inglês (ou no idioma que está usando para ler este livro, se tiver uma tradução). Você começará a ter uma ideia do que o código deve fazer, mesmo que não entenda imediatamente os detalhes.

Algumas partes de Java, como arrays, são definitivamente cheias de detalhes.

Observamos anteriormente que arrays são instâncias de classes especiais de array na linguagem Java. Se os arrays possuem classes, onde eles se enquadram na hierarquia de classes e como estão relacionados? Estas são boas perguntas, mas precisamos falar mais sobre os aspectos orientados a objetos do Java antes de respondê-las. Esse é o assuntocapítulo 5. Por enquanto, acredite que os arrays se enquadram na hierarquia de classes.

Perguntas de revisão

1. Qual formato de codificação de texto é usado por padrão pelo Java em classes compiladas?

2. Quais caracteres são usados para incluir um comentário de múltiplas linhas?

Esses comentários podem ser aninhados?

3. Quais construções de loop o Java suporta?

4. Em uma cadeia de testes if/else if, o que acontece se múltiplas condições forem verdadeiras?

5. Se você quisesse armazenar a capitalização total do mercado de ações dos EUA (cerca de US$ 31 trilhões no fechamento do ano fiscal de 2022) como dólares inteiros, que tipo de dados primitivos você poderia usar?

6. Qual é o valor da expressão 18 - 7 * 2?

7. Como você criaria um array para armazenar os nomes dos dias da semana?

Exercícios de código

Para sua prática de codificação, usaremos dois dos exemplos deste capítulo: 1. Implemente o algoritmo GCD de Euclides como uma classe completa chamada Euclides. Lembre-se dos fundamentos do algoritmo:

intuma = 2701;

interno b = 222;

enquanto (b! = 0) {

se (a > b) {

uma = uma - b;

} outro {

b=b - uma;

}

}

System.out.println("GCD é " + a);

Para sua saída, você consegue pensar em uma maneira de mostrar ao usuário os valores originais de aeb além do denominador comum? A saída ideal seria algo assim:

% java Euclides

O MDC de 2701 e 222 é 37

2. Tente criar o array triangular da seção anterior em uma classe simples ou em jshell. Aqui está uma maneira:

int[][] triângulo = novo int[5][];

```
for (int i = 0; i < triângulo.comprimento; i++) {

triângulo[i] = novo int [i + 1];

para (int j = 0; j < i + 1; j++)

triângulo[i][j] = i + j;

}
```

Agora expanda esse código para imprimir o conteúdo do triângulo na tela. Para ajudar, lembre-se que você pode imprimir o valor de um elemento do array com o método System.out.println():

```
System.out.println(triângulo[3][1]);
```

Sua saída provavelmente será uma longa linha vertical de números como esta: 0

1

2

2

3

4

3

4

5

6

4

5

6

7

8

Exercícios Avançados

1. Se você estiver disposto a enfrentar um desafio maior, tente organizar a saída em um triângulo visual. A instrução acima imprime um elemento sozinho em uma linha. O objeto System.out integrado possui outro método de saída: print(). Este método não imprime uma nova linha depois de imprimir qualquer argumento que foi passado. Você pode encadear várias chamadas

System.out.print() para produzir uma linha de saída:

System.out.print("Olá");

System.out.print(" ");

System.out.print("triângulo!");

System.out.println(); // Queremos completar a linha

// Saída:

// Olá triângulo!

Sua saída final deve ser semelhante a esta:

% Java Triângulo

0

1 2

2 3 4

3 4 5 6

4 5 6 7 8

1Confira o oficialSite UnicodePara maiores informações. Curiosamente, um dos scripts listados como “obsoletos e arcaicos” e atualmente não suportado pelo padrão Unicode é o javanês – uma língua histórica do povo da ilha indonésia de Java.

2Usar um comentário para “ocultar” o código pode ser mais seguro do que simplesmente excluir o código. Se você quiser o código de volta, basta retirar o(s) delimitador(es) de comentário.

3Java usa uma técnica chamada “complemento de dois” para armazenar inteiros. Esta técnica usa um bit no início do número para determinar se é um valor positivo ou negativo. Uma peculiaridade dessa técnica é que o intervalo negativo é sempre maior em um.

4O código comparável em C++ seria:

Carro& meuCarro = *(novo Carro());

Carro& outroCarro = meuCarro;

5Dizemos popular porque muitas linguagens de programação possuem a mesma instrução condicional.

6A palavra “Booleano” vem do matemático inglês George Boole, que lançou as bases para a análise lógica. A

palavra está corretamente capitalizada, mas muitas linguagens de computador têm um tipo “booleano” que usa letras minúsculas – incluindo Java.

Você invariavelmente verá ambas as variações online.

7Não cobriremos outros formulários aqui, mas Java também suporta o uso de tipos de enumeração e correspondência de classes em instruções switch.

8Pular para rótulos nomeados ainda considerado má forma.

9Você deve se lembrar do termo precedência – e seu mnemônico fofo, “Por favor, desculpe minha querida tia Sally” – da álgebra do ensino médio. Java avalia primeiro (p)arênteses, depois quaisquer (e)xpoentes, depois (m)ultiplicação e (d)ivisão e, finalmente, (a)adição e (s)subtração.

10Os computadores representam inteiros de duas maneiras: inteiros com sinal, que permitem números negativos, e sem sinal, que não permitem. Um byte assinado, por exemplo, tem o intervalo -128…127. Um byte não assinado tem o intervalo de 0 a 255.

11O análogo em C ou C++ é uma matriz de ponteiros. No entanto, os ponteiros em C ou C++ são valores de dois, quatro ou oito bytes. Alocar uma matriz de ponteiros é, na verdade, alocar armazenamento para algum número desses valores de ponteiro. Uma matriz de referências é conceitualmente semelhante, embora as referências não sejam objetos. Não podemos manipular referências ou partes de referências a não ser por atribuição, e seus requisitos de armazenamento (ou a falta deles) não fazem parte da especificação da linguagem Java de alto nível.

Capítulo 5. Objetos em Java

Neste capítulo, chegamos ao cerne do Java e exploramos seus aspectos orientados a objetos. O termo design orientado a objetos refere-se à arte de decompor uma aplicação em um certo número de objetos, que são componentes de aplicação independentes que funcionam juntos. O objetivo é dividir seu problema em problemas menores, mais simples e fáceis de manusear e manter. Projetos baseados em objetos provaram seu valor ao longo dos anos, e linguagens orientadas a objetos, como Java, fornecem uma base sólida para escrever aplicativos — desde os muito pequenos até os muito grandes. Java foi projetado desde o início para ser uma linguagem orientada a objetos, e todas as APIs e bibliotecas Java são construídas em torno de padrões de design sólidos baseados em objetos.

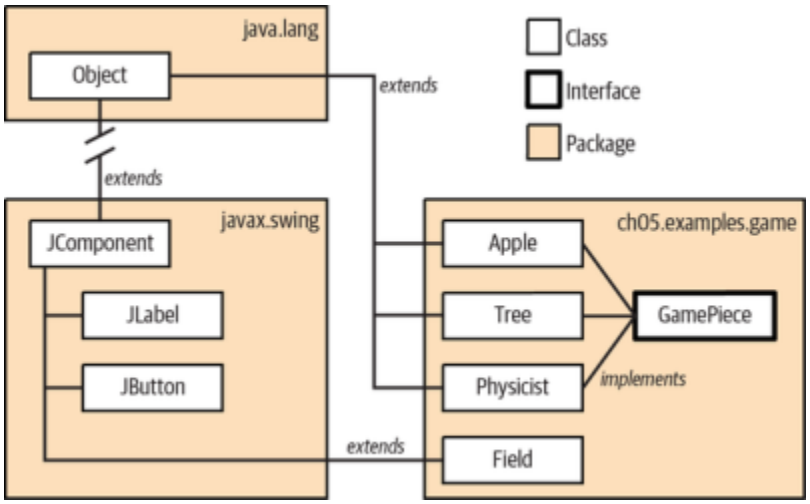
Uma metodologia de design de objetos é um sistema ou conjunto de regras criado para ajudá-lo a dividir seu aplicativo em objetos. Freqüentemente, isso significa mapear entidades e conceitos do mundo real (às vezes chamados de domínio do problema) em componentes do aplicativo. Várias metodologias tentam ajudá-lo a transformar seu aplicativo em um bom conjunto de objetos reutilizáveis. Isto é bom em princípio, mas o problema é que um bom design orientado a objetos ainda é mais arte do que ciência. Embora você possa aprender com metodologias de design disponíveis no mercado, nenhuma delas o ajudará em todas as situações. A verdade é que não há substituto para a experiência.

Não tentaremos empurrá-lo para uma metodologia específica aqui; há prateleiras cheias de livros para fazer

isso.1Em vez disso, forneceremos algumas dicas de bom senso ao longo do caminho, à medida que você começar.

Aulas

As classes são os blocos de construção de um aplicativo Java. Uma classe pode conter métodos (funções), variáveis, código de inicialização e, como discutiremos mais tarde, outras classes. Classes separadas que descrevem partes individuais de uma ideia mais complexa geralmente são agrupadas em pacotes, o que ajuda a organizar projetos maiores. (Cada classe pertence a algum pacote, mesmo os exemplos simples que vimos até agora.) Uma interface pode descrever alguns pontos em comum específicos



entre classes que de outra forma seriam díspares. As classes podem ser relacionadas entre si por extensão ou com interfaces por implementação.Figura 5-1ilustra as ideias neste parágrafo muito denso.

Figura 5-1. Visão geral de classe, interface e pacote

Objeto(no canto superior esquerdo) é a classe fundamental no centro de todas as outras classes em

Java. Faz parte do pacote Java principal, java.lang. Java também possui um pacote para seus elementos gráficos de UI chamado javax.swing. Dentro desse pacote, a classe JComponent define todas as propriedades comuns de baixo nível de coisas gráficas, como quadros, botões e telas. A classe JLabel, por exemplo, estende a classe JComponent. Isso significa que o JLabel herda detalhes do JComponent, mas adiciona coisas específicas aos rótulos. Você deve ter notado que o próprio JComponent se estende de Object ou, pelo menos, eventualmente se estende de volta a Object. Por uma questão de brevidade, deixamos de fora as classes intermediárias e os pacotes intermediários.

Você também pode definir suas próprias classes e pacotes. Por exemplo, o pacote ch05.examples.game no canto inferior direito é um pacote personalizado que construímos para um jogo simples que permite aos físicos jogar maçãs. (Newton terá sua vingança!) Neste pacote temos algumas classes, como Apple e Field, que fazem parte de nossa aplicação. Você também pode ver a interface GamePiece, que conterá alguns elementos comuns e necessários para todas as peças do jogo e é implementada pelas classes Apple, Tree e Physicist. (Em nosso jogo, a classe Field é onde todas as peças do jogo serão mostradas, mas não é uma peça do jogo em si. Observe que ela não implementa a interface GamePiece.)

Este capítulo entrará em muito mais detalhes, com mais exemplos de cada conceito.

Recomendamos vivamente que você experimente os exemplos à medida que avança e use a ferramenta jshell (discutida em“Experimentando Java”) para ajudar a consolidar sua compreensão de novos tópicos.

Declarando e Instanciando Classes

Uma classe serve como modelo para criar instâncias, que são objetos de tempo de execução (cópias individuais) que implementam a estrutura da classe. Você declara uma classe com a palavra-chave class e um nome de sua escolha. No nosso jogo, por exemplo, os físicos, as maçãs e as árvores são bons alvos para se tornarem classes.

Dentro de uma classe, adicionamos variáveis que armazenam detalhes ou outras

informações úteis e métodos que descrevem o que podemos fazer com e com instâncias desta classe.

Vamos começar com uma aula para nossas maçãs. Por convenção (forte!), os nomes das classes começam com letras maiúsculas. Isso faz da Apple um bom nome para usar. Não tentaremos incluir todos os detalhes sobre nossas maçãs do jogo na classe imediatamente, apenas alguns elementos para ajudar a ilustrar como uma classe, variáveis e métodos se encaixam:

```
pacote ch05.examples;

classe Maçã {

massa flutuante;

diâmetro do flutuador = 1,0f;

interno x, y;

boolean isTouching(Apple outro) {

// O código eventualmente irá aqui que executa

// calcula a distância e retorna verdadeiro se
```

```
// esta maçã está tocando outra maçã

}

// Mais métodos aparecerão aqui à medida que
preenchermos mais

// detalhes da nossa maçã

}
```

A classe Apple contém quatro variáveis: massa, diâmetro, x e y. Ele também define um método chamado isTouching(), que usa uma referência a outro Apple como argumento e retorna um valor booleano como resultado. Variáveis e declarações de métodos podem aparecer em qualquer ordem, mas inicializadores de variáveis não podem fazer “referências diretas” para outras variáveis que aparecem

posteriormente. (Em nosso pequeno trecho, a variável diâmetro poderia usar a variável massa para ajudar a calcular seu valor inicial, mas massa não poderia usar a variável diâmetro para fazer o mesmo.) Depois de definirmos a classe Apple, podemos criar um objeto Apple (uma instância dessa classe) para o nosso jogo, assim:

```
// Duas etapas, a declaração e depois a instanciação

maçã a1;

a1 = nova Maçã();

// Ou tudo em uma linha...

Maçã a2 = nova Maçã();
```

Lembre-se de que nossa declaração da variável a1 não cria um objeto Apple; simplesmente cria uma variável que se refere a um objeto do tipo Apple. Ainda temos que criar o objeto, usando a palavra-chave new, conforme mostrado na segunda linha do trecho de código anterior. Mas você pode combinar essas etapas em uma única linha, assim como fizemos para a variável a2. As mesmas ações separadas ocorrem

nos bastidores, é claro. Às vezes, a declaração e a inicialização combinadas parecerão mais legíveis do que a versão multilinhas.

Agora que criamos um objeto Apple, podemos acessar suas variáveis e métodos, como vimos em vários de nossos exemplos deCapítulo 4ou até mesmo nosso aplicativo gráfico “Hello” de“OláJava”. Embora isso não seja muito interessante, poderíamos agora construir outra classe, PrintAppleDetails, que é uma aplicação completa para criar uma instância Apple e imprimir seus detalhes:

```
pacote ch05.examples;

classe pública PrintAppleDetails {

public static void main(String args[]) {

Maçã a1 = nova Maçã();

System.out.println("Apple a1:");

System.out.println(" massa: " + a1.mass);

System.out.println(" diâmetro: " + a1.diâmetro);

System.out.println("posição: (" + a1.x + ", " + a1.y +")");
```

}

}

Se você compilar e executar este exemplo, deverá ver a seguinte saída em seu terminal ou na janela de terminal do seu IDE:

Maçã a1:

massa: 0,0

diâmetro: 1,0

posição: (0, 0)

Mas hmm, por que a1 não tem massa? Se você observar como declaramos as variáveis para nossa classe Apple, inicializamos apenas o diâmetro. Todas as outras variáveis obtêm o valor padrão atribuído por Java de 0, pois são tipos numéricos.2Gostaríamos idealmente de ter uma maçã mais interessante. Vamos ver como fornecer essas partes interessantes.

Acessando Campos e Métodos

Depois de ter uma referência a um objeto, você pode usar e manipular suas variáveis e métodos usando a notação de ponto que viu emCapítulo 4. Vamos criar uma nova classe, PrintAppleDetails2, fornecer alguns valores para a massa e posição da nossa instância a1 e depois imprimir os novos detalhes:

pacote ch05.examples;

classe pública PrintAppleDetails2 {

public static void main(String args[]) {

```
Maçã a1 = nova Maçã();
System.out.println("Apple a1:");
System.out.println(" massa: " + a1.mass);
System.out.println(" diâmetro: " + a1.diâmetro);
System.out.println("posição: (" + a1.x + ", " + a1.y +")");
//preenche algumas informações em a1
a1.massa = 10,0f;
a1.x = 20;
a1.y = 42;
System.out.println("A1 atualizado:");
System.out.println(" massa: " + a1.mass);
System.out.println(" diâmetro: " + a1.diâmetro);
System.out.println("posição: (" + a1.x + ", " + a1.y +")");
}
}
```

Aqui está a nova saída:

```
Maçã a1:
massa: 0,0
diâmetro: 1,0
posição: (0, 0)
```

A1 atualizado:

massa: 10,0

diâmetro: 1,0

posição: (20, 42)

Ótimo! a1 parece um pouco melhor. Mas observe o código novamente. Tivemos que repetir as três linhas que imprimem os detalhes do objeto. Esse tipo de replicação exata exige um método.

Os métodos nos permitem “fazer coisas” dentro de uma classe. Como um exemplo simples, poderíamos melhorar a classe Apple fornecendo estas instruções print em um método:

classe pública Apple {

massa flutuante;

diâmetro do flutuador = 1,0f;

interno x, y;

// outras variáveis e métodos relacionados à Apple

public void printDetails() {

System.out.println("massa:" + massa);

System.out.println(" diâmetro: " + diâmetro);

System.out.println("posição: (" + x + ", " + y +")");

}

```
}
```

Com essas instruções detalhadas realocadas, podemos criar PrintAppleDetails3 que faz seu trabalho de forma mais sucinta que seu antecessor:

```
pacote ch05.examples;

classe pública PrintAppleDetails3 {

public static void main(String args[]) {

Maçã a1 = nova Maçã();

System.out.println("Apple a1:");

// Podemos usar nosso novo método!

a1.printDetails();

//preenche algumas informações em a1

a1.massa = 10,0f;

a1.x = 20;

a1.y = 42;

System.out.println("A1 atualizado:");

// E olhe! Podemos facilmente reutilizar o mesmo método

a1.printDetails();

}

}
```

Dê uma outra olhada no método printDetails() que adicionamos à classe Apple. Dentro de uma classe, podemos acessar variáveis e chamar métodos da classe diretamente pelo nome. As instruções de impressão usam apenas nomes simples como massa e diâmetro.

Ou considere preencher o método isTouching(). Podemos usar nossas próprias coordenadas xey sem nenhum prefixo especial. Mas para aceder às coordenadas de alguma outra maçã, precisamos de voltar à notação de ponto. Aqui está uma maneira de escrever esse método usando um pouco de matemática (mais sobre isso em"A classe java.lang.Math") e a instrução if/then que vimos em"condicionais if/else":

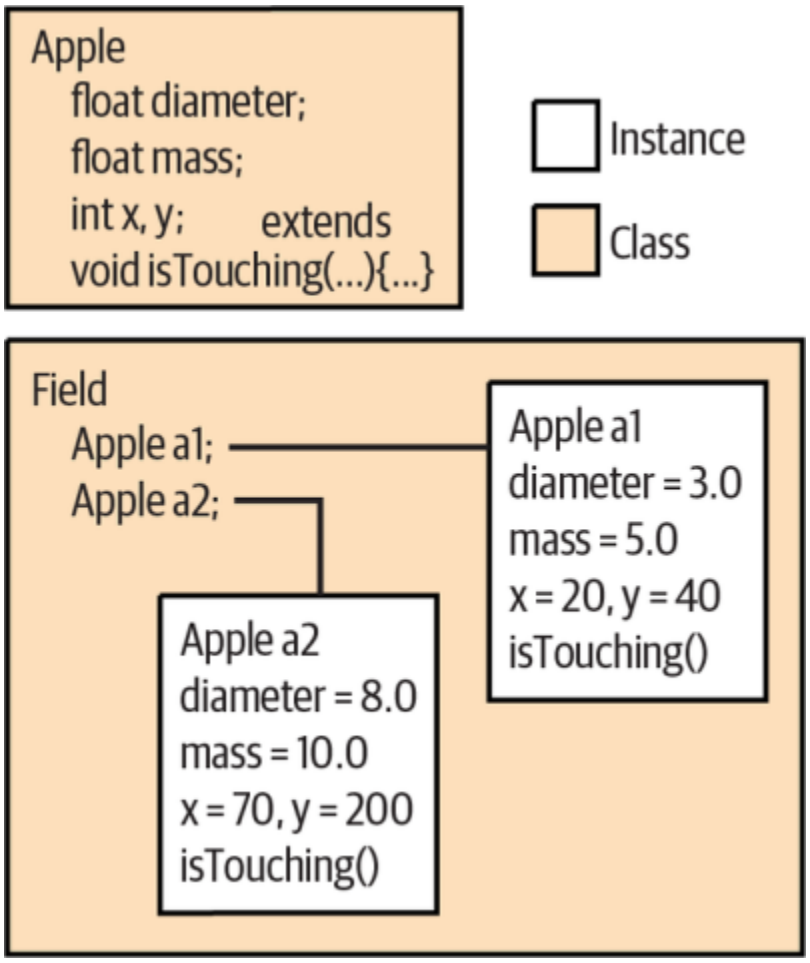
```
// Arquivo: ch05/examples/Apple.java

public boolean isTouching(Apple outro) {

duplo xdiff = x - outro.x;

duplo ydiff = y - outro.y;

distância dupla = Math.sqrt(xdiff * xdiff + ydiff * ydiff);

if (distância < diâmetro / 2 + outro.diâmetro / 2) {

retornar verdadeiro;

} outro {

retorna falso;

}

}
```

Vamos preencher um pouco mais nosso jogo e criar uma classe Field que usa alguns objetos Apple. Ele cria instâncias como variáveis-membro e trabalha com esses objetos nos métodos setupApples() e detectCollision(), invocando métodos Apple e acessando variáveis desses objetos através das referências a1 e a2, visualizadas

emFigura 5-2:

pacote ch05.examples;

classe pública Campo {



Maçã a1 = nova Maçã();

```
Maçã a2 = nova Maçã();
public void setupApples() {
a1.diâmetro = 3,0f;
a1.massa = 5,0f;
a1.x = 20;
a1.y = 40;
a2.diâmetro = 8,0f;
a2.massa = 10,0f;
a2.x = 70;
a2.y = 200;
}
public void detectCollisions() {
if (a1.isTouching(a2)) {
System.out.println("Colisão detectada!");
} outro {
System.out.println("As maçãs não estão se tocando.");
}
}
}
```

Figura 5-2. Instâncias da classe Apple

Podemos provar que Field tem acesso às variáveis e métodos das maçãs com outra iteração de nossa aplicação, PrintAppleDetails4:

```
pacote ch05.examples;

classe pública PrintAppleDetails4 {

public static void main(String args[]) {

Campo f = new Campo();

f.setupApples();

System.out.println("Apple a1:");

f.a1.printDetails();

System.out.println("Apple a2:");

f.a2.printDetails();

f.detectCollisions();

}

}
```

Devemos ver os detalhes familiares da maçã seguidos de uma resposta sobre se as duas maçãs estão ou não se tocando:

```
% java PrintAppleDetails4

Maçã a1:

massa: 5,0
```

diâmetro: 3,0

posição: (20, 40)

Maçã a2:

massa: 10,0

diâmetro: 8,0

posição: (70, 200)

As maçãs não estão se tocando.

Ótimo, exatamente o que esperávamos.

Antes de continuar lendo, experimente mudar a posição das maçãs para que elas se toquem. Você obtém o resultado esperado?

Pré-visualização dos modificadores de acesso

Vários fatores afetam se os membros da classe podem ser acessados de outra classe.

Você pode usar os modificadores de visibilidade public, private e protected para controlar o acesso; classes também podem ser colocadas em um pacote, o que afeta seu escopo. O modificador private, por exemplo, designa uma variável ou método para uso apenas por outros membros da própria classe. No exemplo anterior, poderíamos alterar a declaração da nossa variável diâmetro para privada:

classe Maçã {

// ...

diâmetro do flutuador privado;

// ...

}

Agora não podemos acessar o diâmetro do Campo:

classe Campo {

Maçã a1 = nova Maçã();

Maçã a2 = nova Maçã();

// ...

void setupMaçãs() {

a1.diâmetro = 3,0f; //Erro em tempo de compilação

// ...

a2.diâmetro = 8,0f; //Erro em tempo de compilação

// ...

}

// ...

}

Se ainda precisarmos acessar o diâmetro de alguma forma, normalmente

adicionaríamos os métodos public getDiameter() e setDiameter() à classe Apple: classe pública Apple {

diâmetro do flutuador privado = 1,0f;

```
// ...

public void setDiameter(float newDiameter) {

diâmetro = novoDiâmetro;

}

public float getDiameter() {

diâmetro de retorno;

}

// ...

}
```

Criar métodos como este é uma boa regra de design porque permite flexibilidade futura na alteração do tipo ou comportamento do valor. Veremos mais sobre pacotes, modificadores de acesso e como eles afetam a visibilidade de variáveis e métodos posteriormente neste capítulo.

Membros estáticos

Como dissemos, variáveis e métodos de instância são associados e acessados por meio de uma instância da classe (ou seja, por meio de um objeto específico, como a1 ou f nos exemplos anteriores). Por outro lado, os membros declarados com o modificador estático residem na classe e são compartilhados por todas as instâncias da classe.

Variáveis declaradas com o modificador estático são chamadas de variáveis estáticas ou variáveis de classe; da mesma forma, esses tipos de métodos são chamados

de métodos estáticos ou métodos de classe. Membros estáticos são úteis como sinalizadores e identificadores, que podem ser acessados de qualquer lugar. Podemos adicionar uma variável estática ao nosso exemplo da Apple para armazenar o valor da aceleração da gravidade. Isso nos permite calcular a trajetória de uma maçã lançada quando começamos a animar nosso jogo:

classe Maçã {

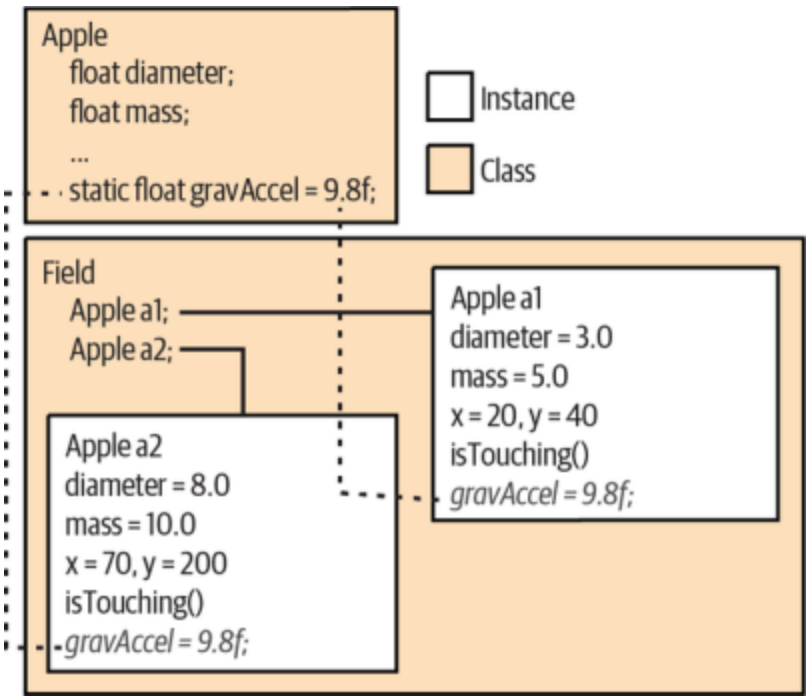
// ...

flutuador estático gravAccel = 9,8f;

// ...

}

Declaramos a nova variável flutuante gravAccel como estática. Isso significa que ele está associado à classe, não a uma instância individual, e se alterarmos seu valor



(diretamente ou por meio de qualquer instância da Apple), o valor muda para todos os objetos Apple, conforme mostrado emFigura 5-3.

Figura 5-3. Variáveis estáticas compartilhadas por todas as instâncias de uma classe Você pode acessar membros estáticos da mesma forma que acessa membros de instância. Dentro da nossa classe Apple, podemos nos referir a gravAccel como qualquer outra variável:

```
classe Maçã {

// ...

float getPeso() {

massa de retorno * gravAccel;

}

// ...

}
```

Porém, como existem membros estáticos na própria classe, independentemente de qualquer instância, também podemos acessá-los diretamente através da classe. Se quisermos jogar maçãs em Marte, por exemplo, não precisamos de um objeto Apple como a1 ou a2 para obter ou definir a variável gravAccel. Em vez disso, podemos usar a classe para alterar a variável para refletir as condições em Marte: Apple.gravAccel = 3,7f;

Isso altera o valor de gravAccel para a classe e todas as suas instâncias. Não precisamos configurar manualmente cada instância da Apple para cair em Marte.

Variáveis estáticas são úteis para qualquer tipo de dados compartilhados entre classes em tempo de execução. Por exemplo, você pode criar métodos para registrar suas instâncias de objetos para que elas possam se comunicar ou para que você possa acompanhar todas elas. Também é comum usar variáveis estáticas para definir valores constantes. Neste caso, usamos o modificador estático junto com o modificador final. Portanto, se nos importássemos apenas com as maçãs sob a influência da atração gravitacional da Terra, poderíamos mudar a Apple da seguinte forma:

```
classe Maçã {

// ...

flutuador final estático EARTH_ACCEL = 9,8f;

// ...

}
```

Seguimos uma convenção comum aqui e nomeamos nossa constante com letras maiúsculas e sublinhados (se o nome tiver mais de uma palavra). O valor de EARTH_ACCEL é uma constante; você pode acessá-lo através da classe Apple ou de suas instâncias, mas não pode alterar seu valor.

É importante usar a combinação de static e final apenas para coisas que são realmente constantes. O compilador pode “incorporar” esses valores nas classes que os referenciam. Isso significa que se você alterar uma variável final estática, poderá ser necessário recompilar todo o código que usa essa classe (esse é realmente o único caso em que você precisa fazer isso em Java). Os membros estáticos também são úteis para valores

necessários na construção da própria instância. Em nosso exemplo, poderíamos declarar vários valores estáticos para representar vários tamanhos de objetos Apple:

```
classe Maçã {

// ...

estático int PEQUENO = 0, MÉDIO = 1, GRANDE = 2;

// ...
```

}

Poderíamos então usar essas opções em um método que defina o tamanho de uma Apple, ou em um construtor especial, como discutiremos em breve:

Apple típicoApple = new Apple();

típicoApple.setSize(Apple.MEDIUM);

Novamente, dentro da classe Apple, também podemos usar membros estáticos diretamente pelo nome. Não há necessidade da Apple. prefixo:

classe Maçã {

// ...

void resetTudo() {

setSize (MÉDIO);

// ...

}

// ...

}

Métodos

Até agora, nossas classes de exemplo foram bastante simples. Mantemos algumas informações por perto – maçãs têm massa, campos têm algumas maçãs, etc. Mas também tocamos na ideia de fazer com que essas classes façam coisas. Todas as nossas diversas classes

PrintAppleDetails possuem uma lista de etapas que são executadas quando executamos o programa, por exemplo. Como observamos

brevemente antes, em Java, essas etapas são agrupadas em um método. No caso de PrintAppleDetails, esse é o método main().

Onde quer que você tenha etapas a seguir ou decisões a tomar, você precisa de um método. Além de armazenar variáveis como massa e diâmetro em nossa classe Apple, também adicionamos alguns trechos de código que continham ações e lógica. Os métodos são tão fundamentais para as aulas que tivemos que criar alguns antes mesmo de chegarmos à discussão formal deles! Pense no método printDetails() na Apple ou no método setupApples() no Field. Mesmo nosso primeiro programa simples exigia um método main().

Esperançosamente, os métodos que discutimos até agora foram simples o suficiente para serem seguidos apenas pelo contexto. Mas os métodos podem fazer muito mais do que imprimir algumas variáveis ou calcular uma distância. Eles podem conter declarações de variáveis locais e outras instruções Java que são executadas quando o método é invocado. Os métodos também podem retornar um valor para o chamador.

Eles sempre especificam um tipo de retorno, que pode ser um tipo primitivo, um tipo de referência ou o void especial, que indica nenhum valor retornado. Os métodos podem receber argumentos, que são valores fornecidos pelo chamador do método.

Aqui está um exemplo simples de um método que aceita argumentos:

```
classe Pássaro {

int xPos, yPos;

mosca dupla (int x, int y) {

distância dupla = Math.sqrt(x*x + y*y);

aba(distância);

xPos = x;

yPos = y;

distância de retorno;

}

// outras coisas de pássaros...

}
```

Neste exemplo, a classe Bird define um método, fly(), que toma como argumentos dois inteiros: x e y. Como resultado, ele retorna um valor do tipo double, usando a palavra-chave return.

Nosso método possui um número fixo de argumentos (dois); entretanto, os métodos podem ter listas de argumentos de comprimento variável, o que permite ao método especificar que pode receber qualquer número de argumentos e classificá-los sozinho em tempo de execução.3

Variáveis locais

Nosso método fly() declara uma variável local chamada distância, que é usada para calcular a distância voada.

Uma variável local é temporária; ele existe apenas dentro do escopo (o bloco) do seu método. Variáveis locais são alocadas quando um método é

invocado; eles normalmente são destruídos quando o método retorna. Eles não podem ser referenciados fora do próprio método. Se o método estiver sendo executado simultaneamente em threads diferentes, cada thread terá sua própria versão das variáveis locais do método. Os argumentos de um método também servem como variáveis locais dentro do escopo do método; a única diferença é que eles são inicializados ao serem passados pelo chamador do método.

Um objeto criado dentro de um método e atribuído a uma variável local pode ou não persistir após o retorno do método. Como veremos em detalhes em“Destruição de

Objetos”, depende se alguma referência ao objeto permanecerá. Se um objeto for criado, atribuído a uma variável local e nunca usado em nenhum outro lugar, esse objeto não será mais referenciado quando a variável local desaparecer do escopo, então a coleta de lixo removerá o objeto. Se, no entanto, atribuirmos o objeto a uma variável de instância de um objeto, passá-lo como argumento para outro método ou devolvê-lo como um valor de retorno, ele poderá ser salvo por outra variável que contém sua referência.

Sombreamento

Se uma variável local ou argumento de método e uma variável de instância tiverem o mesmo nome, a variável local obscurece ou oculta o nome da variável de instância dentro do escopo do método. Isto pode parecer uma situação estranha, mas acontece com bastante

frequência quando a variável de instância tem um nome comum ou óbvio. Por exemplo, poderíamos adicionar um método move à nossa classe Apple.

Nosso método precisará de uma nova coordenada informando onde colocar a maçã.

Uma escolha fácil para os argumentos de coordenadas seria x e y. Mas já temos variáveis de instância com o mesmo nome que mantêm a posição atual da maçã:
classe Maçã {

interno x, y;

public void moveTo(int x, int y) {

System.out.println("Movendo maçã para " + x + ", " + y);

// a lógica de movimento real iria aqui ...

}

}

Se a maçã estiver atualmente na posição (20, 40) e você chamar moveTo(40, 50), o que você acha que a instrução println() mostrará? Dentro de moveTo(), os nomes xey referem-se apenas aos argumentos do método com esses nomes. A saída seria: Movendo a maçã para 40, 50

Se não conseguirmos chegar às variáveis de instância x e y, como podemos mover a maçã? Acontece que Java entende o sombreamento e fornece um mecanismo para contornar essas situações.

A referência “isto”

Você pode usar a referência especial this sempre que precisar se referir explicitamente ao objeto atual ou a um membro do objeto atual. Muitas vezes você não precisa usar isso, porque a referência ao objeto atual está implícita; esse é o caso ao usar variáveis de instância nomeadas de forma inequívoca dentro de uma classe. Mas você pode usar isso para se referir explicitamente a variáveis de instância em um objeto, mesmo que elas estejam sombreadas. O exemplo a seguir mostra como usar isso para permitir nomes de argumentos que ocultam nomes de variáveis de instância.

Esta é uma técnica bastante comum porque evita a necessidade de inventar nomes alternativos. Veja como poderíamos implementar nosso método moveTo() com variáveis sombreadas:

```
classe Maçã {

interno x, y;

diâmetro do flutuador = 1,0f;

public void moveTo(int x, int y) {

System.out.println("Movendo maçã para " + x + ", " + y);

//armazena o novo valor de x

isto.x = x;

//armazena o novo valor de y se for alto o suficiente

se (y > diâmetro) {

isto.y = y;

} outro {
```

```
// caso contrário, defina y para a altura da maçã

this.y = (int)diâmetro;

}

}

}
```

Neste exemplo, a expressão this.x refere-se à variável de instância x e atribui a ela o valor da variável local x, que de outra forma ocultaria seu nome. Fazemos o mesmo para isso, mas adicionamos um pouco de proteção para garantir que não moveremos a maçã para baixo do solo. Observe que o diâmetro não está sombreado neste método.

Como não temos um argumento de diâmetro em moveTo(), não precisamos dizer this.diameter quando o usamos.

A única razão pela qual precisamos usar isso no exemplo anterior é porque usamos nomes de argumentos que ocultam nossas variáveis de instância e queremos fazer referência às variáveis de instância. Você também pode usar a referência this sempre que quiser passar uma referência ao objeto envolvente “atual” para algum outro método, como fizemos para a versão gráfica de nosso aplicativo “Hello Java”

em“HelloJava2: A Sequela”.

Métodos estáticos

Os métodos estáticos (às vezes chamados de métodos de classe), assim como as variáveis estáticas, pertencem

à classe e não a instâncias individuais da classe. O que

isto significa? Bem, acima de tudo, um método estático vive fora de qualquer instância específica. Ele pode ser invocado através do nome da classe e do operador ponto, sem nenhum objeto por perto. Como não está vinculado a um objeto específico, um método estático só pode acessar outros membros estáticos (variáveis estáticas e outros métodos estáticos) da classe. Ele não pode ver diretamente nenhuma variável de instância ou chamar nenhum método de instância, porque para fazer isso seria necessário perguntar “em qual instância?” Métodos estáticos podem ser chamados a partir de instâncias usando a mesma sintaxe dos métodos de instância, mas o importante é que eles também podem ser usados de forma independente.

Nosso método isTouching() usa um método estático, Math.sqrt(), que é definido pela classe java.lang.Math; exploraremos essa classe em detalhes emCapítulo 8. Por

enquanto, o importante a notar é que Math é o nome de uma classe e não uma instância de um objeto Math.4Como os métodos estáticos podem ser invocados sempre que o nome da classe estiver disponível, os métodos de classe estão mais próximos das funções do estilo C. Os métodos estáticos são particularmente úteis para métodos utilitários que realizam trabalhos úteis independentemente das instâncias ou no trabalho em instâncias. Por exemplo, em nossa classe Apple, poderíamos enumerar todos os tamanhos disponíveis como strings legíveis por humanos a partir das constantes que criamos em“Acessando Campos e Métodos”:

classe Maçã {

```
público estático final int PEQUENO = 0;

público estático final int MÉDIO = 1;

público estático final int LARGE = 2;

// outras coisas da maçã...

public static String[] getAppleSizes() {

// Retorna nomes para nossas constantes

return new String[] { "PEQUENO", "MÉDIO", "GRANDE" };

}

}
```

Aqui, definimos um método estático, getAppleSizes(), que retorna um array de strings contendo nomes de tamanho de maçã. Tornamos o método estático porque a lista de tamanhos é a mesma, independentemente do tamanho de qualquer instância da Apple. Ainda podemos usar getAppleSizes() de dentro de uma instância da Apple se quisermos, assim como um método de instância. Poderíamos alterar o método printDetails (não estático) para imprimir um nome de tamanho em vez de um diâmetro exato, por exemplo:

```
public void printDetails() {

System.out.println("massa:" + massa);

// Imprime o diâmetro exato:

//System.out.println(" diâmetro: " + diâmetro);

// Ou um valor aproximado agradável e amigável
```

```
String niceNames[] = getAppleSizes();

if (diâmetro <5,0f) {

System.out.println(niceNomes[PEQUENO]);

} else if (diâmetro <10,0f) {

System.out.println(niceNomes[MÉDIO]);

} outro {

System.out.println(niceNames[LARGE]);

}

System.out.println("posição: (" + x + ", " + y +")");

}
```

Entretanto, também podemos chamá-lo de outras classes, usando o nome da classe Apple com a notação de ponto. Por exemplo, a primeira classe PrintAppleDetails poderia usar uma lógica semelhante para imprimir uma instrução resumida usando nosso método estático e variáveis estáticas, assim:

```
classe pública PrintAppleDetails {

public static void main(String args[]) {

String niceNames[] = Apple.getAppleSizes();

Maçã a1 = nova Maçã();

System.out.println("Apple a1:");

System.out.println(" massa: " + a1.mass);
```

```
System.out.println(" diâmetro: " + a1.diâmetro);

System.out.println("posição: (" + a1.x + ", " + a1.y +")");
if (a1.diâmetro <5,0f) {

System.out.println("Este é um " +
niceNames[Apple.SMALL] + "maçã.");

} senão if (a1.diâmetro < 10,0f) {

System.out.println("Este é um " +
niceNames[Apple.MEDIUM] + "maçã.");

} outro {

System.out.println("Este é um " +
niceNames[Apple.LARGE] + "maçã.");

}

}

}
```

Aqui temos nossa instância confiável da classe Apple, a1, mas não precisamos de a1

para obter a lista de tamanhos. Observe que carregamos a lista de nomes legais antes mesmo de a1 existir. Mas tudo ainda funciona, como pode ser visto na saída: Maçã a1:

massa: 0,0

diâmetro: 1,0

posição: (0, 0)

Esta é uma maçã PEQUENA.

Os métodos estáticos também desempenham um papel importante em vários padrões de design, onde você limita o uso do operador new para uma classe a um método —

um método estático chamado método de fábrica. Falaremos mais sobre construção de objetos em“Construtores”. Não há convenção de nomenclatura para métodos de fábrica, mas é comum ver usos como este:

Apple bigApple = Apple.createApple(Apple.LARGE);

Não escreveremos nenhum método de fábrica, mas é provável que você os encontre por aí, especialmente ao pesquisar perguntas em sites como o Stack Overflow.

Inicializando Variáveis Locais

Ao contrário das variáveis de instância, que recebem valores padrão se não fornecermos um valor explícito, as variáveis locais devem ser inicializadas antes de poderem ser usadas. Você receberá um erro em tempo de compilação se tentar acessar uma variável local sem primeiro atribuir um valor a ela:

```
// variáveis de instância sempre obtêm valores padrão se

// você não os inicializa

int foo;

void meuMetodo() {

//variáveis locais não recebem valores padrão
```

barra interna;

foo += 1; // Tudo bem, foo tem o valor 0

barra += 1; //erro em tempo de compilação, bar não foi inicializado

barra = 99; // Tudo bem, estamos definindo o valor inicial da barra

barra += 1; //Agora esse cálculo está ok

}

Observe que isso não implica que você sempre precise inicializar variáveis locais ao declará-las, apenas que você deve atribuir algum valor a elas antes da primeira vez que você as referencie. Possibilidades mais sutis surgem ao fazer atribuições dentro de condicionais:

void meuMetodo {

barra interna;

if (alguma condição) {

barra = 42;

}

barra += 1; // Ainda é um erro em tempo de compilação, a barra pode não s er inicializada

}

Neste exemplo, bar é inicializado somente se someCondition for verdadeiro. O

compilador não permite que você faça essa aposta, então sinaliza o uso de bar após nossa instrução if como um erro.

Poderíamos corrigir esta situação de várias maneiras. Poderíamos inicializar a variável com um valor padrão antecipadamente ou mover o uso dentro da condicional. Também poderíamos garantir que o fluxo de controle não alcance a variável não inicializada por outros meios, dependendo do que faz sentido para nossa aplicação específica. Por exemplo, poderíamos simplesmente garantir que atribuímos

a bar um valor em ambos os ramos else se someCondition for false. Ou poderíamos retornar do método abruptamente:

```
void meuMetodo {

barra interna;

if (alguma condição) {

barra = 42;

} outro {

retornar;

}

barra += 1; // Está tudo bem!

}
```

Nesse caso, someCondition é verdadeira e bar está definida como 42, ou é falsa e o controle retorna de myMethod(). Não há chance de atingir bar em um estado

não inicializado, então o compilador permite esse uso de bar após a condicional.

Por que Java é tão exigente com variáveis locais? Uma das fontes de erros mais comuns (e insidiosas) em outras linguagens, como C ou C++, é esquecer de inicializar variáveis locais. Variáveis locais nessas linguagens começam com valores aparentemente aleatórios e causam todo tipo de frustração para o programador. Java tenta ajudar e força você a atribuir valores bons e conhecidos.

Passagem de argumentos e referências

No início deCapítulo 4, descrevemos a distinção entre tipos primitivos, que são passados por valor (por cópia), e objetos, que são passados por referência. Agora que você conhece melhor os métodos em Java, vamos ver um exemplo:

```
//declara um método com alguns argumentos

void meuMetodo(int num, SomeKindOfObject o) {

// faz algumas coisas úteis com num e o

}

//usa o método

int eu = 0;

SomeKindOfObject obj = new SomeKindOfObject();

meuMetodo(i, obj);
```

Este pedaço de código chama myMethod(), passando dois argumentos. O primeiro argumento, i, é passado por

valor; quando o método é chamado, o valor de i é copiado no primeiro parâmetro do método (uma variável local) denominado num. Se myMethod() alterar o valor de num, ele alterará apenas sua variável local. Nosso eu não serei afetado.

Da mesma forma, Java coloca uma cópia da referência a obj no argumento o de myMethod(). Mas como é uma referência, obj e o referem-se ao mesmo objeto.

Quaisquer alterações feitas por meio de o ou obj afetam a instância real do objeto. Se alterarmos o valor de, digamos, o.size, a alteração será visível tanto como o.size (dentro de myMethod()) quanto como obj.size (no chamador após a conclusão de

myMethod()). No entanto, se myMethod() reatribuir a referência o para apontar para um objeto diferente, essa atribuição afetará apenas a referência da variável local.

Atribuir o a outra coisa não afeta a variável obj do chamador, que ainda se refere ao objeto original.

Passar referências a métodos nos dá uma ideia do outro uso da palavra-chave this que mencionamos anteriormente. Você pode usar isso para passar uma referência do objeto atual para algum outro objeto. Vejamos um pouco de código para ver como isso (sem trocadilhos) funciona:

elemento de classe {

número interno;

peso duplo;

void printMyDetails() {

```
System.out.println(este);

}

}
```

Nosso exemplo é artificial, é claro, mas a sintaxe está correta. Dentro do método printMyDetails(), chamamos nosso velho amigo, System.out.println(). O argumento que passamos para println() é este, o que significa que queremos que o objeto do elemento atual seja impresso. Trabalharemos com relacionamentos de objetos mais complexos em capítulos posteriores e frequentemente precisaremos de acesso à instância atual. A palavra-chave this nos dá esse acesso.

Wrappers para tipos primitivos

Como abordamos brevemente em“Tipos Primitivos”, há uma divisão no mundo Java entre tipos de classe (objetos) e tipos primitivos (números, caracteres e valores booleanos). Java aceita essa compensação por motivos de eficiência. Ao processar números, você deseja que seus cálculos sejam leves; ter que usar objetos para tipos primitivos complica as otimizações de desempenho. Não é comum, mas às vezes é necessário armazenar um valor primitivo como um objeto. Para essas ocasiões, Java fornece uma classe wrapper padrão para cada um dos tipos primitivos, conforme mostrado emTabela 5-1.

Tabela 5-1. Wrappers de tipo primitivo

Primitivo

Embrulho

vazio

java.lang.Void

boleano

java.lang.Boolean

Caracteres java.lang.Character

byte

java.lang.Byte

curto

java.lang.Short

Primitivo

Embrulho

interno

java.lang.Integer

longo

java.lang.Long

flutuador

java.lang.Float

dobro

java.lang.Double

Uma instância de uma classe wrapper encapsula um único valor de seu tipo correspondente. É um objeto imutável que serve como um contêiner para armazenar o

valor e permite recuperá-lo mais tarde. Você pode construir um objeto wrapper a partir de um valor primitivo ou de uma representação String do valor. As seguintes declarações são equivalentes:

Flutuador pi = novo Flutuador(3.14);

Flutuador pi = novo Flutuador("3.14");

Os construtores do wrapper numérico lançam uma NumberFormatException quando encontram um erro ao analisar uma string.

Cada um dos wrappers numéricos implementa a interface java.lang.Number, que fornece métodos de “valor” para acessar seu valor em todas as formas primitivas.

Você pode recuperar valores escalares com os métodos doubleValue(), floatValue(), longValue(), intValue(), shortValue() e byteValue():

Tamanho duplo = novo Duplo (32,76);

duplo d = size.doubleValue(); //32,76

float f = size.floatValue(); //32.76f

longo l = size.longValue(); //32L

int i = size.intValue(); //32

Este código é equivalente a converter o valor duplo primitivo para os vários tipos.

A necessidade mais comum desses wrappers é quando você deseja passar um valor primitivo para um método que requer um objeto. Por exemplo, emCapítulo 7,

veremos coleções Java, um conjunto sofisticado de classes para lidar com grupos de objetos, como listas, conjuntos e mapas. As coleções funcionam apenas com tipos de objetos, portanto, os primitivos devem ser agrupados quando armazenados nelas. Como veremos na próxima seção, Java torna esse processo de empacotamento transparente e automático. Por enquanto, porém, vamos fazer isso nós mesmos. Como veremos, uma Lista é uma coleção extensível de Objetos. Podemos usar wrappers para armazenar números simples em uma Lista (junto com outros objetos):

```
//Envolvendo manualmente um inteiro

Lista meusNúmeros = new ArrayList();

Inteiro trintaTrês = novo Inteiro(33);

meusNúmeros.add(trinta e três);
```

Aqui, criamos um objeto wrapper Integer para que possamos inserir o número na Lista, usando o método add(), que aceita um objeto. Posteriormente, quando estivermos extraindo elementos da Lista, podemos recuperar o valor int da seguinte forma:

```
// Desembrulhando manualmente um inteiro

Inteiro oNúmero = (Inteiro)meusNúmeros.get(0);

int n = theNumber.intValue(); //33
```

Felizmente, o Java pode fazer grande parte desse trabalho automaticamente. Java chama o empacotamento e desempacotamento automático de tipos primitivos de autoboxing. Como mencionamos anteriormente, permitir que o Java faça isso por nós

torna o código mais conciso e seguro. O uso da classe wrapper fica oculto para nós pelo compilador, mas ainda está sendo usado internamente. Aqui está mais um exemplo que inclui informações extras de tipo (genéricos no jargão da linguagem de computador) e usa autoboxing:

```
// Usando autoboxing e genéricos

List<Integer> meusNúmeros = new ArrayList<Integer>
();

meusNúmeros.add(33);

int n = meusNúmeros.get(0);
```

Observe que não criamos nenhuma instância explícita do wrapper Integer, embora incluamos essas informações extras de tipo entre colchetes angulares (<Integer>) quando declaramos nossa variável. Veremos mais gen[éricos emCapítulo 7.]

Sobrecarga de método

Sobrecarga de método é a capacidade de definir vários métodos com o mesmo nome em uma classe; quando o método é invocado, o compilador escolhe o método correto com base nos argumentos passados ao método. Isto implica que os métodos sobrecarregados devem ter diferentes números ou tipos de argumentos. [(Em“Métodos]

[de substituição”,] veremos a substituição de métodos, que ocorre quando declaramos métodos com assinaturas idênticas em subclasses.)

A sobrecarga de método (também chamada de polimorfismo ad hoc) é um recurso poderoso e útil. A ideia é criar métodos que atuem da mesma forma em diferentes tipos de argumentos. Isto cria a ilusão de que um único método pode operar em muitos tipos de argumentos. O método print() na classe PrintStream padrão é um bom exemplo de sobrecarga de método em ação. Como você provavelmente já deduziu, você pode imprimir uma representação em string de praticamente qualquer coisa usando esta expressão:

System.out.print(argumento);

A variável out é uma referência a um objeto (um PrintStream) que define nove versões diferentes e “sobrecarregadas” do método print(). As versões aceitam argumentos dos seguintes tipos: Object, String, char[], char, int, long, float, double e boolean:

classe PrintStream{

void imprimir (objeto arg) { ... }

void imprimir (String arg) { ... }

void imprimir(char[] arg) { ... }

// ...

}

Você pode invocar o método print() com qualquer um desses tipos como argumento, e o valor será impresso de maneira apropriada. Em uma linguagem sem sobrecarga de métodos, isso requer algo mais complicado, como um método com nome exclusivo para imprimir cada tipo de

objeto. Nesse caso, é sua responsabilidade descobrir qual método usar para cada tipo de dados.

Em Java, print() foi sobrecarregado para suportar dois tipos de referência: Object e String. E se tentarmos chamar print() com algum outro tipo de referência? Digamos, um objeto Date? Quando não há uma correspondência exata de tipo, o compilador procura uma correspondência aceitável e atribuível. Como Date, como todas as classes, é uma subclasse de Object, um objeto Date pode ser atribuído a uma variável do tipo Object. Portanto, é uma correspondência aceitável e o compilador seleciona a versão Object do método.

E se houver mais de uma correspondência possível? Por exemplo, e se quisermos imprimir o literal “Olá”? Esse literal pode ser atribuído a String (já que é uma String) ou a Object, a classe pai de String. Aqui, o compilador decide qual correspondência é

“melhor” e seleciona esse método. Neste caso, seleciona a versão String.

A explicação intuitiva para selecionar a versão String é que a classe String está “mais próxima” do tipo do nosso literal “Olá” na hierarquia de herança. É uma partida mais específica. Uma maneira um pouco mais rigorosa de especificar isso seria dizer que um determinado método é mais específico que outro método se os tipos de argumentos do primeiro método forem todos atribuíveis aos tipos de argumentos do segundo método. Neste caso, o método String é mais específico porque o tipo String pode ser atribuído ao tipo Object. O contrário não é verdade.

Se você estiver prestando atenção, deve ter notado que dissemos que o compilador resolve métodos sobrecarregados. A sobrecarga de métodos não é algo que acontece em tempo de execução; Esta é uma distinção importante. Tomar esta decisão durante a compilação significa que uma vez selecionado o método sobrecarregado, a escolha é fixa até que o código seja recompilado, mesmo que a classe que contém o método chamado seja revisada posteriormente e um método sobrecarregado ainda mais específico seja adicionado.

Essa seleção em tempo de compilação contrasta com os métodos substituídos, que estão localizados em tempo de execução e podem ser encontrados mesmo que não existissem quando a classe de chamada foi compilada. Na prática, essa distinção geralmente não será relevante para você, pois provavelmente você recompilará todas as classes necessárias ao mesmo tempo. Falaremos sobre substituição de métodos posteriormente neste capítulo.

Criação de Objeto

Objetos em Java são alocados em um espaço de memória “heap” do sistema. Ao contrário de algumas outras línguas, porém, não precisamos gerenciar essa memória nós mesmos. Java cuida da alocação e desalocação de memória para você. Java aloca explicitamente armazenamento para um objeto quando você o cria com o operador new. Mais importante ainda, os objetos são removidos pela coleta de lixo quando não são mais referenciados.

Construtores

Os objetos são alocados com o operador new usando um construtor. Um construtor é um método especial com o

mesmo nome de sua classe e sem tipo de retorno. É

chamado quando uma nova instância de classe é criada, o que dá à classe a oportunidade de configurar o objeto para uso. Os construtores, como outros métodos, podem aceitar argumentos e podem ser sobrecarregados. Eles não são, entretanto, herdados como outros métodos:

data da aula {

dia interno;

//outras variáveis de data...

//Construtor "padrão" simples

Data() {

dia =diaatual();

}

Data(data da string) {

dia = analisarDia(data);

}

// outros métodos de data ...

}

Neste trecho, a classe Date possui dois construtores. O primeiro não aceita argumentos; é conhecido como o construtor padrão. Os construtores padrão desempenham um papel especial: se você não definir nenhum construtor para uma classe, o compilador fornecerá um construtor padrão vazio para você. O

construtor padrão é chamado sempre que você cria um objeto chamando seu construtor sem argumentos.

Aqui implementamos o construtor padrão para que ele defina a variável de instância day chamando um método hipotético, currentDay(), que presumivelmente sabe como procurar o dia atual. O segundo construtor recebe um argumento String. Nesse caso, a String contém uma string de data que pode ser analisada para definir a variável day.

Dados esses construtores, podemos criar um objeto Date das seguintes maneiras: Data agora = new Date();

Data natal = new Date("25 de dezembro de 2022");

Em cada caso, Java escolhe o construtor apropriado em tempo de compilação com base nas regras para seleção de métodos sobrecarregados.

Se posteriormente removermos todas as referências a um objeto alocado, ele será coletado como lixo, como discutiremos em breve:

natal = nulo; // o Natal agora é um jogo justo para o coletor de lixo Definir esta referência como nula significa que ela não está mais apontando para o objeto de data "25 de dezembro de 2022". Definir a variável natal para qualquer outro valor teria o mesmo efeito. A menos que outra variável também se refira ao objeto de data original, a data agora estará inacessível e poderá ser coletada como lixo. Não estamos sugerindo que você tenha que definir referências como null para obter os valores coletados como lixo. Muitas vezes isso acontece naturalmente quando as variáveis locais ficam fora do escopo, mas as variáveis de instância dos objetos vivem enquanto o próprio objeto viver (através de referências a

ele) e as variáveis estáticas vivem efetivamente para sempre.

Mais algumas notas: você pode declarar construtores com os mesmos modificadores de visibilidade (público, privado ou protegido) que outros métodos, para controlar sua acessibilidade. Você não pode, entretanto, tornar os construtores abstratos, finais ou sincronizados. Falaremos em detalhes sobre modificadores abstratos, finais e de visibilidade posteriormente neste capítulo e abordaremos sincronizados em Capítulo

9.

Trabalhando com construtores sobrecarregados

Um construtor pode se referir a outro construtor na mesma classe ou à superclasse imediata usando formas especiais de this e super referências. Discutiremos o primeiro caso aqui e retornaremos ao do construtor de superclasse depois de termos falado mais sobre a criação de subclasses (geralmente chamadas simplesmente de subclasses) e herança. Um construtor pode invocar outro construtor sobrecarregado em sua classe usando o método auto-referencial chamado this() com argumentos apropriados para selecionar o construtor desejado. Se um construtor chamar outro construtor, ele deverá fazê-lo como sua primeira instrução:

classe Carro {

Modelo de corda;

portas internas;

Carro(modelo String, portas internas) {

```
este.modelo = modelo;

this.doors = portas;

// outras coisas complicadas

}

Carro(Modelo de string) {

this(modelo, 4 /* portas */);

}

}
```

Neste exemplo, a classe Car possui dois construtores. A primeira, mais explícita, aceita argumentos que especifiquem o modelo do carro e o seu número de portas. O segundo construtor toma apenas o modelo como argumento e, por sua vez, chama o primeiro construtor com um valor padrão de quatro portas. A vantagem dessa abordagem é que você pode ter um único construtor para fazer todo o complicado trabalho de configuração; outros construtores mais convenientes simplesmente alimentam os argumentos apropriados para esse construtor primário.

A chamada especial para this() deve aparecer como a primeira instrução em nosso construtor de delegação. A sintaxe é restrita dessa forma porque é necessário identificar uma cadeia de comando clara ao chamar construtores. No final da cadeia, Java invoca o construtor da superclasse (se não fizermos isso explicitamente em nosso código) para garantir que os membros herdados sejam inicializados corretamente antes de prosseguirmos.

Há também um ponto na cadeia, logo após invocar o construtor da superclasse, onde são avaliados os inicializadores das variáveis de instância da classe atual. Antes disso, não podemos nem referenciar as variáveis de instância da nossa classe. Explicaremos esta situação novamente em detalhes depois de falarmos sobre herança.

Por enquanto, tudo que você precisa saber é que você pode invocar um segundo construtor (delegar a ele) apenas como a primeira instrução do seu construtor. Por exemplo, o código a seguir é ilegal e causa um erro em tempo de compilação: Carro(String m) {

int portas = determinePortas();

isto(m, portas); // Erro: chamada do construtor

// deve ser a primeira instrução

}

O construtor de nome de modelo simples não pode fazer nenhuma configuração adicional antes de chamar o construtor mais explícito. Não pode nem se referir a um membro da instância para obter um valor constante:

classe Carro {

final int default_doors = 4;

Carro(String m) {

this(m, portas_padrão); // Erro: referenciando

//variável não inicializada

}

}

A variável de instância defaultDoors não é inicializada até um ponto posterior na cadeia de chamadas do construtor que configura o objeto, portanto o compilador não nos permite acessá-la ainda. Felizmente, podemos resolver este problema específico usando uma variável estática em vez de uma variável de instância:

classe Carro {

final estático int DEFAULT_DOORS = 4;

Carro(String m) {

isto(m, DEFAULT_DOORS); // OK!

}

}

Os membros estáticos de uma classe são inicializados quando a classe é carregada pela primeira vez na máquina virtual. O compilador pode determinar o valor desses membros estáticos para que seja seguro acessá-los em um construtor.

Destruição de Objetos

Agora que vimos como criar objetos, é hora de falar sobre como destruí-los. Se você está acostumado a programar em C ou C++, provavelmente já passou algum tempo procurando vazamentos de memória em seu código. Os programadores

acidentalmente permitem o vazamento de memória criando objetos (que consomem memória) e esquecendo

de destruí-los (o que retorna a memória alocada) quando os objetos não são mais necessários. Java cuida da destruição de objetos para você; você não precisa se preocupar com vazamentos de memória tradicionais e pode se concentrar em tarefas de programação mais importantes.5

Coleta de lixo

Java usa uma técnica conhecida como coleta de lixo para remover objetos que não são mais necessários. O coletor de lixo é o Grim Reaper do Java. Ele permanece em segundo plano, perseguindo objetos e aguardando sua morte. Ele os encontra e os observa, contando periodicamente referências a eles para ver quando chegou a sua hora. Quando todas as referências a um objeto desaparecem e ele não está mais acessível, o mecanismo de coleta de lixo declara o objeto inacessível e recupera seu espaço de volta ao conjunto de recursos disponíveis. Um objeto inacessível é aquele que não pode mais ser encontrado por meio de nenhuma combinação de referências

“ativas” no aplicativo em execução.

A coleta de lixo usa uma variedade de algoritmos; a arquitetura da máquina virtual Java não requer um esquema específico. Vale a pena notar, entretanto, como algumas implementações de Java realizaram essa tarefa. No início, Java usava uma técnica chamada “marcar e varrer”. Neste esquema, Java primeiro percorre a árvore de todas as referências de objetos acessíveis e as marca como vivas. Em seguida, ele verifica o heap, procurando objetos identificáveis que não estejam marcados. Usando essa técnica, Java pode localizar objetos no heap porque eles são armazenados de uma maneira característica e

possuem uma assinatura específica de bits em seus identificadores que provavelmente não será reproduzida naturalmente. Este tipo de algoritmo não se confunde com o problema das referências cíclicas, nas quais os objetos podem referenciar-se mutuamente e parecer vivos mesmo quando estão mortos (Java lida com esse problema automaticamente). Porém, esse esquema não era

o método mais rápido e causava pausas nos programas. Desde então, as implementações tornaram-se muito mais sofisticadas.

Os coletores de lixo Java modernos são executados continuamente sem forçar nenhum atraso longo na execução do aplicativo Java. Por fazerem parte de um sistema de tempo de execução, eles também podem realizar algumas coisas que não poderiam ser feitas estaticamente. Por exemplo, Java divide o heap de memória em diversas áreas para objetos com diferentes tempos de vida estimados. Os objetos de vida curta são colocados em uma parte especial da pilha, o que reduz drasticamente o tempo necessário para reciclá-los. Objetos que duram mais tempo podem ser movidos para outras partes menos voláteis do heap. Em implementações recentes, o coletor de lixo pode até mesmo “ajustar-se” ajustando os tamanhos das partições de heap com base no desempenho real do aplicativo. A melhoria na coleta de lixo do Java desde os primeiros lançamentos tem sido notável e é uma das razões pelas quais Java é hoje aproximadamente equivalente em velocidade a muitas linguagens tradicionais que colocam o fardo do gerenciamento de memória sobre os ombros do programador.

Em geral, você não precisa se preocupar com o processo de coleta de lixo. Mas um método de coleta de lixo pode ser útil para depuração. Você pode solicitar que o coletor de lixo faça uma limpeza explicitamente invocando o método System.gc(). Este método é completamente dependente da implementação e não pode fazer nada, mas você pode usá-lo se quiser alguma garantia de que Java pelo menos tentou limpar a memória antes de realizar uma atividade.

Pacotes

Mesmo quando nos limitamos a exemplos simples, você deve ter notado que resolver problemas em Java requer a criação de classes. Para as nossas aulas de jogo acima, temos as nossas maçãs, as nossas árvores e o nosso campo de jogo, para citar apenas alguns. Para aplicações ou bibliotecas mais complexas, você pode ter centenas ou até milhares de classes. Você precisa de uma maneira de organizar as coisas, e Java usa a noção de pacote para realizar essa tarefa. Vejamos alguns exemplos.

Lembre-se do nosso segundo exemplo de Hello World emCapítulo 2. As primeiras linhas do arquivo nos mostram muitas informações sobre onde o código está: pacote ch02.examples;

importar javax.swing.*;

classe pública HelloJava {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Olá, Java!");

Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

```
// ...

}

}
```

Nomeamos o arquivo Java como HelloJava.java de acordo com a classe principal (HelloJava) desse arquivo. Quando falamos sobre organizar coisas que vão em arquivos, você pode naturalmente pensar em usar pastas para organizar esses arquivos. Isso é essencialmente o que Java faz. Neste exemplo, usamos a palavra-chave package e atribuímos um nome de pacote ch02.examples. Os pacotes são mapeados para nomes de pastas da mesma forma que as classes são mapeadas para nomes de arquivos. No diretório onde você instalou os exemplos deste livro, então, esta classe deve ser encontrada no arquivo ch02/examples/HelloJava.java. Dê uma olhada para

trásFigura 5-1onde temos algumas classes agrupadas em seus pacotes. Se você estivesse olhando o código-fonte Java para os componentes Swing que usamos no HelloJava, por exemplo, você encontraria uma pasta chamada javax, e abaixo dela, uma chamada swing, e abaixo dela você encontraria arquivos como JFrame.java e JLabel .Java.

Cada classe pertence a exatamente um pacote. Os nomes dos pacotes seguem as mesmas regras gerais de outros identificadores Java e estão todos em minúsculas por convenção. Se você não especificar um pacote, Java atribuirá sua classe ao pacote

“padrão”. Usar o pacote padrão é adequado para demonstrações únicas, mas caso contrário você deve usar o pacote para suas aulas. O pacote padrão tem

diversas limitações — por exemplo, uma classe no pacote padrão não pode ser usada com jshell — e não deve ser usada além do teste.

Importando Classes

Um dos maiores pontos fortes do Java reside na vasta coleção de bibliotecas de suporte disponíveis sob licenciamento comercial e de código aberto. Precisa exportar um PDF? Existe uma biblioteca para isso. Precisa importar uma planilha? Existe uma biblioteca para isso. Precisa ligar aquela lâmpada inteligente no porão de um servidor web na nuvem? Há uma biblioteca para isso também. Se os computadores estiverem realizando uma tarefa ou outra, quase sempre você encontrará uma biblioteca Java para ajudá-lo a escrever código para executar essa tarefa também.

Para usar qualquer uma dessas bibliotecas maravilhosas, você as importa com a palavra-chave import habilmente nomeada. Usamos import com nosso exemplo HelloJava para que pudéssemos adicionar os componentes de quadro e rótulo da biblioteca gráfica Swing. Você pode importar classes individuais ou pacotes inteiros.

Vejamos alguns exemplos.

Importando classes individuais

Na programação, você ouvirá frequentemente a máxima de que “menos é mais”.

Menos código é mais sustentável. Menos sobrecarga significa mais rendimento e assim por diante. (Embora, ao seguirmos esse método de codificação, queiramos lembrá-lo de seguir outra citação famosa de ninguém menos que um pensador como Einstein:

“Tudo deve ser o mais simples possível, mas não mais simples.”) Se você precisar de apenas um ou mais duas classes de um pacote externo, você poderá importar

exatamente essas classes. Isso torna seu código um pouco mais legível – outras pessoas sabem exatamente quais classes você usará.

Vamos reexaminar o trecho anterior do HelloJava. Usamos uma importação geral (mais sobre isso na próxima seção), mas poderíamos restringir um pouco as coisas importando apenas as classes que precisamos, assim:

```
pacote ch02.examples;

importar javax.swing.JFrame;

importar javax.swing.JLabel;

classe pública HelloJava {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Olá, Java!");

Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

// ...

}

}
```

Esse tipo de configuração de importação é certamente mais detalhado, mas, novamente, significa que qualquer pessoa que esteja lendo ou compilando seu código conhece suas dependências exatas. Muitos IDEs ainda

possuem uma função “Otimizar Importações” que encontrará automaticamente essas dependências e as listará individualmente. Depois que você adquirir o hábito de listar e ver essas importações explícitas, será surpreendente como elas se tornam úteis ao se orientar em uma classe nova (ou talvez há muito esquecida).

Importando pacotes inteiros

É claro que nem todos os pacotes se prestam a importações individuais. Esse mesmo pacote Swing, javax.swing, é um ótimo exemplo. Se você estiver escrevendo um aplicativo gráfico para desktop, quase certamente usará o Swing – e muitos, muitos de seus componentes. Você pode importar todas as classes do pacote usando a sintaxe que abordamos anteriormente:

```
importar javax.swing.*;

classe pública HelloJava {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Olá, Java!");

Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

// ...

}

}
```

O * é uma espécie de curinga para importações de classes. Esta versão da instrução import informa ao compilador para ter todas as classes do pacote prontas para uso.

Você verá esse tipo de importação com bastante frequência para muitos dos pacotes Java comuns, como AWT, Swing, utilitários e E/S. Novamente, funciona para qualquer

pacote, mas onde fizer sentido ser mais específico, você obterá alguns aumentos de desempenho em tempo de compilação e melhorará a legibilidade do seu código.

Aviso

Embora possa parecer natural que uma importação curinga inclua tanto as classes do pacote nomeado quanto as classes de quaisquer subpacotes, Java não permite importações recursivas. Se você precisar de algumas classes de java.awt e mais algumas classes de java.awt.event, deverá fornecer linhas de importação separadas para cada pacote.

Ignorando importações

Você tem outra opção para usar classes externas de outros pacotes — você pode usar seus nomes totalmente qualificados diretamente no seu código. Por exemplo, nossa classe HelloJava usou as classes JFrame e JLabel do pacote javax.swing. Poderíamos importar apenas a classe JLabel se quiséssemos:

importar javax.swing.JLabel;

classe pública HelloJava {

public static void main(String[] args) {

quadro javax.swing.JFrame = novo javax.swing.JFrame("Olá, Java!"); Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

```
// ...

}

}
```

Isso pode parecer muito detalhado para uma linha onde criamos nosso quadro, mas em classes maiores com listas de importações já longas, usos únicos podem realmente tornar seu código mais legível. Essa entrada totalmente qualificada geralmente aponta para o uso exclusivo dessa classe em um arquivo. Se você estivesse usando essa classe muitas vezes, você a importaria ou seu pacote. Esse tipo de uso de nome completo nunca é um requisito, mas você o verá de vez em quando.

Pacotes Personalizados

À medida que você continua aprendendo Java, escreve mais código e resolve problemas maiores, sem dúvida começará a coletar um número cada vez maior de classes. Você pode usar pacotes para ajudar a organizar essa coleção. Você usa a palavra-chave package para declarar um pacote personalizado. Em seguida, você coloca o arquivo com sua classe dentro de uma estrutura de pastas correspondente ao nome do pacote. Como um rápido lembrete, os pacotes usam todos os nomes em letras minúsculas (por convenção) separados por pontos, como em nosso pacote de interface gráfica, javax.swing.

Outra convenção amplamente aplicada a nomes de pacotes é algo chamado de nomenclatura de “nome de domínio reverso”. Além dos pacotes associados diretamente ao Java, bibliotecas de terceiros e outros códigos contribuídos são

geralmente organizados usando o nome de domínio da empresa ou o endereço de email do indivíduo. Por exemplo, a Mozilla Foundation contribuiu com uma variedade de bibliotecas Java para a comunidade de código aberto. A maioria dessas bibliotecas e utilitários estarão em pacotes começando com o domínio da Mozilla, mozilla.org, na ordem inversa: org.mozilla. Essa nomenclatura reversa tem o efeito colateral útil (e intencional) de manter a estrutura de pastas no topo bastante pequena. Não é incomum que projetos de bom tamanho usem bibliotecas apenas dos domínios de nível superior com e org.

Se você estiver construindo seus próprios pacotes separados de qualquer empresa ou contrato de trabalho, poderá usar seu endereço de e-mail e revertê-lo, semelhante aos nomes de domínio da empresa. Outra opção popular para código distribuído online é usar o domínio do seu provedor de hospedagem. O GitHub, por exemplo, hospeda muitos projetos Java para amadores e entusiastas. Você pode criar um pacote chamado com.github.myawesomeproject (onde myawesomeproject seria obviamente substituído pelo nome real do seu projeto). Esteja ciente de que repositórios em sites como o GitHub geralmente permitem nomes que não são válidos em nomes de pacotes. Você pode ter um repositório chamado my-awesome-project, mas traços não são permitidos em nenhuma parte do nome de um pacote. Freqüentemente, esses caracteres ilegais são simplesmente omitidos para criar um nome válido.

Você já deve ter notado que colocamos vários exemplos deste livro em pacotes.

Embora a organização de aulas dentro de pacotes seja um tópico confuso, sem boas práticas recomendadas disponíveis, adotamos uma abordagem projetada para facilitar a localização dos exemplos enquanto você lê o livro. Para quaisquer exemplos completos em um capítulo, você verá um pacote como ch05.examples. Para o exemplo do jogo contínuo, usamos um subpacote de jogo. Colocamos os exercícios de final de capítulo em ch05.exercises.

Observação

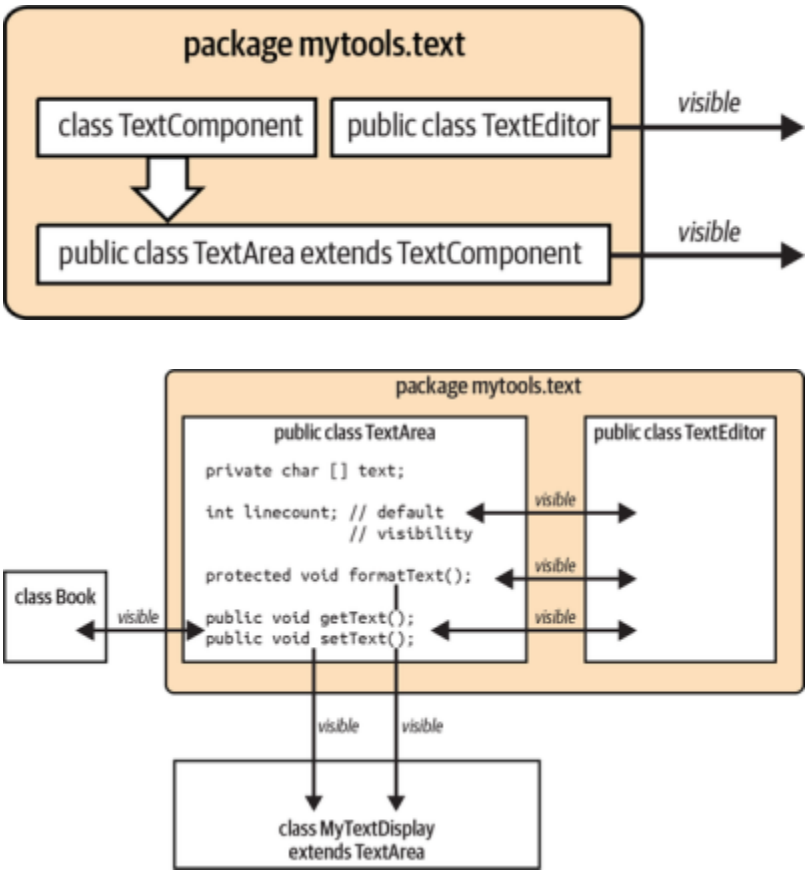
Ao compilar uma classe empacotada, você precisa informar ao compilador onde o arquivo real reside no sistema de arquivos, para usar seu caminho, com os elementos do pacote separados pelo separador do sistema de arquivos (normalmente / ou \).

Por outro lado, ao executar uma classe empacotada, você especifica seu nome totalmente qualificado e separado por pontos.

Se você estiver usando um IDE, ele gerenciará com prazer esses problemas de pacote para você. Basta criar e organizar suas aulas e continuar identificando a aula principal que dá início ao seu programa.

Visibilidade e acesso dos membros

Já falamos um pouco sobre os modificadores de acesso que você pode usar ao declarar variáveis e métodos. Tornar algo público significa que qualquer pessoa, em qualquer lugar, pode ver sua variável ou chamar seu método. Tornar algo protegido significa que qualquer subclasse pode acessar a variável, chamar o método ou substituir o método para fornecer alguma funcionalidade alternativa mais apropriada à sua



subclasse. O modificador private significa que a variável ou método está disponível apenas dentro da própria classe.

Os pacotes afetam membros protegidos. Além de serem acessíveis por qualquer subclasse, tais membros são visíveis e podem ser substituídos por outras classes no mesmo pacote. Os pacotes também entram em jogo se você deixar de lado o modificador. Considere alguns exemplos de componentes de texto no pacote personalizado mytools.text, conforme mostrado emFigura 5-4.

Figura 5-4. Pacotes e visibilidade de classe

A classe TextComponent não possui modificador. Possui visibilidade padrão ou visibilidade “pacote privado”. Isso

significa que outras classes no mesmo pacote podem acessar a classe, mas quaisquer classes fora do pacote não podem. Isso pode ser muito útil para classes específicas de implementação ou auxiliares internos. Você pode usar os elementos privados do pacote livremente na construção de seu código, mas outros programadores podem usar apenas seus elementos públicos e

protegidos.Figura 5-5mostra mais detalhes, com subclasses e classes externas usando variáveis e métodos para uma classe de amostra.

Figura 5-5. Pacotes e visibilidade dos membros

Observe que estender a classe TextArea dá acesso aos métodos públicos getText() e setText(), bem como ao método protegido formatText(). Mas MyTextDisplay (mais sobre subclasses e se estende em breve em“Subclasse e Herança”) não tem acesso à variável linecount do pacote privado. Dentro do pacote mytools.text onde criamos a classe TextEditor, entretanto, podemos chegar ao linecount, bem como aos métodos que são públicos ou protegidos. Nosso armazenamento interno para o conteúdo, texto, permanece privado e indisponível para qualquer pessoa que não seja a própria classe TextArea.

Tabela 5-2resume os níveis de visibilidade disponíveis em Java; geralmente vai do mais para o menos restritivo. Métodos e variáveis são sempre visíveis dentro de uma classe declarante, portanto a tabela não aborda esse escopo.

Tabela 5-2. Modificadores de visibilidade

Modificador

Visibilidade fora da aula

privado

Nenhum

Nenhum modificador

Aulas no pacote

(padrão)

protegido

Classes no pacote e subclasses dentro ou fora do

pacote

público

Todas as aulas

Compilando com Pacotes

Você já viu alguns exemplos de uso de um nome de classe totalmente qualificado para compilar um exemplo simples. Se não estiver usando um IDE, você terá outras opções disponíveis. Por exemplo, você pode querer compilar todas as classes de um determinado pacote. Se sim, você pode fazer isso:

% javac ch02/examples/*.java

% java ch02.examples.HelloJava

Observe que, para aplicativos comerciais, você costuma ver nomes de pacotes mais complexos que incluem vários segmentos. Como mencionamos anteriormente,

uma prática comum é reverter o nome de domínio de internet da sua empresa. Por exemplo, este livro poderia usar mais apropriadamente um prefixo de pacote completo, como com.oreilly.learningjava6e. Cada capítulo seria um subpacote sob esse prefixo. Compilar e executar classes nesses pacotes é bastante simples, embora um pouco prolixo:

```
% javac com/oreilly/learningjava6e/ch02/examples/*.java

% java
com.oreilly.learningjava6e.ch02.examples.HelloJava
```

O comando javac também entende a dependência básica de classe. Se sua classe principal usar algumas outras classes na mesma hierarquia de origem — mesmo que elas não estejam todas no mesmo pacote — compilar essa classe principal irá “pegar”

as outras classes dependentes e compilá-las também.

Além de programas simples com algumas classes em um único pacote, é mais provável que você confie em seu IDE ou em uma ferramenta de gerenciamento de compilação, como Gradle ou Maven. Essas ferramentas estão fora do escopo deste livro, mas há muitas referências on-line sobre elas. O Maven, em particular, é popular para gerenciar grandes projetos com muitas dependências. Ver *Maven: o guia definitivo* do criador do Maven, Jason Van Zyl, e sua equipe da Sonatype para uma verdadeira exploração dos recursos e capacidades desta ferramenta poderosa.6

Design de Classe Avançado

Você pode se lembrar de“HelloJava2: A Sequela”que tínhamos duas classes no mesmo arquivo. Isso

simplificou o processo de escrita e compilação, mas não concedeu a nenhuma das classes acesso especial à outra. À medida que você começar a pensar em problemas mais complexos, você encontrará casos em que um design de classe mais avançado que conceda acesso especial não é apenas útil, mas fundamental para escrever código sustentável.

Subclasse e herança

As classes em Java existem em uma hierarquia. Você pode declarar uma classe em Java como uma subclasse de outra classe usando a palavra-chave extends. Uma subclasse herda variáveis e métodos de sua superclasse e pode usá-los como se tivessem sido declarados dentro da própria subclasse:

```
classe Animal {

peso flutuante;

void comer() {

//comendo coisas

}

// outras coisas de animais

}

class Mamífero estende Animal {

//herda o peso

int frequência cardíaca;

// herda comer()
```

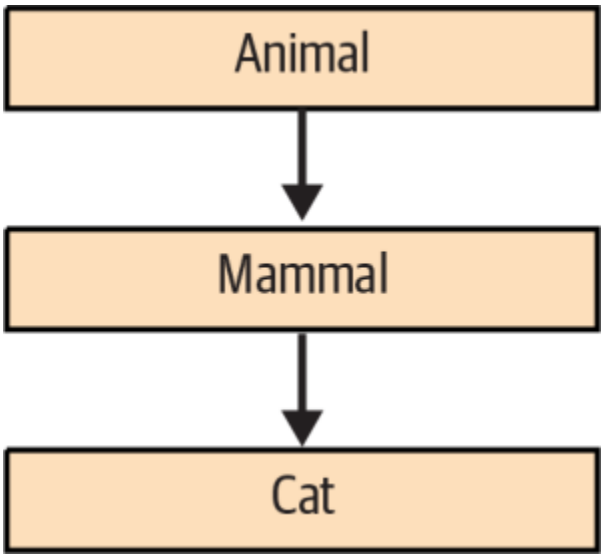
```
void respirar() {

//respirar

}

}
```

Neste exemplo, um objeto do tipo Mammal possui a variável de instância peso e o método eat(). Eles são herdados do Animal.

Uma classe pode estender apenas uma outra classe. Para usar a terminologia adequada, Java permite herança única de implementação de classe. Mais adiante neste capítulo, falaremos sobre interfaces, que substituem a herança múltipla encontrada em outras linguagens.

Uma subclasse pode ser subclassificada posteriormente. Normalmente, a subclasse especializa ou refina uma classe adicionando variáveis e métodos (você não pode remover ou ocultar variáveis ou métodos por meio da subclasse). Por exemplo: class Gato estende Mamífero {

// herda peso e heartRate




```
booleano cabelo longo;

// herda comer() e respirar()

void ronronar() {

//faz sons legais

}

}
```

A classe Cat é um tipo de Mamífero que é, em última análise, um tipo de Animal. Os objetos Gato herdam todas as características dos objetos Mamíferos e, por sua vez, dos objetos Animais. Cat também fornece comportamento adicional na forma do método purr() e da variável longHair. Podemos denotar o relacionamento de classe em um diagrama, como mostrado emFigura 5-6.

Figura 5-6. Uma hierarquia de classes

Uma subclasse herda todos os membros de sua superclasse não designados como privados. Como

discutiremos em breve, outros níveis de visibilidade afetam quais membros herdados da classe podem ser vistos fora da classe e de suas subclasses, mas, no mínimo, uma subclasse sempre tem o mesmo conjunto de membros visíveis que seu pai. Por esse motivo, o tipo de uma subclasse pode ser considerado um subtipo de seu pai, e as instâncias do subtipo podem ser usadas em qualquer lugar onde as instâncias do supertipo sejam permitidas. Considere o seguinte exemplo: Gato simon = new Gato();

Criatura animal = simão;

A instância Cat simon neste exemplo pode ser atribuída à variável de tipo Animal criatura porque Cat é um subtipo de Animal. Da mesma forma, qualquer método que aceite um objeto Animal também aceitaria uma instância de um tipo Cat ou de qualquer tipo Mammal. Este é um aspecto importante do polimorfismo em uma linguagem orientada a objetos como Java. Veremos como ele pode ser usado para refinar o comportamento de uma classe, bem como adicionar novos recursos a ela.

Variáveis sombreadas

Vimos que uma variável local com o mesmo nome de uma variável de instância obscurece (oculta) a variável de instância. Da mesma forma, uma variável de instância em uma subclasse pode ocultar uma variável de instância de mesmo nome em sua classe pai, conforme mostrado emFigura 5-7.

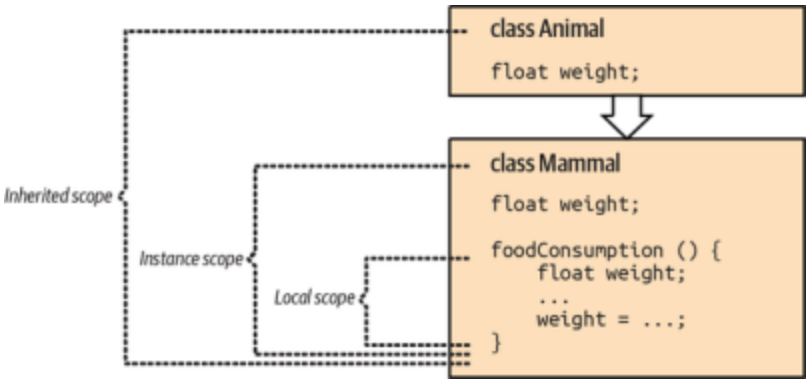


Figura 5-7. O escopo das variáveis sombreadas

A variável peso é declarada em três lugares: como variável local no método foodConsumption() da classe Mammal, como variável de instância da própria classe Mammal e como variável de instância da classe Animal. A variável real que você referenciaria no código dependeria do escopo em que você está trabalhando e de como você qualifica a referência a ela.

No exemplo anterior, todas as variáveis eram do mesmo tipo. Um uso um pouco mais plausível de variáveis sombreadas envolveria a alteração de seus tipos. Poderíamos, por exemplo, sombrear uma variável int com uma variável double em uma subclasse que precisa de valores decimais em vez de valores inteiros. Podemos fazer isso sem alterar o código existente porque, como o próprio nome sugere, quando sombreamos variáveis, não as substituímos, mas sim as mascaramos. Ambas as variáveis ainda existem; os métodos da superclasse veem a variável original e os métodos da subclasse veem a nova versão. Quais variáveis os vários métodos veem são determinadas em tempo de compilação.

Aqui está um exemplo simples:

classCalculadoraInteira {

```
soma interna;

// outras coisas inteiras ...

}

class CalculadoraDecimal estende CalculadoraInteira {

soma dupla;

// outras coisas de ponto flutuante ...

}
```

Neste exemplo, sombreamos a variável de instância sum para alterar seu tipo de int para double.7Os métodos definidos na classe IntegerCalculator veem a soma da variável inteira, enquanto os métodos definidos em DecimalCalculator veem a soma da variável de ponto flutuante. No entanto, ambas as variáveis realmente existem para uma determinada instância de DecimalCalculator e podem ter valores independentes.

Na verdade, qualquer método que DecimalCalculator herda de IntegerCalculator realmente vê a soma da variável inteira. Se isso parece confuso, certamente pode ser.

O sombreamento é algo que você deve evitar sempre que possível. Mas nem sempre é possível evitá-lo, por isso queremos ter certeza de que você viu alguns exemplos, ainda que artificiais.

Como ambas as variáveis existem em DecimalCalculator, precisamos de uma maneira de fazer referência à variável herdada de IntegerCalculator. Fazemos isso

usando a palavra-chave super como qualificador na referência:

int s = super.soma;

Dentro de DecimalCalculator, a palavra-chave super usada desta maneira seleciona a variável de soma definida na superclasse. Explicaremos o uso de super mais detalhadamente em breve.

Outro ponto importante sobre variáveis sombreadas tem a ver com como elas funcionam quando nos referimos a um objeto por meio de um tipo menos derivado (um tipo pai). Por exemplo, podemos nos referir a um objeto DecimalCalculator como um IntegerCalculator usando-o por meio de uma variável do tipo IntegerCalculator. Se fizermos isso e acessarmos a variável soma, obteremos a variável inteira, não a decimal:

CalculadoraDecimal dc = new CalculadoraDecimal();

Calculadora Inteira ic = dc;

int s = ic.soma; // acessa a soma do IntegerCalculator

O mesmo aconteceria se acessássemos o objeto usando uma conversão explícita para o tipo IntegerCalculator ou ao passar uma instância para um método que aceita esse tipo pai.

Para reiterar, a utilidade das variáveis sombreadas é limitada. É muito melhor abstrair o uso de variáveis como essa de outras maneiras do que usar regras complicadas de escopo. Entretanto, é importante entender os conceitos aqui antes de falarmos sobre fazer a mesma coisa com métodos. Veremos um tipo de comportamento diferente e mais dinâmico quando os métodos

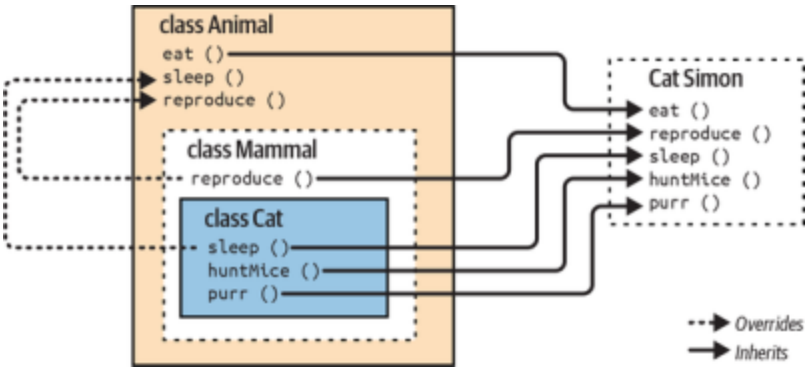
obscurecem outros métodos ou, para usar a terminologia correta, substituem outros métodos. Substituir métodos em uma subclasse é bastante comum e pode ser muito poderoso.

Substituindo métodos

Vimos que podemos declarar métodos sobrecarregados (métodos com o mesmo nome, mas com um número ou tipo de argumentos diferente) dentro de uma classe. A seleção de métodos sobrecarregados funciona da maneira que descrevemos em todos os métodos disponíveis para uma classe, incluindo os herdados. Isso significa que uma subclasse pode definir métodos sobrecarregados adicionais que se somam aos métodos sobrecarregados fornecidos por uma superclasse.

Uma subclasse pode fazer mais do que isso; ele pode definir um método que tenha exatamente a mesma assinatura de método (nome e tipos de argumento) que um método em sua superclasse. Nesse caso, o método na subclasse substitui o método na superclasse e efetivamente substitui sua implementação, conforme mostrado

emFigura 5-8. A substituição de métodos para alterar o comportamento dos objetos é



chamada de polimorfismo de subtipo. É o uso que a maioria das pessoas pensa quando fala sobre o poder das linguagens orientadas a objetos.

Figura 5-8. Substituição de método

EmFigura 5-8, Mammal substitui o método reproduz() de Animal, talvez para especializar o método para o comportamento de mamíferos que dão à luz filhotes vivos.8O comportamento de sono do objeto Cat também é substituído para ser diferente daquele de um Animal geral, talvez para acomodar cochilos. A classe Cat também adiciona comportamentos mais exclusivos de ronronar e caçar ratos.

Pelo que você viu até agora, os métodos sobrescritos provavelmente se parecem com métodos sombreados em superclasses, assim como as variáveis. Mas os métodos substituídos são, na verdade, mais poderosos que isso. Quando existem múltiplas implementações de um método na hierarquia de herança de um objeto, aquele na classe “mais derivada” (a mais abaixo na hierarquia) sempre substitui os outros, mesmo se nos referirmos ao objeto através de uma referência de um dos os tipos de superclasse.9

Por exemplo, se tivermos uma instância Cat atribuída a uma variável do tipo mais geral Animal, e chamarmos seu método sleep(), ainda teremos o método sleep() implementado na classe Cat, não aquele em Animal:

```
Gato simon = new Gato();

Criatura animal = simão;

// ...
```

criatura.sleep(); // acessa Cat sleep();

Em outras palavras, para fins de comportamento (invocação de métodos), um Gato age como um Gato, independentemente de você se referir a ele como tal. Você deve se lembrar que ao acessar uma variável sombreada por meio de nossa variável Animal, a criatura encontraria essa variável na classe Animal, não na classe Cat. No entanto, como os métodos são localizados dinamicamente, pesquisando primeiro as subclasses, o tempo de execução invocará o método apropriado na classe Cat, mesmo que o tratemos de forma mais geral como um objeto Animal. Isso significa que o comportamento dos objetos é dinâmico. Podemos lidar com objetos especializados como se fossem tipos mais gerais e ainda tirar vantagem de suas implementações especializadas de comportamento.

Classes e métodos abstratos

Às vezes você não tem um bom comportamento padrão para um método. Pense em como os animais se comunicam. Latido do cachorro. Gatos miam. Vacas mugem.

Realmente não existe um som padrão. Em Java, você pode criar um método abstrato que defina exatamente a aparência de um método, sem especificar nenhum comportamento específico. Você usa o modificador abstrato ao declarar o método. E

em vez de fornecer um corpo, você simplesmente termina a definição com um ponto e vírgula. Considere um método makeSound() para nossos animais:

classe pública Animal {

```
peso flutuante;

// outras características dos animais...

public abstract void makeSound(int duração);

// outros comportamentos animais...

}
```

Observe que nosso método para produzir um som tem uma assinatura completa (lembre-se“Executando aplicativos Java”). É nulo (não tem valor de retorno) e leva um argumento do tipo int. Mas não tem corpo. Este tipo de método é explicitamente projetado para ser substituído. Você não pode chamar um método abstrato; você receberá um erro em tempo de compilação. Você deve criar uma nova subclasse que forneça a lógica para o método abstrato antes de poder usá-lo:

```
classe pública Gato estende Animal {

// ...

public void makeSound(int duração) {

for (int contagem = 0; contagem < duração; contagem++) {

// assume que nosso som leva um segundo para ser produzido e podemos

// repete o som para corresponder à duração solicitada

System.out.println("miau!");

}
```

```
}

// ...

}
```

Com uma instância de Cat, agora podemos chamar o método makeSound() e o compilador sabe o que fazer. Mas como Animal agora contém um método abstrato, não podemos criar uma instância dessa classe. Para usar Animal temos que criar uma subclasse e preencher makeSound() como fizemos com nosso Cat.

Na verdade, se incluirmos um método abstrato em nossa classe, também teremos que declarar a própria classe como abstrata. Nosso trecho de Animal acima não seria compilado. Usamos a mesma palavra-chave abstrata para a classe:

```
classe abstrata pública Animal {

// Observe que a definição da classe precisa do modificador "abstrato"

public abstract void makeSound(int duração);

// ...

}
```

Esta declaração informa ao compilador (e outros desenvolvedores) que você projetou esta classe para fazer parte de um programa maior. Você espera (na verdade, exige) que as subclasses estendam sua classe abstrata e preencham quaisquer detalhes ausentes. Essas subclasses, por sua vez, podem ser instanciadas e realizar trabalho real.

Classes abstratas também podem conter métodos típicos completos com corpo. No caso do Animal, por exemplo, poderíamos adicionar alguns métodos auxiliares para o peso do animal:

```
classe abstrata pública Animal {

peso duplo privado;

// ...

public abstract void makeSound(int duração);

// ...

public void setWeight(duplo w) {

este.peso = w;

}

public double getPeso() {

peso de retorno;

}

}
```

Isso é comum e uma boa prática de design. Sua classe Animal contém o máximo possível de informações e comportamentos básicos e compartilhados. Mas para coisas que não são compartilhadas, você cria uma subclasse com as características e ações desejadas.

Interfaces

Java expande o conceito de métodos abstratos com interfaces. Muitas vezes é desejável especificar um grupo de métodos abstratos que definam algum comportamento para um objeto sem vinculá-lo a nenhuma implementação. Em Java, isso é chamado de interface. Uma interface define um conjunto de métodos que uma classe deve implementar. Uma classe em Java pode declarar que implementa uma interface se implementar os métodos necessários. Ao contrário da extensão de uma classe abstrata, uma classe que implementa uma interface não precisa herdar de nenhuma parte específica da hierarquia de herança ou usar uma implementação específica.

As interfaces são como distintivos de mérito do Escotismo. Um batedor que aprendeu a construir uma casa de passarinho pode andar por aí usando um remendo de pano ou uma faixa com a foto de uma. Isto diz ao mundo: “Eu sei como construir uma casa de passarinho”. Da mesma forma, uma interface é uma lista de métodos que definem algum conjunto de comportamento para um objeto. Qualquer classe que implemente cada método listado na interface pode declarar em tempo de compilação que

implementa a interface e usar, como seu emblema de mérito, um tipo extra – o tipo da interface.

Tipos de interface agem como tipos de classe. Você pode declarar variáveis como sendo de um tipo de interface, pode declarar argumentos de métodos para aceitar tipos de interface e pode especificar que o tipo de retorno de um método é um tipo de interface. Em cada caso, você está dizendo que qualquer objeto que implemente a interface (ou seja, use o distintivo de mérito correto) pode preencher essa função.

Nesse sentido, as interfaces são ortogonais à hierarquia de classes. Eles ultrapassam os limites do tipo de objeto que um item é e lidam com ele apenas em termos do que ele pode fazer. Você pode implementar quantas interfaces forem necessárias para qualquer classe. As interfaces em Java substituem grande parte da necessidade de herança múltipla em outras linguagens (e todas as complicações complicadas que vêm da verdadeira herança múltipla).

Uma interface se parece, essencialmente, com uma classe puramente abstrata (uma classe com apenas métodos abstratos). Você define uma interface com a palavra-chave interface e lista seus métodos sem corpos, apenas protótipos (assinaturas): interface acionável {

booleano startEngine();

void stopEngine();

float acelerar(float acc);

giro booleano (direção de direção);

}

O exemplo anterior define uma interface chamada Driveable com quatro métodos. É

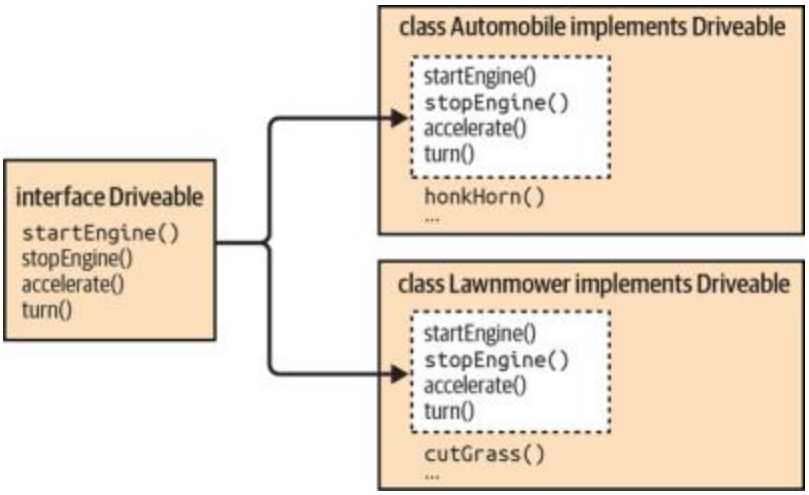
aceitável, mas não necessário, declarar os métodos em uma interface com o modificador abstrato; não fizemos isso aqui. Mais importante ainda, os métodos de uma interface são sempre considerados públicos e, opcionalmente, você pode declará-los como tal. Por que público? Bem, o usuário da interface não seria necessariamente capaz de vê-los de outra forma, e as

interfaces geralmente têm como objetivo descrever o comportamento de um objeto, não sua implementação.

As interfaces definem capacidades, por isso é comum nomear as interfaces com base em suas capacidades. Dirigível, Executável e Atualizável são bons nomes de interface.

Qualquer classe que implemente todos os métodos pode então declarar que implementa a interface usando uma cláusula implements especial em sua definição de classe. Por exemplo:

```
class Automóvel implementa Driveable {

// Traços do automóvel poderiam ir aqui...

//construi todos os métodos Driveable

public boolean startEngine() {

if (notTooCold)

motorRunning = verdadeiro;

// ...

}
```




```
public void stopEngine() {

motorRunning = falso;

}

public float acelerar(float acc) {

// ...

}

public boolean turn(diretório de direção) {

// ...

}

// Faça outras coisas no carro...

}
```

Aqui, a classe Automobile implementa os métodos da interface Driveable e se declara um tipo de Driveable usando a palavra-chave implements.

Como mostrado emFigura 5-9, outra classe, como Lawnmower, também pode implementar a interface Driveable. A figura ilustra a interface Driveable sendo implementada por duas classes diferentes. Embora seja possível que tanto o automóvel quanto o cortador de grama possam derivar de algum tipo primitivo de veículo, eles não precisam fazê-lo neste cenário.

Figura 5-9. Implementando a interface Driveable

Após declarar a interface, temos um novo tipo, Driveable. Podemos declarar variáveis do tipo Driveable e atribuir a elas qualquer instância de um objeto Driveable:
Automóvel auto = new Automóvel();

Cortador de grama = novo Cortador de grama();

Veículo dirigível;

veículo = automóvel;

veículo.startEngine();

veículo.stopEngine();

veículo = cortador;

veículo.startEngine();

veículo.stopEngine();

Tanto o Automóvel quanto o Cortador de grama implementam Driveable, portanto podem ser considerados objetos intercambiáveis desse tipo.

Como mencionamos anteriormente, as interfaces desempenham um papel crítico no poder e na popularidade do Java. Nós os usaremos ao longo dos

capítulos restantes. Se eles não fizerem muito sentido, continue lendo e trabalhando nos exercícios de código.

Você terá muito mais prática com eles. A prática nem sempre leva à perfeição, mas definitivamente torna algo menos estranho e opaco.

Classes Internas

Todas as classes que vimos até agora neste livro são classes “independentes” de nível superior, declaradas em nível de arquivo e pacote. Mas as classes em Java podem, na verdade, ser declaradas em qualquer nível de escopo, dentro de qualquer conjunto de chaves – em outras palavras, em quase qualquer lugar onde você possa colocar qualquer outra instrução Java. Essas classes internas pertencem a outra classe ou método como uma variável faria e podem ter sua visibilidade limitada ao seu escopo da mesma forma.

As classes internas são um recurso útil e esteticamente agradável para estruturar código. Suas primas, classes internas anônimas, são uma abreviatura ainda mais poderosa que faz parecer que é possível criar novos tipos de objetos dinamicamente no ambiente de tipo estaticamente do Java. Em Java, classes internas anônimas desempenham parte do papel de fechamentos em outras linguagens, dando o efeito de tratar estado e comportamento independentemente das classes. (Você também pode usar lambdas em muitos lugares onde classes internas ou anônimas funcionam.

Lambdas encapsulam bits de lógica e são comuns em muitas linguagens funcionais e LISPs. Veremos eles com muito mais detalhes emCapítulo 11.)

Entretanto, à medida que nos aprofundamos em seu funcionamento interno, veremos que as classes internas não são tão esteticamente agradáveis ou dinâmicas quanto parecem. As classes internas são puro açúcar sintático; o tempo de execução Java não os suporta. Em vez disso, o compilador mapeia o código de uma classe interna para uma classe regular nomeada de maneira inteligente. Como programador, talvez você nunca precise saber disso; você pode simplesmente confiar em classes internas como qualquer outra construção de linguagem. No entanto, você deve saber um pouco sobre como eles funcionam para entender melhor o código compilado e observar alguns efeitos colaterais potenciais.

As classes internas são essencialmente classes aninhadas. Por exemplo: classe Animal {

peso duplo;

classe Cérebro {

volume duplo;

// mais coisas cerebrais...

}

}

Aqui, a classe Brain é uma classe interna: é uma classe declarada dentro do escopo da classe Animal. Embora os detalhes do que isso significa exijam um pouco de explicação, começaremos dizendo que Java tenta fazer com que o significado, tanto quanto possível, seja o mesmo que para os outros membros (métodos e

variáveis) que vivem nesse nível de escopo. . Por exemplo, vamos adicionar um método à classe Animal:

```
classe Animal {

peso duplo;

classe Cérebro {

volume duplo;

// mais coisas cerebrais...

}

void performBehavior() { ... }

}
```

A classe interna Brain, o método performBehavior() e a variável de peso estão dentro do escopo de Animal. Portanto, em qualquer lugar dentro do Animal, podemos nos referir ao Brain, performBehavior() e ao peso diretamente, pelo nome. Dentro do Animal, podemos chamar o construtor do Brain (new Brain()) para obter um objeto Brain ou invocar performBehavior() para executar a função desse método. Mas nenhum desses elementos é geralmente acessível fora da classe Animal sem alguma qualificação adicional.

Dentro do corpo da classe interna Brain e do corpo do método performBehavior(), temos acesso direto à variável de peso, bem como a todos os outros métodos e variáveis da classe Animal. Portanto, assim como o método performBehavior() poderia funcionar com a classe Brain e criar instâncias de Brain, os métodos dentro da classe Brain podem invocar o método

performBehavior() de Animal ou trabalhar com a variável de peso. A classe Brain “vê” todos os métodos e variáveis da classe Animal diretamente em seu escopo.

O acesso do cérebro às variáveis e métodos do Animal tem consequências importantes. De dentro do Brain, podemos invocar o método performBehavior(); isto é, de dentro de uma instância de Brain, podemos invocar o método performBehavior() de uma instância de Animal. Bem, qual instância de Animal? Se tivermos vários objetos Animal por aí (digamos, alguns gatos e cachorros), precisamos saber de quem é o método performBehavior() que estamos chamando. O que significa uma definição de classe estar “dentro” de outra definição de classe? A resposta é que um objeto Cérebro sempre vive dentro de uma única instância de Animal: aquela sobre a qual foi informado quando foi criado. Chamaremos o objeto que contém qualquer instância de Brain de sua instância envolvente.

Um objeto Brain não pode viver fora de uma instância envolvente de um objeto Animal. Onde quer que você veja uma instância de Brain, ela estará vinculada a uma instância de Animal. Embora seja possível construir um objeto Brain de outro lugar

(talvez outra classe), Brain sempre requer uma instância envolvente de Animal para

“segurá-lo”. Se você encontrar uma classe Brain of the Animal, ela ainda estará explicitamente associada a Animal como a classe Animal.Brain. Assim como acontece com o método performBehavior(), modificadores podem ser aplicados para restringir sua visibilidade. Todos os modificadores de visibilidade usuais se aplicam, e as classes internas também podem ser

declaradas estáticas, como discutiremos em capítulos posteriores.

Classes internas anônimas

Agora chegamos à melhor parte. Como regra geral, quanto mais profundamente encapsuladas e limitadas em escopo forem nossas classes, mais liberdade teremos para nomeá-las. Vimos isso em nosso exemplo anterior de iterador. Esta não é apenas uma questão puramente estética. A nomenclatura é uma parte importante da escrita de código legível e de fácil manutenção. Geralmente, use os nomes mais concisos e significativos possíveis. Como corolário, evite atribuir nomes a objetos temporários que serão usados apenas uma vez.

Classes internas anônimas são uma extensão da sintaxe da nova operação. Ao criar uma classe interna anônima, você combina uma declaração de classe com a alocação de uma instância dessa classe, criando efetivamente uma classe “única” e uma instância em uma operação. Após a palavra-chave new, você especifica o nome de uma classe ou interface, seguido por um corpo de classe. O corpo da classe se torna uma classe interna. Ele estende a classe especificada ou, no caso de uma interface, espera-se que implemente a interface. Uma única instância da classe é criada e retornada como valor.

Por exemplo, poderíamos revisitar a aplicação gráfica de“HelloJava2: A Sequela”.

Como você deve se lembrar, esse aplicativo cria um HelloComponent2 que estende JComponent e implementa a interface MouseMotionListener. (Esse exemplo faz mais sentido agora?) Nunca esperamos que

HelloComponent2 responda a eventos de movimento do mouse vindos de outros componentes. Pode fazer mais sentido criar uma classe interna anônima especificamente para mover nosso rótulo “Hello”.

Na verdade, como HelloComponent2 se destina apenas ao uso em nossa

demonstração, poderíamos refatorar (um processo comum de desenvolvedor para otimizar ou melhorar o código que já está funcionando) essa classe separada em uma classe interna. Agora que sabemos um pouco mais sobre construtores e herança, também poderíamos tornar nossa classe uma extensão de JFrame em vez de construir um quadro dentro de nosso método main(). E para colocar um pouco de cereja neste bolo recém-refatorado, podemos mover nosso código de ouvinte de mouse para uma classe interna anônima dedicada ao nosso componente personalizado.

Aqui está nosso HelloJava3 com essas refatorações bacanas em vigor:

pacote ch05.examples;

importar java.awt.*;

importar java.awt.event.*;

importar javax.swing.*;

classe pública HelloJava3 estende JFrame {

public static void main(String[] args) {

Demonstração HelloJava3 = new HelloJava3();

```
demo.setVisible(verdadeiro);

}

public HelloJava3() {

super("OláJava3");

add(new HelloComponent3("Olá, Java interno!"));

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

setSize(300, 300);

}

classe HelloComponent3 estende JComponent {

String theMessage;

int mensagemX = 125, mensagemY = 95; //coordenadas
da mensagem

public HelloComponent3(String mensagem) {

aMensagem = mensagem;

addMouseMotionListener(new MouseMotionListener() {

public void mouseDragged(MouseEvent e) {

mensagemX = e.getX();

mensagemY = e.getY();

repintar();

}
```

```
public void mouseMoved(MouseEvent e) { }

});

}

public void paintComponent(Gráficos g) {

g.drawString(theMessage, mensagemX, mensagemY);

}

}

}
```

Tente compilar e executar este exemplo. Ele deve se comportar exatamente como o aplicativo HelloJava2 original. A verdadeira diferença é como organizamos as classes e quem pode acessá-las (e as variáveis e métodos dentro delas). HelloJava3

provavelmente parece um pouco complicado em comparação com HelloJava2 e é detalhado para uma demonstração tão pequena.

O poder das classes e interfaces internas começará a brilhar à medida que você desenvolver aplicativos mais complexos. Praticar a estrutura e as regras desses recursos o ajudará a escrever um código mais sustentável no longo prazo.

Organizando conteúdo e planejando falhas

As classes são a ideia mais importante em Java. Eles formam o núcleo de todo programa executável, biblioteca portátil ou auxiliar. Analisamos o conteúdo das aulas e como as classes se relacionam entre si em um

projeto maior. Sabemos mais sobre como criar e destruir objetos com base nas classes que escrevemos. E vimos como as classes internas (e as classes internas anônimas) podem nos ajudar a escrever um código mais sustentável. Veremos mais dessas classes internas à medida que nos aprofundarmos em tópicos, como tópicos emCapítulo 9e balançarCapítulo 12.

Ao criar suas classes, aqui estão algumas diretrizes que você deve ter em mente: **Oculte o máximo possível da sua implementação**

Nunca exponha mais partes internas de um objeto do que o necessário. Esta é a chave para construir código reutilizável e sustentável. Evite usar variáveis públicas em seus objetos, com a notável exceção de constantes. Em vez disso, defina métodos acessadores para definir e retornar valores. Isso é útil mesmo que sejam tipos simples

– pense em métodos como getWeight() e setWeight(). Você será capaz de modificar e estender o comportamento de seus objetos no futuro sem quebrar outras classes que dependem deles.

Use composição em vez de herança

Especialize objetos apenas quando for necessário. Ao usar um objeto em sua forma existente, como parte de um novo objeto, você está compondo objetos. Ao alterar ou refinar o comportamento de um objeto (por meio de subclasses), você está usando herança. Tente reutilizar objetos por composição em vez de herança sempre que possível. Ao compor objetos, você aproveita ao máximo as ferramentas existentes. A herança envolve quebrar o encapsulamento de um objeto, então faça isso somente

quando houver uma vantagem real. Pergunte a si mesmo se você realmente precisa herdar a classe inteira (você quer que ela seja um “tipo” daquele objeto?) ou se você pode apenas incluir uma instância dessa classe em sua própria classe e delegar algum trabalho para essa instância incluída .

Minimize os relacionamentos entre objetos e tente organizar objetos relacionados em pacotes

Pacotes Java (lembre-seFigura 5-1) também pode ocultar classes que não são de interesse geral. Exponha apenas classes que você pretende que outras pessoas usem.

Quanto mais fracamente acoplados seus objetos, mais fácil será reutilizá-los posteriormente.

Podemos aplicar estes princípios mesmo em pequenos projetos. A pasta

ch05/examples contém versões simples das classes e interfaces que usaremos para criar nosso jogo de lançar maçãs. Reserve um momento para ver como as classes Apple, Tree e Physicist implementam a interface GamePiece — como o método draw() que cada classe inclui. Observe como Field estende JComponent e como a classe principal do jogo, AppleToss, estende JFrame. Você pode ver essas peças simples tocando juntas emFigura 5-10.

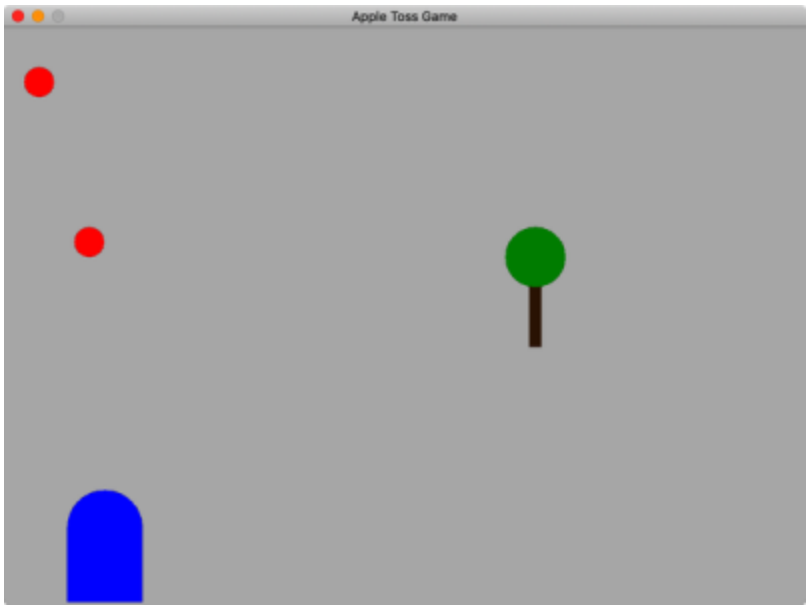


Figura 5-10. Nossas primeiras aulas de jogos em ação

O exercício de código final deste capítulo ajudará você a começar. Veja os comentários nas aulas. Tente ajustar algumas coisas. Adicione outra árvore. Mais brincadeira é sempre bom. Estaremos desenvolvendo essas classes ao longo dos capítulos restantes, portanto, ficar confortável com a forma como elas se encaixam tornará tudo mais fácil.

Independentemente de como você organiza os membros em suas classes, as classes em seus pacotes ou os pacotes em seu projeto, você terá que lidar com erros. Alguns serão erros simples de sintaxe que você corrigirá em seu editor. Outros erros são mais interessantes e podem surgir apenas enquanto o programa está em execução. O

próximo capítulo abordará a noção do Java sobre esses problemas e ajudará você a lidar com eles.

Perguntas de revisão

1. Qual é a principal unidade organizadora em Java?

2. Qual operador você usa para criar um objeto (ou instância) de uma classe?

3. Java não suporta herança múltipla clássica. Que mecanismos o Java oferece como alternativas?

4. Como você pode organizar várias classes relacionadas?

5. Como você inclui classes de outros pacotes para uso em seu próprio código?

6. Como você chama uma classe definida dentro do escopo de outra classe? Quais são alguns recursos que tornam essa classe útil em algumas circunstâncias?

7. Como você chama um método projetado para ser substituído que possui nome, tipo de retorno e lista de argumentos, mas não possui corpo?

8. O que é um método sobrecarregado?

9. Se você quiser ter certeza de que nenhuma outra classe poderá usar uma variável que você definiu, qual modificador de acesso você deve usar?

Exercícios de código

1. Para sua primeira prática de codificação, crie um pequeno zoológico de criaturas comunicativas. O arquivo ch05/exercises/Zoo.java contém um esboço

completo para criar alguns animais e fazê-los “falar” usando classes internas.

Comece preenchendo o método speak() da classe Gibbon incluída usando a classe interna Lion concluída como exemplo. Ao compilar e executar seu Zoo, você deverá ver um resultado como este (com os ruídos de seus próprios animais, é claro):

% cd ch05/exercícios

% javac Zoo.java

% java Zoológico

Vamos ouvir alguns animais!

O leão vai "rugir"

O gibão faz "hoot"

2. Agora adicione seu próprio animal ao zoológico. Crie uma nova classe interna semelhante ao Lion. Preencha o som apropriado para o seu animal e adicione-o à seção de saída do método listen(). Sua nova saída será semelhante a:

% java Zoológico

Vamos ouvir alguns animais!

O leão vai "rugir"

O gibão faz "hoot"

A foca vai "latir"

3. Vamos limpar esse método listen(). Atualmente usamos métodos print() e println() separados para cada animal. Se adicionarmos outro animal (ou vários), isso exigirá copiar, colar e ajustar as linhas de saída – tarefas

que podem introduzir erros. Adicione outro método abstrato ao Animal chamado getSpecies(). Nas subclasses, este método deve retornar o nome do animal como String, como “leão” ou “selo” dos exemplos acima.

Com esse método implementado, refatore a seção de saída para colocar seus animais em uma pequena matriz e, em seguida, use um loop para produzir a saída. (Sinta-se à vontade para editar sua classe Zoo existente. Nossa solução está em uma nova classe, Zoo2, para que você possa examinar a solução para este problema, bem como a solução para o exercício anterior.)

4. Execute o jogo de lançamento de maçãs. Compile e execute a classe ch05.exercises.game.AppleToss na pasta ch05/exercises/game usando as

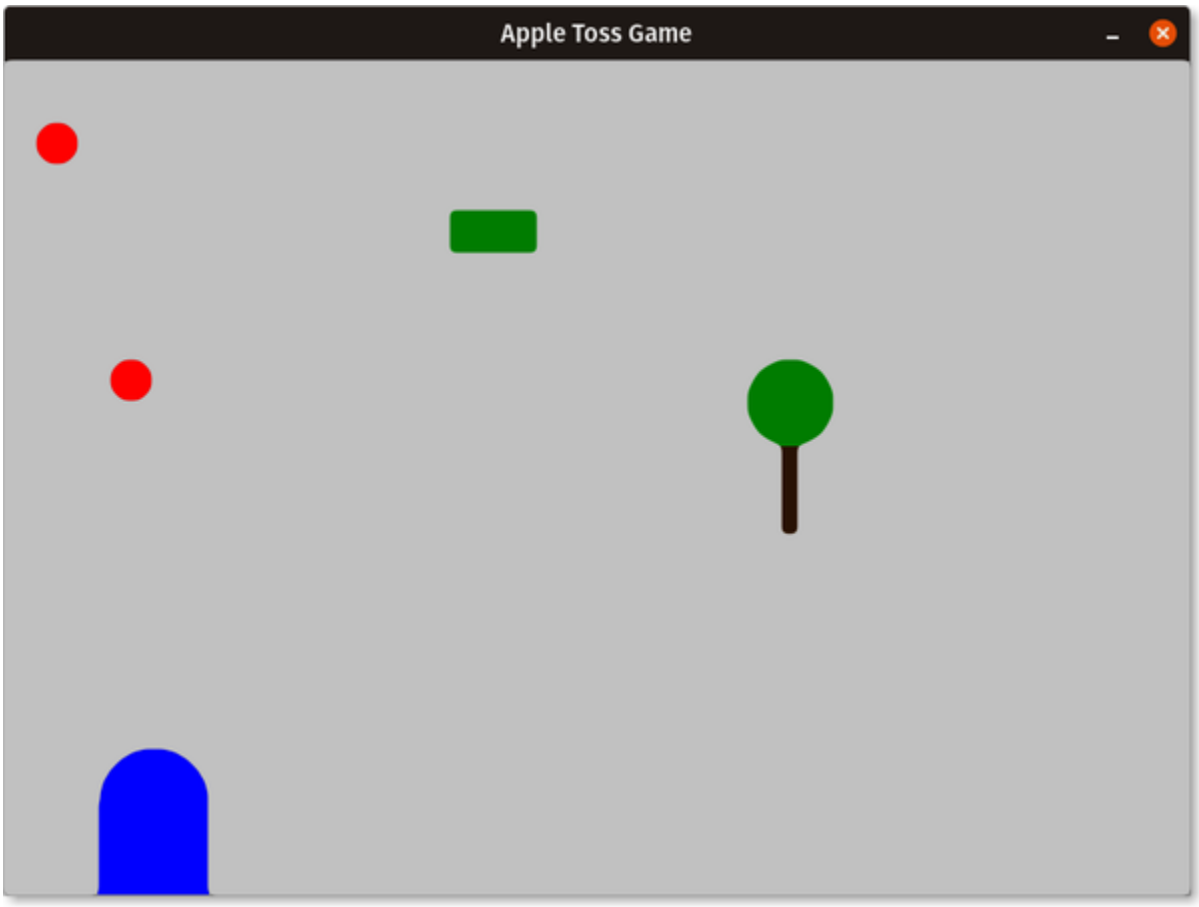
etapas discutidas anteriormente em“Pacotes Personalizados”.

Exercícios Avançados

1. Para um desafio mais avançado, vamos expandir nosso jogo de lançamento de maçãs criando um novo tipo de obstáculo. Use a classe Tree como modelo e crie uma classe Hedge. Você pode desenhar sua cerca viva como um retângulo verde. Para desenhar um retângulo:

public void paintComponent(Gráficos g) {

// x e y definem o canto superior esquerdo



g.fillRect(x,y,largura,altura);

}

Adicione uma cerca viva ao seu campo. O jogo final deve ser algo comoFigura 5-

11.

Figura 5-11. Nosso novo obstáculo de hedge no campo

Continuaremos a expandir este jogo ao longo do livro, mas sinta-se à vontade para fazer sua própria expansão agora. Brinque desenhando outras formas ou altere as cores dos elementos atuais. Odocumentação on-linepara a aula de Gráficos será útil.

1Depois de ter alguma experiência com conceitos básicos de orientação a objetos, você pode querer dar uma olhada em Design Patterns: Elements of Reusable Object-Oriented Software, de Erich Gamma et al. (Addison-Wesley). Este livro cataloga projetos úteis orientados a objetos que foram refinados ao longo dos anos. Muitos desses padrões aparecem no design das APIs Java.

2O char também recebe 0, mas geralmente é expresso como o caractere nulo, \0. Uma variável booleana obtém o padrão falso e os tipos de referência obtêm o padrão nulo.

3Não entramos em detalhes dessas listas de argumentos, mas se você estiver curioso e quiser ler um pouco por conta própria, pesquise on-line pela palavra-chave “varargs”

no idioma do programador.

4Acontece que a classe Math não pode ser instanciada. Ele contém apenas métodos estáticos. Tentar chamar new Math() resultaria em um erro do compilador.

5Ainda é possível escrever código em Java que retenha objetos para sempre (por acidente, temos certeza), consumindo cada vez mais memória. Isso não é realmente um vazamento, mas sim um acúmulo. Esse acúmulo em Java geralmente é mais fácil de rastrear do que um vazamento em C.

6Maven mudou suficientemente o cenário para gerenciamento de dependências em Java e até mesmo em outras linguagens baseadas em JVM, então agora você pode encontrar ferramentas comoGradle, que foram baseados no sucesso de Maven.

7Observe que a melhor maneira de projetar nossas calculadoras seria ter uma classe Calculator abstrata com duas subclasses separadas: IntegerCalculator e DecimalCalculator.

8O ornitorrinco é um mamífero que põe ovos muito incomum. Poderíamos substituir o comportamento reproduz() novamente para Platypus em sua própria subclasse de Mammal.

9Um método substituído em Java atua como um método virtual em C++.

Capítulo 6. Tratamento de Erros

Você sempre encontrará erros no mundo real. A maneira como você lida com eles ajuda a mostrar a qualidade do seu código.

Java tem suas raízes em sistemas embarcados – software executado em dispositivos especializados, como computadores portáteis, telefones celulares e torradeiras sofisticadas que hoje podemos considerar parte da Internet das Coisas (IoT). Nessas aplicações, é especialmente importante que os erros de software sejam tratados de forma robusta. A maioria dos usuários concordaria que é inaceitável que seu telefone trave regularmente ou que sua torrada (e talvez sua casa) queime porque algum software falhou. Dado que não podemos eliminar a possibilidade de erros de software, reconhecer e lidar metodicamente com erros no nível do aplicativo é um bom passo na direção certa.

Algumas linguagens deixam a responsabilidade de lidar com erros inteiramente com o programador. A linguagem em si não fornece nenhuma ajuda na identificação de tipos de erros e nenhuma ferramenta para lidar com eles

facilmente. Na linguagem C, por exemplo, as funções geralmente indicam uma falha retornando um valor “não razoável” (como o idiomático -1 ou um nulo). Como programador, você deve saber o que constitui um resultado ruim e o que isso significa. Muitas vezes é complicado contornar as limitações da passagem de valores de erro no caminho normal do fluxo de dados.1Um problema ainda pior é que certos tipos de erros podem ocorrer legitimamente em quase qualquer lugar, e é lento e caro testá-los explicitamente em todos os pontos do software.

Neste capítulo consideraremos como Java aborda o problema dos problemas.

Examinaremos a noção de exceções para ver como e por que elas ocorrem, bem como como e onde tratá-las. Também examinaremos erros e afirmações. Erros são problemas mais sérios que muitas vezes não podem ser corrigidos em tempo de execução, mas ainda podem ser registrados para depuração. As asserções são uma

forma popular de inocular seu código contra exceções ou erros, verificando antecipadamente se existem condições seguras.

Exceções

Java oferece uma solução elegante para ajudar o programador a resolver problemas comuns de codificação e tempo de execução por meio de exceções. (O tratamento de exceções Java é semelhante, mas não exatamente igual ao tratamento de exceções em C++.) Uma exceção indica uma condição incomum ou uma condição de erro. Quando ocorre um problema, o tempo de execução transfere o controle (ou “lança”) para uma

seção especialmente designada do seu código que pode manipular (ou “capturar”) a condição. Desta forma, o tratamento de erros é independente do fluxo normal do programa. Não precisamos de valores de retorno especiais para todos os nossos métodos; os erros são tratados por um mecanismo separado. Java pode passar o controle a uma longa distância de uma rotina profundamente aninhada e tratar erros em um único local quando isso for desejável, ou um erro pode ser tratado imediatamente em sua origem. Alguns métodos Java padrão ainda retornam um valor especial como -1, mas geralmente são limitados a situações em que esperar e manipular um valor especial é relativamente simples.2

Você deve especificar quaisquer exceções conhecidas que seus métodos possam lançar, e o compilador garante que os chamadores do método as tratem. Dessa forma, Java trata as informações sobre quais erros um método pode produzir com o mesmo nível de importância que seus argumentos e tipos de retorno. Você ainda pode decidir ignorar alguns erros, mas em Java você deve fazer isso explicitamente. (Discutiremos exceções e erros de tempo de execução, que não exigem essa declaração explícita, em instantes.)

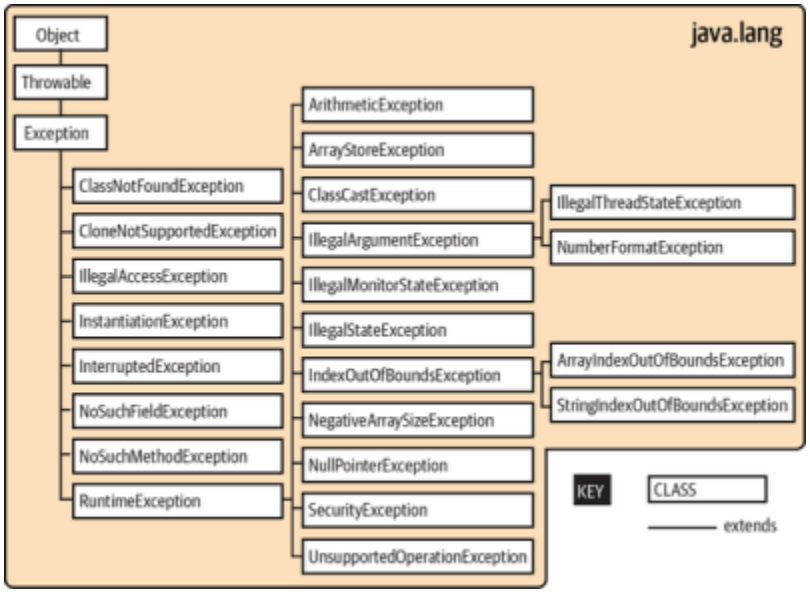
Exceções e classes de erro

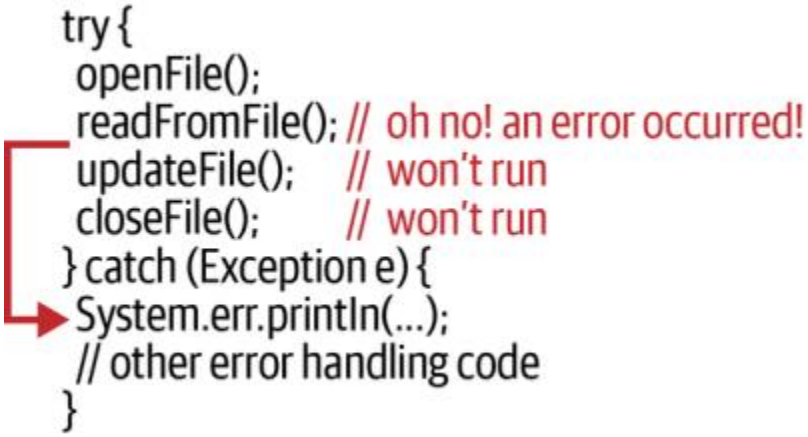
As exceções são representadas por instâncias da classe java.lang.Exception e suas subclasses. Subclasses de Exception podem conter informações especializadas (e possivelmente comportamento) para diferentes tipos de condições excepcionais. No entanto, mais frequentemente são simplesmente subclasses “lógicas” que servem apenas para identificar um novo tipo de exceção.Figura 6-1mostra as subclasses de Exception no

pacote java.lang. Deve dar uma ideia de como as exceções são organizadas. A maioria dos pacotes define seus próprios tipos de exceção, que geralmente são subclasses da própria Exception ou de sua importante subclasse RuntimeException, da qual falaremos em breve.

Por exemplo, vejamos outra classe de exceção importante: java.io.IOException. A classe IOException estende Exception e possui muitas subclasses próprias para problemas típicos de E/S, como FileNotFoundException. Observe como o nome da classe é explícito (e útil). Muitas exceções de rede estendem ainda mais IOException

— elas envolvem entrada e saída — mas seguindo as convenções, exceções como





MalformedURLException pertencem ao pacote java.net junto com outras classes de rede.

Figura 6-1. As subclasses java.lang.Exception

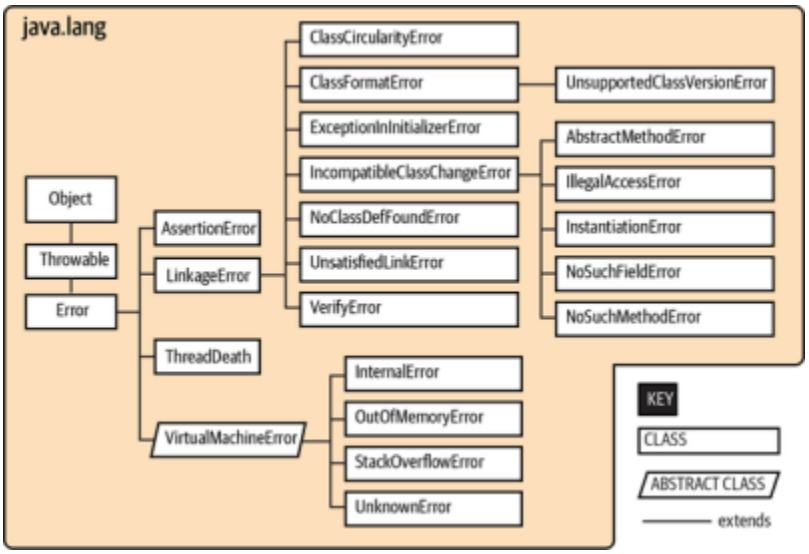
O tempo de execução cria um objeto Exception no ponto onde surge a condição de erro. Ele pode ser projetado para conter qualquer informação necessária para descrever a condição excepcional. Ele também inclui um rastreamento de pilha completo para depuração. Um rastreamento de pilha é a lista (ocasionalmente complicada) de todos os métodos chamados e a ordem em que foram chamados do método main() até o ponto em que a exceção foi lançada. (Veremos essas listas úteis com mais detalhes em“Traços de pilha”.) O objeto Exception é passado como um argumento para o bloco de código de manipulação, junto com o fluxo de controle. É

daí que vêm os termos throw e catch: o objeto Exception é lançado de um ponto no código e capturado por outro, onde a execução é retomada, conforme mostrado

emFigura 6-2.

Figura 6-2. O fluxo de controle quando ocorre uma exceção

Java também define a classe java.lang.Error para erros irrecuperáveis. As subclasses de Error no pacote java.lang são mostradas emFigura 6-3. Um tipo de erro notável é AssertionError, que é usado pela instrução Java assert (mais sobre essa instrução posteriormente neste capítulo) para indicar uma falha. Alguns outros pacotes definem suas próprias subclasses de Error, mas subclasses de Error são muito menos comuns (e menos úteis) que subclasses de Exception. Geralmente, você não precisará se preocupar com esses erros no seu código; eles têm como objetivo indicar problemas fatais ou erros de máquina virtual, que geralmente fazem com que o interpretador Java exiba uma mensagem e saia. Os projetistas de Java desencorajam ativamente os desenvolvedores de tentar detectar ou se recuperar desses erros porque eles deveriam indicar um bug fatal do programa, possivelmente na própria JVM, e não uma condição de rotina.



Tanto Exception quanto Error são subclasses de Throwable. A classe Throwable é a classe base para objetos que podem ser “lançados” com a instrução throw. Embora você mesmo possa estender tecnicamente Throwable, geralmente você deve estender apenas Exception, Error ou uma de suas subclasses se quiser criar seu próprio tipo throwable.

Figura 6-3. As subclasses java.lang.Error

Manipulação de exceção

Para capturar e tratar uma exceção _Italicized Textix, você agrupa blocos do seu código em instruções de proteção try/catch:

```
tentar {

readFromFile("foo");

//faz outras coisas no arquivo...

} catch (Exceção e) {

//Trata o erro

System.out.println("Exceção ao ler arquivo: " + e);

}
```

Neste exemplo, quaisquer exceções que ocorram no corpo da parte try da instrução são direcionadas para a cláusula catch para possível tratamento. A cláusula catch atua como um método; especifica o tipo de exceção que deseja tratar. Se for invocada, a cláusula recebe o objeto Exception como argumento. Aqui recebemos o objeto na variável e e o imprimimos junto com uma mensagem.

Podemos tentar isso nós mesmos. Lembre-se do programa simples para calcular o máximo denominador comum (MDC) usando o algoritmo de Euclides lá atrásCapítulo

4. Poderíamos aumentar esse programa para permitir que o usuário passe os dois números a e b como argumentos de linha de comando por meio do array args[] no método main(). No entanto, essa matriz é do tipo String. Se trapacearmos um pouco e roubarmos algum códigoAnalisando Números Primitivos”, podemos usar um método de análise de texto para transformar essas strings em valores int. No entanto, esse método de análise pode gerar uma exceção se não passarmos um número válido. Aqui está uma olhada em nossa nova classe Euclid2:

```
classe pública Euclides2 {

public static void main(String args[]) {

intuma = 2701;

interno b = 222;

// Só tentamos analisar argumentos se tivermos exatamente 2

if (args.comprimento == 2) {

tentar {

a = Integer.parseInt(args[0]);

b = Integer.parseInt(args[1]);

} catch (NumberFormatException nfe) {
```

```
System.err.println("Os argumentos não eram ambos
números."); System.err.println("Usando padrões.");

}

} outro {

System.err.print("Número errado de argumentos");

System.err.println(" esperado 2).");

System.err.println("Usando padrões.");

}

System.out.print("O MDC de " + a + " e " + b + " é ");
enquanto (b! = 0) {

se (a > b) {

uma = uma - b;

} outro {

b=b - uma;

}

}

System.out.println(a);

}

}
```

Observe que limitamos nosso try/catch apenas ao código potencialmente problemático. É comum ver vários blocos

try/catch diferentes em um método. Esse pequeno escopo nos permite adaptar melhor o código no bloco catch a quaisquer problemas que estejamos antecipando. Nesse caso, sabemos que podemos receber alguma entrada incorreta do usuário, então podemos verificar a

NumberFormatException (muito) específica e imprimir uma mensagem amigável para o usuário.

Se executarmos este programa a partir de uma janela de terminal ou usarmos a opção de argumentos de linha de comando em nosso IDE, como fizemos emFigura 2-10,

agora podemos encontrar o MDC de vários pares de números sem recompilar: $ javac ch06/examples/Euclid2.java

$ java ch06.examples.Euclid2

O GCD de 18 e 6 é 6

$ java ch06.examples.Euclid2 547832 2798

O GCD de 547832 e 2798 é 2

Mas se passarmos argumentos que não sejam numéricos, obteremos essa

NumberFormatException e veremos nossa mensagem de erro. Observe, entretanto, que nosso código se recupera normalmente e ainda fornece alguma saída. Esta recuperação é a essência do tratamento de erros:

$ java ch06.examples.Euclid2 maçãs laranjas

Os argumentos não eram ambos números.

Usando padrões.

O MDC de 2701 e 222 é 37

Uma instrução try pode ter várias cláusulas catch que especificam diferentes tipos (subclasses) de Exception:

tentar {

readFromFile("foo");

//faz qualquer outra coisa no arquivo

} catch (FileNotFoundException e) {

// Trata arquivo não encontrado

} catch (IOException e) {

//Trata erro de leitura

} catch (Exceção e) {

// Trata todos os outros erros

}

As cláusulas catch são avaliadas em ordem e Java escolhe a primeira correspondência atribuível. No máximo, uma cláusula catch é executada, o que significa que as exceções devem ser listadas da mais específica para a mais genérica. No exemplo anterior, prevemos que o método hipotético readFromFile() pode lançar dois tipos diferentes de exceções: uma para um arquivo não encontrado e outra para um erro de leitura mais geral. Talvez o arquivo exista, mas não temos permissão para abri-lo.

FileNotFoundException é uma subclasse de IOException, portanto, se tivéssemos trocado as duas primeiras

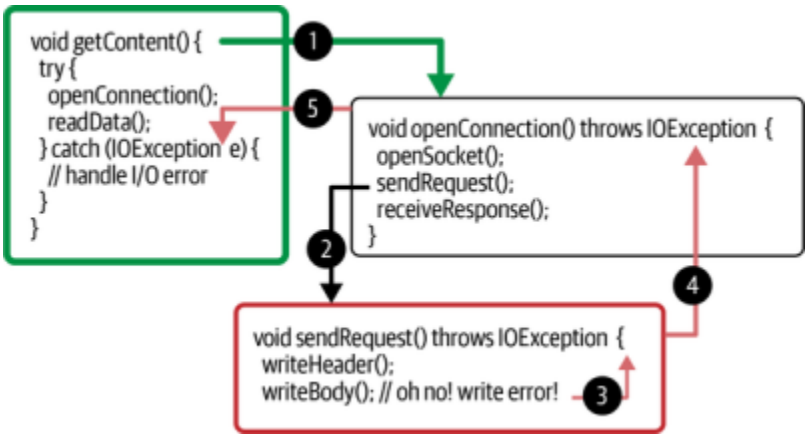
cláusulas catch, a cláusula IOException mais geral capturaria a exceção de arquivo ausente.

E se invertermos completamente a ordem das cláusulas catch? Você pode atribuir qualquer subclasse de Exception ao tipo pai Exception, para que essa cláusula capture todas as exceções. Se você usar um catch com tipo Exception, coloque-o sempre como a última cláusula possível. Ele atua como o caso padrão em uma instrução switch e lida com quaisquer possibilidades restantes.

Uma vantagem do esquema try/catch é que qualquer instrução no bloco try pode assumir que todas as instruções anteriores no bloco foram bem-sucedidas. Um problema não surgirá repentinamente porque você esqueceu de verificar o valor de retorno de um método. Se uma instrução anterior falhar, a execução salta imediatamente para uma cláusula catch; instruções posteriores dentro do try nunca são executadas.

Existe uma alternativa ao uso de múltiplas cláusulas catch. Você pode lidar com vários tipos de exceção discretos em uma única cláusula catch, usando a sintaxe ou (escrita usando a barra vertical, “|”):

```
tentar {

// lê da rede...

// escreve no arquivo..

} catch (ZipException | SSLException e) {
```



logException(e);

}

Usando este “|” sintaxe, recebemos os dois tipos de exceção na mesma cláusula catch.

Qual é o tipo real da variável e que estamos passando para nosso método log? O que podemos fazer com isso? Nesse caso, o tipo de e não será ZipException nem SSLException, mas IOException, que é o ancestral comum mais próximo (o tipo de classe pai mais próximo ao qual ambos podem ser atribuídos) para as duas exceções.

Em muitos casos, o tipo comum mais próximo entre os dois ou mais tipos de exceção de argumento pode ser simplesmente Exception, o pai de todos os tipos de exceção.

A diferença entre capturar esses tipos de exceção discretos com uma cláusula catch de vários tipos e simplesmente capturar o tipo de exceção pai comum é que estamos limitando nossa captura apenas a esses tipos de exceção especificamente enumerados.

Não capturaremos nenhum dos outros tipos de IOException. A combinação de cláusulas catch de vários tipos com a ordenação das cláusulas de tipos específicos para tipos amplos oferece grande flexibilidade no tratamento de exceções. Você pode consolidar a lógica de tratamento de erros quando apropriado e evitar a repetição de código. Há mais nuances nesse recurso e voltaremos a ele depois de discutirmos as exceções de "lançamento" e "relançamento".

Borbulhando

E se não tivéssemos capturado a exceção? Para onde teria ido? Bem, se não houver nenhuma instrução try/catch delimitadora, a exceção aparecerá no método em que foi originada (interrompendo a execução desse método) e será lançada desse método até seu chamador. Se esse método de chamada tiver uma cláusula try, o controle passa para a cláusula catch correspondente. Caso contrário, a exceção continuará se propagando na pilha de chamadas, de um método para seu chamador. A exceção borbulha até ser detectada ou até sair do topo do programa e finalizá-lo com uma mensagem de erro em tempo de execução. Às vezes há um pouco mais do que isso; o compilador pode forçá-lo a lidar com a exceção ao longo do caminho."Exceções

verificadas e não verificadas"fala sobre essa distinção com mais detalhes.

Vejamos outro exemplo. EmFigura 6-4, o método getContent() invoca o método openConnection() de dentro de uma instrução try/catch (etapa 1 na figura). Por sua vez, openConnection() invoca o método sendRequest() (etapa 2), que chama o método write() para enviar alguns dados.

Figura 6-4. Propagação de exceção

Nesta figura, a segunda chamada para write() lança uma IOException (etapa 3). Como sendRequest() não contém uma instrução try/catch para tratar a exceção, ela é lançada novamente a partir do ponto onde foi chamada no método openConnection() (etapa 4). Mas openConnection() também não captura a exceção, então ela é lançada mais uma vez (etapa 5). Finalmente, ele é capturado pela instrução try em getContent() e tratado por sua cláusula catch. Observe que cada método de lançamento deve declarar que pode lançar um tipo específico de exceção com uma cláusula throws. Discutiremos isso em“Exceções verificadas e não verificadas”.

Adicionar uma instrução try de alto nível no início do seu código também pode ajudar a lidar com erros que podem surgir de threads em segundo plano. Discutiremos os tópicos com muito mais detalhes emCapítulo 9, mas exceções não detectadas podem causar dores de cabeça de depuração em programas maiores e mais complexos.

Rastreamentos de pilha

Como uma exceção pode surgir bastante antes de ser capturada e tratada, precisamos encontrar uma maneira de determinar exatamente onde ela foi lançada. Também é importante saber como esse código problemático foi alcançado. Quais métodos chamaram quais outros métodos para chegar a esse ponto? Para depuração, todas as exceções podem despejar um rastreamento de pilha que lista seu método de origem e todas as chamadas de método aninhadas que elas fizeram para chegar lá. Mais comumente, o usuário vê um

rastreamento de pilha quando ele é impresso usando o método printStackTrace():

```
tentar {

// tarefa complexa e profundamente aninhada

} catch (Exceção e) {

// despeja informações sobre onde ocorreu a exceção

e.printStackTrace(System.err);

}
```

Esse rastreamento de pilha para uma exceção pode ser assim:

```
java.io.FileNotFoundException: meuarquivo.xml

em java.io.FileInputStream.<init>(FileInputStream.java)

em java.io.FileInputStream.<init>(FileInputStream.java)

em MyApplication.loadFile(MyApplication.java:137)

em MyApplication.main(MyApplication.java:5)
```

Este rastreamento de pilha indica que a linha 5 (em seu código-fonte) do método main() da classe MyApplication chamou o método loadFile(). O método loadFile() tentou construir um FileInputStream na linha 137, o que gerou

FileNotFoundException.

Depois que o rastreamento de pilha atinge as classes do sistema Java (como FileInputStream), os números das linhas podem ser perdidos. Isso também pode acontecer

se o código tiver sido otimizado. Normalmente, existe uma maneira de desativar temporariamente a otimização para encontrar os números exatos das linhas,

mas às vezes outras técnicas de depuração podem ser necessárias. Veremos muitas dessas técnicas posteriormente neste capítulo.

Os métodos da classe Throwable permitem recuperar as informações de rastreamento de pilha programaticamente, usando o método getStackTrace(). (Lembre-se de que Throwable é a classe pai de Exception e Error.) Este método retorna uma matriz de objetos StackTraceElement, cada um dos quais representa uma chamada de método na pilha. Você pode solicitar detalhes a StackTraceElement sobre a localização desse método usando os métodos getFileName(), getClassName(), getMethodName() e getLineNumber(). O elemento zero da matriz é o topo da pilha, a linha final do código que causou a exceção; os elementos subsequentes recuam uma chamada de método cada até que o método main() original seja alcançado no último elemento.

Exceções verificadas e não verificadas

Mencionamos anteriormente que Java nos obriga a ser explícitos sobre nosso tratamento de erros, mas não é necessário exigir que todo tipo concebível de erro seja tratado explicitamente em todas as situações. As exceções Java são, portanto, divididas em duas categorias: verificadas e não verificadas. A maioria das exceções em nível de aplicação são verificadas, o que significa que qualquer método que lance uma deve declará-la com uma cláusula especial em sua definição, como esta:

```
void readFile(String s) lança IOException,
InterruptedException {

// faz algum trabalho de E/S, talvez usando threads para
processamento em segundo plano

}
```

Nosso método readFile() antecipa o lançamento de dois tipos de exceções: gerando-as ele mesmo (como discutiremos em "Lançando exceções") e ignorando aqueles que ocorrem dentro dele. Por enquanto, tudo que você precisa saber é que os métodos devem declarar as exceções verificadas que podem lançar ou permitir que sejam lançadas.

A cláusula throws informa ao compilador que um método é uma possível fonte desse tipo de exceção verificada e que qualquer pessoa que chame esse método deve estar preparada para lidar com isso. O chamador deve então usar um bloco try/catch para lidar com isso ou, por sua vez, declarar que pode lançar a exceção de si mesmo.

Por outro lado, as exceções que são subclasses da classe java.lang.RuntimeException ou da classe java.lang.Error são desmarcadas. VerFigura 6-1para as subclasses de RuntimeException. Não é um erro em tempo de compilação ignorar a possibilidade dessas exceções; os métodos também não precisam declarar que podem lançá-los. Em todos os outros aspectos, as exceções não verificadas se comportam da mesma forma que outras exceções. Você é livre para capturá-los se desejar, mas neste caso não é obrigado a fazê-lo.

As exceções verificadas destinam-se a cobrir problemas no nível do aplicativo, como arquivos ausentes e hosts de rede indisponíveis. Como bons programadores (e

cidadãos íntegros), deveríamos projetar software para se recuperar normalmente

desses tipos de condições. As exceções não verificadas destinam-se a problemas no nível do sistema, como “índice de array fora dos limites”. Embora possam indicar erros de programação no nível do aplicativo, eles podem ocorrer em praticamente qualquer lugar. Felizmente, por serem exceções não verificadas, você não precisa agrupar cada uma de suas operações de array em uma instrução try/catch ou declarar todos os métodos de chamada como uma fonte potencial delas.

Resumindo, as exceções verificadas são problemas que um aplicativo razoável deve tentar resolver normalmente. Exceções não verificadas (exceções ou erros de tempo de execução) são problemas dos quais normalmente não esperaríamos que nosso software se recuperasse, mas você pode fornecer uma mensagem educada e, esperançosamente, informativa ao usuário sobre o que aconteceu. Tipos de erros, como erros de “falta de memória”, são condições das quais normalmente não podemos nos recuperar.

Lançando exceções

Podemos lançar nossas próprias exceções — sejam instâncias de Exception, uma de suas subclasses existentes ou nossas próprias classes de exceção especializadas. Tudo o que precisamos fazer é criar uma instância da classe de exceção apropriada e lançá-la com a instrução throw:

lançar nova IOException();

A execução é interrompida e transferida para a instrução try/catch mais próxima que possa manipular o tipo de

exceção. Observe que não colocamos a nova exceção em uma variável. Não faz muito sentido manter uma referência ao objeto IOException que criamos aqui, pois a instrução throw interrompe imediatamente o fluxo atual em nosso código.

Um construtor alternativo para exceções nos permite especificar uma string com uma mensagem de erro:

throw new IOException("Manchas solares!");

Você pode recuperar essa string usando o método getMessage() do objeto Exception.

Muitas vezes, porém, você pode simplesmente imprimir o próprio objeto de exceção para obter a mensagem e o rastreamento de pilha.

Por convenção, todos os tipos de Exception possuem um construtor String como este.

As “manchas solares!” a mensagem é extravagante, mas não muito útil. Normalmente, você lançará uma subclasse mais específica de Exception, que captura detalhes extras sobre a falha ou pelo menos fornece uma explicação de string mais útil. Aqui está outro exemplo:

public void checkRead (String s) lança SecurityException
{

// ...

if (novo(s) arquivo(s).isAbsolute() || (s.indexOf("..") != -1)) lançar nova SecurityException(

"Acesso ao arquivo: "+ s +" negado.");

```
// ...

}
```

Neste código, implementamos parcialmente um método para verificar um caminho ilegal. Se encontrarmos uma, lançamos uma SecurityException com algumas informações sobre a transgressão.

É claro que poderíamos incluir qualquer outra informação que seja útil em nossas próprias subclasses especializadas de Exception. Muitas vezes, porém, apenas ter um novo tipo de exceção é suficiente porque é suficiente para ajudar a direcionar o fluxo de controle. Por exemplo, se estivermos construindo um programa para ler e analisar o conteúdo de uma página web, podemos querer criar nosso próprio tipo de exceção para indicar uma falha específica:

```
class ParseException estende exceção {

private int lineNumber;

ParseException() {

super();

this.lineNumber = -1;

}

ParseException(String desc, int lineNumber) {

super(desc);

this.lineNumber = lineNumber;

}
```

```
public int getLineNumber() {

retornar número da linha;

}

}
```

Ver“Construtores”para obter uma descrição completa das classes e construtores de classes. O corpo da nossa classe Exception aqui simplesmente permite que uma ParseException seja criada da maneira convencional (genericamente ou com um pouco de informação extra). Agora que temos nosso novo tipo de exceção, podemos nos proteger contra qualquer conteúdo mal formatado como este:

```
// Obtém alguma entrada de um arquivo e analisa-a

tentar {

parseStream(entrada);

} catch (ParseException pe) {

// Entrada incorreta... Podemos até dizer qual linha estava incorreta!

System.err.println("Entrada incorreta na linha " + pe.getLineNumber());

} catch (IOException ioe) {

// Outro problema de comunicação de baixo nível

}
```

Mesmo sem informações especiais, como o número da linha onde nossa entrada causou um problema, nossa exceção personalizada nos permite distinguir um erro de análise de algum outro erro de E/S no mesmo pedaço de código.

Encadeando e lançando novamente exceções

Às vezes, você desejará realizar alguma ação com base em uma exceção e, em seguida, inverter e lançar uma nova exceção em seu lugar. Isso é comum ao construir estruturas onde exceções detalhadas de baixo nível são tratadas e representadas por exceções de nível superior que podem ser gerenciadas com mais facilidade. Por exemplo, você pode querer capturar uma IOException em um pacote de

comunicações, possivelmente realizar alguma limpeza e, em seguida, lançar sua própria exceção de nível superior, talvez algo como LostServerConnection.

Você pode fazer isso da maneira óbvia, simplesmente capturando a exceção e lançando uma nova, mas então você perde informações importantes, incluindo o rastreamento de pilha da exceção “causal” original. Para lidar com isso, você pode usar a técnica de encadeamento de exceções. Isso significa que você inclui a exceção causal na nova exceção lançada. Java tem suporte explícito para encadeamento de exceções.

A classe base Exception pode ser construída com uma exceção como argumento ou a mensagem String padrão e uma exceção:

throw new Exception("Aqui está a história...", causalException); Você pode obter acesso à exceção agrupada posteriormente com o método getCause().

Mais importante ainda, Java imprime automaticamente ambas as exceções e seus respectivos rastreamentos de pilha se você imprimir a exceção ou se ela for mostrada ao usuário.

Você pode adicionar esse tipo de construtor às suas próprias subclasses de exceção (delegando ao construtor pai). Você também pode aproveitar esse padrão usando o método Throwable initCause() para definir explicitamente a exceção causal após construir sua própria exceção e antes de lançá-la:

tentar {

// ...

} catch (causa de IOException) {

Exceção e =

new IOException("O que temos aqui é uma falha na comunicação..."); e.initCause(causa);

jogue e;

}

Às vezes, basta simplesmente fazer algum registro ou realizar alguma ação intermediária e, em seguida, lançar novamente a exceção original:

tentar {

// ...

} catch (causa de IOException) {

log(causa); //Registre-o

lançar causa; //joga novamente

}

Você vê esse padrão surgir quando há exceções que não contêm informações suficientes para serem tratadas localmente. Você pode fazer algo com as informações disponíveis (como imprimir uma mensagem de erro para ajudar durante a depuração), mas não tem informações suficientes para se recuperar do problema.

Você tem que passar a exceção e esperar que algum método de chamada com mais recursos saiba o que fazer.

tente rastejar

A instrução try impõe uma condição às instruções que ela protege. Diz que se ocorrer uma exceção dentro dele, as instruções restantes serão abandonadas. Isto tem consequências para a inicialização da variável local. Se o compilador não puder determinar se uma atribuição de variável local colocada dentro de um bloco try/catch acontecerá, ele não nos permitirá usar a variável. Por exemplo:

void meuMetodo() {

int foo;

tentar {

foo = getResultados();

}

pegar (Exceção e) {

// trata nossa exceção ...

}

barra interna = foo; // Erro em tempo de compilação: foo pode não ter sid o inicializado

}

Neste exemplo, não podemos usar foo no local indicado porque existe a possibilidade de nunca ter sido atribuído um valor. Uma opção é mover a atribuição for bar para dentro da instrução try:

tentar {

foo = getResultados();

barra interna = foo; // Ok porque só chegamos aqui

//se a tarefa anterior for bem-sucedida

}

pegar (Exceção e) {

// trata nossa exceção ...

}

Às vezes isso funciona muito bem. No entanto, agora temos o mesmo problema se quisermos usar bar posteriormente em myMethod(). Se não tomarmos cuidado, podemos acabar colocando tudo na instrução try. A situação muda, entretanto, se transferirmos o controle para fora do método na cláusula catch:

tentar {

```
foo = getResultados();

}

pegar (Exceção e) {

// registra nossa exceção ou mostra ao usuário uma mensagem de aviso

retornar;

}

barra interna = foo; // Ok porque só chegamos aqui

//se o bloco try anterior tiver sucesso
```

O compilador é inteligente o suficiente para saber que se um erro tivesse ocorrido na cláusula try, não teríamos alcançado a atribuição da barra, então isso nos permite fazer referência a foo. Seu código pode ter requisitos diferentes; só queremos que você esteja ciente das opções.

A cláusula final

E se tivermos algo importante para fazer antes de sairmos do nosso método de uma das cláusulas catch? Para evitar a duplicação do código em cada ramificação catch e tornar a limpeza mais explícita, você pode usar a cláusula finalmente. Uma cláusula finalmente pode ser adicionada após um bloco try e suas cláusulas catch associadas.

Qualquer instrução no corpo da cláusula final tem garantia de execução, não importa como o controle sai

do corpo try, independentemente de uma exceção ser lançada ou não:

```
tentar {

// Faça algo aqui

}

catch (FileNotFoundException e) {

// trata de um arquivo ausente ...

}

pegar (IOException e) {

// lidar com outros problemas de arquivo ...

}

pegar (Exceção e) {

// eek, resolva problemas ainda maiores...

}

finalmente {

// Qualquer limpeza aqui é sempre executada

}
```

Neste exemplo, as instruções no ponto de limpeza são executadas eventualmente, não importando como o controle sai da tentativa. Se o controle for transferido para uma das cláusulas catch, as instruções em finalmente serão executadas após a conclusão da

captura. Se nenhuma das cláusulas catch tratar a exceção, as instruções finalmente serão executadas antes que a exceção se propague para o próximo nível.

Mesmo que as instruções na tentativa sejam executadas de forma limpa, ou se executarmos um return, break ou continue, as instruções na cláusula finalmente ainda

serão executadas. Para garantir que algumas operações serão executadas, podemos até usar try e finalmente sem nenhuma cláusula catch:

```
tentar {

// Faça algo aqui que possa causar uma exceção

retornar;

} finalmente {

System.out.println("Uau!");

}
```

Exceções que ocorrem em uma cláusula catch ou finalmente são tratadas normalmente; a busca por um try/catch envolvente começa fora da instrução try ofensiva, após o finalmente ter sido executado.

tente com recursos

Um uso comum da cláusula final é garantir que os recursos usados em uma cláusula try sejam limpos, não importa como o código sai do bloco. Considere abrir um soquete de rede (mais sobre isso emCapítulo 13):

```
tentar {
```

```
Soquete meia = new Soquete(...);

//trabalha com o soquete

} catch(IOException e) {

// lidar com nosso problema de rede ...

} finalmente {

if (meia! = nulo) { meia.close(); }

}
```

O que queremos dizer com “limpar” aqui é desalocar recursos caros ou fechar conexões com coisas como arquivos, soquetes de rede ou bancos de dados. Em alguns casos, esses recursos podem ser limpos por conta própria, eventualmente, à medida que o Java recupera o lixo, mas, na melhor das hipóteses, isso aconteceria em um momento desconhecido no futuro. Na pior das hipóteses, a limpeza pode nunca acontecer ou não acontecer antes que você fique sem recursos. Você deve sempre se proteger contra essas situações.

No mundo real, porém, existem dois problemas em manter o controle dessa alocação de recursos. Primeiro, é necessário trabalho extra para realizar um padrão de limpeza adequado em todo o seu código, incluindo coisas importantes como verificações de nulos, conforme mostrado em nosso exemplo hipotético. Segundo, se você estiver manipulando vários recursos em um único bloco finalmente, você terá a possibilidade de seu próprio código de limpeza lançar uma exceção (por exemplo, em close()) e deixar o trabalho inacabado.

A forma “tentar com recursos” da cláusula try pode ajudar. Nesse formato estendido, você coloca uma ou mais instruções de inicialização de recursos entre parênteses após a palavra-chave try. Esses recursos serão automaticamente “fechados” para você quando o controle sair do bloco try:

```
tentar (

Soquete meia = novo Soquete("192.168.100.1", 80);

Arquivo FileWriter = new FileWriter("foo");

)

{

//trabalha com meia e arquivo

} catch (IOException e) {

// ...

}

// Tanto o sock quanto o arquivo foram limpos neste ponto
```

Neste exemplo, inicializamos um objeto Socket e um objeto FileWriter na cláusula try-with-resources e podemos usá-los no corpo da instrução try. Quando o controle sai da instrução try, após a conclusão bem-sucedida ou devido a uma exceção, o Java fecha automaticamente ambos os recursos chamando seus respectivos métodos close(). Java fecha esses recursos na ordem inversa em que você os construiu, para que

você possa acomodar quaisquer dependências entre eles.

Java suporta esse comportamento para qualquer classe que implemente a interface AutoCloseable (o que, na contagem atual, mais de cem classes integradas diferentes fazem). O método close() desta interface é prescrito para liberar todos os recursos associados ao objeto, e você também pode implementar isso facilmente em suas próprias classes. Agora, ao usar try-with-resources, não precisamos adicionar nenhum código especificamente para fechar o arquivo ou soquete; isso é feito para nós automaticamente.

Outro problema que o try-with-resources resolve é a situação incômoda em que uma exceção é lançada durante uma operação fechada. Olhando para o exemplo anterior em que usamos uma cláusula finalmente para fazer nossa limpeza, se uma exceção tivesse sido levantada pelo método close(), a nova exceção teria sido lançada naquele ponto, abandonando completamente a exceção original do corpo da cláusula try. Mas o try-with-resources preserva a exceção original. Se ocorrer uma exceção no corpo do try e uma ou mais exceções forem levantadas durante as operações subsequentes de fechamento automático, será a exceção original do corpo do try que irá aparecer para o chamador. Vejamos um exemplo:

```
tentar (

// exceção potencial #3

Soquete meia = novo Soquete("192.168.100.1", 80);

// exceção potencial #2
```

Arquivo FileWriter = new FileWriter("foo");

)

{

// trabalha com meia e arquivo // exceção potencial #1

}

Depois que a tentativa for iniciada, se ocorrer uma exceção no ponto de exceção nº 1, o Java tentará fechar ambos os recursos na ordem inversa, levando a possíveis exceções nos locais nº 2 e nº 3. Neste caso, o código de chamada ainda receberá a

exceção nº 1. Entretanto, as exceções 2 e 3 não foram perdidas; eles são meramente

“suprimidos”. Você pode recuperá-los por meio do método getSuppressed() da exceção lançada ao chamador. Este método retorna uma matriz de todas as exceções suprimidas.

Problemas de desempenho

Devido à forma como o Java VM é implementado, o uso de um bloco try para se proteger contra o lançamento de uma exceção é gratuito, o que significa que não adiciona nenhuma sobrecarga à execução do seu código. No entanto, lançar uma exceção não é gratuito. Quando uma exceção é lançada, o Java precisa localizar o bloco try/catch apropriado e executar outras atividades demoradas em tempo de execução.

É por isso que você deve lançar exceções apenas em circunstâncias verdadeiramente

“excepcionais” e evitar usá-las para condições esperadas, especialmente quando o desempenho for um problema. Por exemplo, se você tiver um loop, pode ser melhor realizar um pequeno teste em cada passagem e evitar um bloco try em vez de lançar uma exceção várias vezes durante a execução do loop. Por outro lado, se a exceção for lançada apenas uma vez em um zilhão de vezes, você pode querer eliminar a sobrecarga de suas pequenas verificações e não se preocupar com o custo de lançar essa exceção muito rara. A regra geral deve ser que as exceções sejam usadas para situações anormais, e não para condições rotineiras ou esperadas (como o fim de um arquivo ou a falta de um recurso da web).

Asserções

Java suporta asserções como outro mecanismo para validar o estado do seu programa.

Uma asserção é um simples teste de aprovação/reprovação de alguma condição, executado enquanto seu aplicativo está em execução. Você pode usar asserções para

“verificar a integridade” do seu código. As asserções são distintas de outros tipos de testes porque verificam condições que nunca devem ser violadas em um nível lógico: se a asserção falhar, significa que algum código que você escreveu não está fazendo seu trabalho e o aplicativo geralmente para com uma mensagem de erro apropriada.

As asserções são suportadas diretamente pela linguagem Java e podem ser ativadas ou desativadas em tempo de execução para remover qualquer penalidade de

desempenho associada à inclusão delas em seu código.

Usar asserções para testar o comportamento correto do seu aplicativo é uma técnica simples, mas poderosa, para garantir a qualidade do software. Ele preenche uma lacuna entre os aspectos do software que o compilador pode verificar

automaticamente e aqueles geralmente verificados por “testes unitários” ou testes humanos. As asserções testam suas próprias suposições sobre o comportamento do programa e transformam essas suposições em garantias (pelo menos enquanto as asserções são ativadas).

Se você já programou antes, pode ter visto algo como o seguinte:3

se (!condição)

lançar novo AssertionError("erro fatal: 42");

Uma asserção em Java é equivalente a este exemplo, mas você usa a palavra-chave assert. É necessária uma condição booleana e um valor de expressão opcional. Se a asserção falhar, um AssertionError será lançado, o que geralmente faz com que o Java saia do aplicativo. A ideia por trás do resgate é que uma falha de asserção revela uma falha lógica em seu código – e é sua responsabilidade como programador encontrar e corrigir essa falha.

A expressão opcional pode ser avaliada como um valor primitivo ou um tipo de objeto.

De qualquer forma, seu único propósito é ser transformado em uma string e mostrado ao usuário se a asserção falhar. Na maioria das vezes você usará uma

mensagem de string explicitamente. aqui estão alguns exemplos:

afirmar falso;

afirmar (array.length > min);

assert a > 0 : a // mostra o valor de a para o usuário

assert foo != null : "foo é nulo!" // mostra a mensagem "foo is null!" ao usuário

Em caso de falha, as duas primeiras asserções imprimem apenas uma mensagem genérica. O terceiro imprime o valor de a e o último imprime foo como nulo!

mensagem.

O importante sobre as asserções não é apenas que elas sejam mais concisas do que a condição if equivalente, mas que você possa ativá-las ou desativá-las ao executar o aplicativo. Desabilitar asserções significa que suas condições de teste nem mesmo são avaliadas, portanto não há penalidade de desempenho por incluí-las em seu código (além, talvez, de ainda consumir um pouco de espaço nos arquivos de classe quando são carregadas).

Habilitando e desabilitando asserções

Você ativa ou desativa as asserções em tempo de execução. Quando desabilitadas, as asserções ainda existem nos arquivos de classe, mas não são executadas e não consomem tempo de CPU. Você pode ativar e desativar asserções para um aplicativo inteiro, pacote por pacote ou até mesmo classe por classe. Lembre-se de que as asserções devem ser verificações de sanidade para você durante o desenvolvimento.

Geralmente, eles não devem ser vistos pelos usuários finais. Usá-los enquanto você trabalha em seu projeto, mas desativá-los para “produção” é uma tática comum.

Por padrão, as asserções estão desativadas em Java. Para habilitá-los em seu código, use o sinalizador de comando java -ea ou -enableassertions:

% java -ea MeuAplicativo

Para ativar asserções para uma classe específica, anexe o nome da classe:

% java -ea:com.oreilly.examples.Myclass MeuApplication

Para ativar asserções apenas para pacotes específicos, anexe o nome do pacote com reticências à direita (...):

% java -ea:com.oreilly.examples... MeuAplicativo

Quando você habilita asserções para um pacote, Java também habilita todos os nomes de pacotes subordinados (por exemplo, com.oreilly.examples.text). No entanto, você pode ser mais seletivo usando o sinalizador -da ou -disableassertions correspondente para negar pacotes ou classes individuais. Você pode combinar tudo isso para obter agrupamentos arbitrários, como este:

% java -ea:com.oreilly.examples... \

-da:com.oreilly.examples.text \

-ea:com.oreilly.examples.text.MonkeyTypewriters\

Minha aplicação

Este exemplo permite asserções para o pacote com.oreilly.examples como um todo, exclui o pacote com.oreilly.examples.text, mas depois ativa exceções para a classe MonkeyTypewriters nesse pacote.

Usando Asserções

Uma asserção impõe uma regra sobre algo que deveria ser estável em seu código e, de outra forma, não seria verificado. Você pode usar uma asserção para maior segurança em qualquer lugar onde desejar verificar suas suposições sobre o comportamento do programa que o compilador pode não ser capaz de verificar.

Uma situação comum que clama por uma afirmação é testar múltiplas condições ou valores onde um sempre deve ser encontrado. Nesse caso, uma afirmação com falha como padrão ou comportamento de “falha” indica que o código está quebrado. Por exemplo, suponha que temos um valor chamado direção que deve sempre conter uma de duas constantes, ESQUERDA ou DIREITA:

if (direção == ESQUERDA)

vá para esquerda();

senão if (direção == DIREITA)

Vire a direita()

outro

assert false : "má direção";

O mesmo se aplica ao caso padrão de um switch:

mudar (direção) {

caso ESQUERDA:

vá para esquerda();

quebrar;

caso DIREITO:

Vire a direita();

quebrar;

padrão:

afirmar falso;

}

Em geral, você não deve usar asserções para verificar a validade dos argumentos dos métodos. Você deseja que esse comportamento de validação faça parte do seu aplicativo, e não apenas um teste de controle de qualidade que pode ser desativado.

Os métodos exigem entradas válidas como parte de suas pré-condições, e você geralmente deve lançar uma exceção se alguma pré-condição não for atendida. O uso de exceções eleva as pré-condições a parte do “contrato” do método com o usuário. No entanto, verificar a exatidão dos resultados dos seus métodos com asserções antes de retorná-los pode ser útil. Essas verificações de finalização são chamadas de pós-condições.

Às vezes, determinar o que é ou não uma pré-condição depende do seu ponto de vista.

Por exemplo, quando um método é usado internamente dentro de uma classe, suas pré-condições já podem ser

garantidas pelos métodos que o chamam. Os métodos públicos da classe provavelmente deveriam lançar exceções quando suas pré-condições forem violadas, mas um método privado pode usar asserções porque seus chamadores são sempre códigos intimamente relacionados que devem obedecer ao comportamento correto.

Exceções do mundo real

A adoção de exceções pelo Java como uma técnica de tratamento de erros torna muito mais simples para os desenvolvedores escrever código robusto. O compilador força você a pensar antecipadamente nas exceções verificadas. Exceções não verificadas certamente aparecerão, mas as asserções podem ajudá-lo a ficar atento a esses problemas de tempo de execução e, com sorte, evitar travamentos.

O recurso try-with-resources torna ainda mais simples para os desenvolvedores manter seu código limpo e “fazer a coisa certa” ao interagir com recursos limitados do sistema, como arquivos e conexões de rede. Como observamos no início do capítulo, outras línguas certamente possuem facilidades ou costumes para lidar com esses problemas. Java, como linguagem, trabalha muito para ajudá-lo a considerar cuidadosamente os problemas que podem surgir em seu código. Quanto mais você trabalhar para resolver esses problemas, mais estável será seu aplicativo (e mais satisfeitos seus usuários).

Muitos de nossos exemplos até agora foram diretos e não exigiram nenhuma verificação sofisticada de erros. Tenha certeza de que exploraremos códigos mais interessantes com muitas coisas que merecem tratamento de

exceções. Os capítulos posteriores cobrirão tópicos como programação multithread e redes. Esses tópicos estão repletos de situações que podem dar errado em tempo de execução, como um grande cálculo descontrolado ou uma queda na conexão WiFi. Perdoe o trocadilho, mas em breve você tentará todos esses novos truques de exceção!

Perguntas de revisão

1. Qual instrução você deve usar para gerenciar possíveis exceções em seu código?

2. Quais exceções o compilador exige que você manipule ou lance?

3. Onde você coloca qualquer código de limpeza que deseja sempre executar após usar alguns recursos em um bloco try?

4. As asserções têm uma grande penalidade de desempenho quando são

desativadas?

Exercícios de código

1. O programa Pause.java na pasta ch06/exercises não será compilado. Ele usa um método, Thread.sleep(), para pausar o programa por cinco segundos. Esse método sleep() pode lançar uma exceção verificada. Corrija o programa para que ele seja compilado e executado. (Veremos mais threads e Thread.sleep()

emCapítulo 9.)

2. Os exercícios incluem outra variação do nosso programa “Hello, World”

deCapítulo 2chamado OláZero. Use uma asserção para garantir que as coordenadas x e y iniciais da mensagem gráfica sejam maiores que zero.

Tente executar o programa e ativar as asserções. O que acontece se você atribuir um número negativo a uma das coordenadas? Execute o programa novamente, mas deixe as asserções desabilitadas. (Lembre-se de que

“desativado” é o comportamento padrão, portanto não os habilite.) O que acontece neste caso?

Exercícios Avançados

1. Vamos fingir que um máximo denominador comum (MDC) de 1 é uma condição de erro que precisamos sinalizar. Crie uma nova classe, Euclid3 , que fará o trabalho normal de encontrar o GCD, mas lançará uma exceção se o

denominador comum for 1. (Sinta-se à vontade para começar copiando

qualquer uma de suas outras classes euclidianas.) Crie uma classe de exceção personalizada chamada GCDException que armazena o par de números

incorreto como detalhes da exceção.

Modifique Euclid3 para testar um GCD de 1 e lance sua nova GCDException se esse for o resultado. (Fornecer dois números primos é uma maneira rápida de garantir um resultado de 1.)

Antes de adicionar qualquer código de tratamento de exceções, tente compilar.

O javac avisou você sobre a exceção? Deveria! Vá em frente e adicione um protetor try/catch ou edite a definição de main() para lançar sua exceção. Se você lidar com a nova exceção, certifique-se de imprimir uma bela mensagem de erro para o usuário que inclua os números “ruins” da exceção capturada.

1As instruções setjmp() e longjmp() um tanto obscuras em C podem salvar um ponto na execução do código e posteriormente retornar a ele incondicionalmente de um local profundamente enterrado. Num sentido limitado, esta é a funcionalidade das exceções em Java que exploramos neste capítulo.

2Por exemplo, o método getHeight() da classe AWT Image retorna -1 se a altura ainda não for conhecida. Nenhum erro ocorreu; a altura estará disponível assim que a imagem for carregada. Nessa situação, lançar uma exceção seria excessivo e poderia impactar o desempenho.

3Se você fez alguma programação, esperamos que suas mensagens de erro não sejam tão opacas! Quanto mais úteis e explicativas forem suas mensagens, melhor.

Capítulo 7. Coleções e Genéricos

À medida que começarmos a usar nosso crescente conhecimento de objetos para lidar com problemas cada vez mais interessantes, uma questão recorrente surgirá. Como armazenamos os dados que manipulamos para resolver esses problemas?

Definitivamente usaremos variáveis de todos os tipos diferentes, mas também precisaremos de opções de armazenamento maiores e mais sofisticadas. Os arrays que discutimos anteriormente "Matrizes" são um começo, mas os arrays têm algumas limitações. Neste capítulo veremos como obter acesso eficiente e flexível a grandes quantidades de dados usando a ideia de coleções do Java. Veremos também como lidar com os vários tipos de dados que queremos armazenar nesses grandes contêineres, assim como fazemos com valores individuais em variáveis. É aí que entram os genéricos. Chegaremos àqueles em "Limitações de tipo".

Coleções

Coleções são estruturas de dados fundamentais para todos os tipos de programação.

Sempre que precisamos nos referir a um grupo de objetos, temos algum tipo de coleção. No nível da linguagem principal, Java oferece suporte a coleções na forma de arrays. Mas os arrays são estáticos e, por terem comprimento fixo, são inadequados para grupos de coisas que aumentam e diminuem ao longo da vida útil de um aplicativo. Matrizes também são ruins para representar relacionamentos abstratos entre objetos. No início, a plataforma Java tinha apenas duas classes básicas para atender a essas necessidades: a classe java.util.Vector, que representa uma lista dinâmica de objetos, e a classe java.util.Hashtable, que contém um mapa de chaves/

pares de valores. Hoje, Java tem uma abordagem mais abrangente chamada estrutura de coleções. A estrutura padroniza como você trabalha com uma variedade de coleções. As classes mais antigas ainda existem, mas

foram adaptadas à estrutura (com algumas excentricidades) e geralmente não são mais utilizadas.

Embora conceitualmente simples, as coleções são uma das partes mais poderosas de qualquer linguagem de programação. Eles implementam estruturas de dados que estão no centro do gerenciamento de problemas complexos. Grande parte da ciência da computação básica é dedicada a descrever as maneiras mais eficientes de implementar certos tipos de algoritmos em coleções. (Como você encontra algo rapidamente em uma coleção grande? Como você classifica itens em uma coleção?

Como você adiciona ou remove itens com eficiência?) Ter essas ferramentas à sua disposição e entender como usá-las pode tornar seu código muito menor e mais rápido. Também pode evitar que você reinvente a roda.

A estrutura de coleções original tinha duas desvantagens principais. A primeira era que as coleções não eram necessariamente digitadas e funcionavam apenas com objetos indiferenciados em vez de tipos específicos como datas e strings. Isso significava que você tinha que realizar uma conversão de tipo toda vez que retirava um objeto de uma coleção. Isso foi contra a segurança do tipo em tempo de compilação do Java. Mas, na prática, isso era menos um problema do que simplesmente complicado e tedioso. A segunda questão era que, por razões práticas, as coleções só podiam funcionar com objetos e não com tipos primitivos. Isso significava que sempre que você quisesse colocar um número ou outro tipo primitivo em uma coleção, você teria que armazená-lo primeiro em uma classe wrapper e descompactá-lo mais tarde, ao recuperá-lo. Essa combinação de fatores

tornou o código que trabalha com coleções menos legível e mais perigoso.

Tipos genéricos (novamente, mais sobre isso em“Limitações de tipo”) possibilitam que coleções verdadeiramente seguras estejam sob o controle do programador.

Juntamente com os genéricos, o autoboxing e o unboxing de tipos primitivos significam que geralmente você pode tratar objetos e primitivos como iguais no que diz respeito a coleções. A combinação desses novos recursos adiciona um pouco de segurança e pode reduzir significativamente a quantidade de código que você escreve.

Como veremos, todas as classes de coleções agora aproveitam esses recursos.

A estrutura de coleções é baseada em algumas interfaces do pacote java.util. Essas interfaces são divididas em duas hierarquias. A primeira hierarquia desce da interface Collection. Esta interface (e seus descendentes) representa um contêiner que contém outros objetos. A segunda hierarquia separada é baseada na interface Map, outro contêiner que representa um grupo de pares chave/valor onde a chave pode ser usada para recuperar o valor de maneira eficiente.

A interface da coleção

A mãe de todas as coleções é uma interface apropriadamente chamada Collection.

Serve como um contêiner que contém outros objetos, seus elementos. Não especifica exatamente como os objetos são organizados; não diz, por exemplo, se objetos

duplicados são permitidos ou se os objetos são ordenados de alguma forma. Esses tipos de detalhes são deixados para interfaces filhas ou classes de implementação. No entanto, a interface Collection define algumas operações básicas comuns a todas as coleções:

adição booleana pública (elemento)

Adiciona o objeto fornecido a esta coleção. Se a operação for bem-sucedida, este método retornará verdadeiro. Se o objeto já existir nesta coleção e a coleção não permitir duplicatas, será retornado falso. Além disso, algumas coleções são somente leitura. Essas coleções lançam uma UnsupportedOperationException se esse método for chamado.

remoção booleana pública (elemento)

Remove o objeto especificado desta coleção. Assim como o método add(), este método retorna verdadeiro se o objeto for removido da coleção. Se o objeto não existir nesta coleção, será retornado false. Coleções somente leitura lançam uma

UnsupportedOperationException se esse método for chamado.

booleano público contém (elemento)

Retornará verdadeiro se a coleção contiver o objeto especificado.

tamanho interno público()

Retorna o número de elementos nesta coleção.

booleano público isEmpty()

Retorna verdadeiro se esta coleção não tiver elementos.

iterador público iterador()

Examina todos os elementos desta coleção. Este método retorna um Iterator, que é um objeto que você pode usar para percorrer os elementos da coleção. Falaremos mais sobre iteradores na próxima seção.

Além disso, os métodos addAll(), removeAll() e containsAll() aceitam outra coleção e adicionam, removem ou testam todos os elementos da coleção fornecida.

Tipos de coleção

A interface Collection possui três interfaces filhas. Set representa uma coleção na qual elementos duplicados não são permitidos. Lista é uma coleção cujos elementos possuem uma ordem específica. A interface Queue é um buffer para objetos com a noção de um elemento “head” que é o próximo na fila para processamento.

Definir

Definirnão possui métodos além daqueles que herda de Collection. Ele simplesmente impõe sua regra de não duplicação. Se você tentar adicionar um elemento que já existe em um Conjunto, o método add() simplesmente retornará falso. SortedSet mantém os elementos em uma ordem prescrita; como uma lista ordenada que não pode conter duplicatas. Você pode recuperar subconjuntos (que também são classificados) usando os métodos subSet(), headSet() e tailSet(). Esses métodos aceitam um ou dois elementos que marcam os limites. As chamadas first() e last() fornecem acesso ao primeiro e ao último elementos, respectivamente. E o método

comparator() retorna o objeto usado para comparar os elementos (mais sobre isso em“Uma análise mais

detalhada: o método sort()”).

Conjunto Navegávelestende SortedSet e adiciona métodos para encontrar a correspondência mais próxima maior ou menor que um valor alvo na ordem de classificação do Conjunto. Você pode implementar essa interface de forma eficiente usando técnicas como listas de pulos, que agilizam a localização de elementos ordenados.

Lista

A próxima interface filha de Collection é List. Uma Lista é uma coleção ordenada, semelhante a um array, mas com métodos para manipular a posição dos elementos na lista:

adição booleana pública (Eelemento)

Adiciona o elemento especificado ao final da lista.

adição pública nula (intíndice, elemento E)

Insere o objeto fornecido na posição fornecida na lista. Se a posição for menor que zero ou maior que o comprimento da lista, uma IndexOutOfBoundsException será lançada. O elemento que estava anteriormente na posição fornecida, e todos os elementos depois dele, são movidos uma posição de índice para cima.

remoção de void público (intíndice)

Remove o elemento na posição especificada. Todos os elementos subsequentes descem uma posição de índice.

público E get(intíndice)

Retorna o elemento na posição especificada, mas não altera a lista.

conjunto de objetos públicos (intíndice, elemento E)

Altera o elemento na posição especificada para o objeto especificado. Já deve haver um objeto no índice ou então uma IndexOutOfBoundsException será lançada. Nenhum outro elemento da lista é afetado.

O tipo E nesses métodos refere-se ao tipo de elemento parametrizado da classe List.

Coleção, Conjunto e Lista são todos tipos de interface. Este é um exemplo do recurso Genérico que sugerimos na introdução deste capítulo, e veremos implementações concretas deles em breve.

Fila

Uma fila é uma coleção que atua como um buffer para elementos. A fila mantém a ordem de inserção dos itens nela colocados e tem a noção de item “cabeça”. As filas podem ser primeiro a entrar, primeiro a sair (FIFO ou “em ordem”) ou último a entrar, primeiro a sair (LIFO, às vezes ordem “mais recente” ou “inversa”), dependendo da implementação:

oferta booleana pública (elemento E), adição booleana pública (elemento E) O método offer() tenta colocar o elemento na fila, retornando verdadeiro se for bem-sucedido. Diferentes tipos de fila podem ter diferentes limites ou restrições nos tipos

de elementos (incluindo capacidade). Este método difere do método add() herdado de Collection porque retorna um valor booleano em vez de lançar uma exceção para indicar que a coleção não pode aceitar o elemento.

público Eenquete(), público E remover()

O método poll() remove o elemento no início da fila e o retorna. Este método difere do método Collection remove() porque, se a fila estiver vazia, null será retornado em vez de lançar uma exceção.

público Eolhadinha()

Retorna o elemento head sem removê-lo da fila. Se a fila estiver vazia, será retornado nulo.

A interface do mapa

A estrutura de coleções também inclui java.util.Map, que é uma coleção de pares chave/valor. Outros nomes para um mapa são “dicionário” ou “matriz associativa”. Os mapas armazenam e recuperam elementos com valores-chave; eles são muito úteis para coisas como caches e bancos de dados minimalistas. Ao armazenar um valor em um mapa, você associa um objeto-chave a esse valor. Quando você precisa procurar o valor, o mapa o recupera usando a chave.

Com genéricos (aquele tipo E novamente), um tipo Map é parametrizado com dois tipos: um para as chaves e outro para os valores. O trecho a seguir usa um HashMap, que é um tipo de implementação de mapa eficiente, mas não ordenado, que discutiremos mais tarde:

Map<String, Data> dateMap = new HashMap<String, Data>();

dateMap.put("hoje", new Date());

Data hoje = dateMap.get("hoje");

No código legado, os mapas simplesmente mapeiam tipos de objetos para tipos de objetos e exigem uma conversão apropriada para recuperar valores.

As operações básicas no Map são diretas. Nos métodos a seguir, o tipo K refere-se ao tipo de parâmetro chave e o tipo V refere-se ao tipo de parâmetro valor: **público V colocar (Kchave, valor V)**

Adiciona o par chave/valor especificado ao mapa. Se o mapa já contiver um valor para a chave especificada, o valor antigo será substituído e retornado como resultado.

público V obtém (Kchave)

Recupera o valor correspondente à chave do mapa.

público V remover (Kchave)

Remove o valor correspondente à chave do mapa. O valor removido é retornado.

tamanho interno público()

Retorna o número de pares chave/valor neste mapa.

Você pode recuperar todas as chaves ou valores no mapa usando os seguintes métodos:

Conjunto público keySet()

Este método retorna um Set que contém todas as chaves neste mapa.

valores da coleção pública()

Use este método para recuperar todos os valores neste mapa. A coleção retornada pode conter elementos duplicados.

Conjunto público entradaSet()

Este método retorna um conjunto que contém todos os pares chave/valor (como objetos Map.Entry) neste mapa.

Mapatem uma interface filha, SortedMap. Um SortedMap mantém seus pares chave/valor classificados em uma ordem específica de acordo com as chaves. Ele fornece os métodos subMap(), headMap() e tailMap() para recuperar subconjuntos de mapas classificados. Assim como SortedSet, ele também fornece um método comparator(), que retorna um objeto que determina como as chaves do mapa são classificadas. Falaremos mais sobre isso em“Uma análise mais detalhada: o método

sort()”. Java 7 adicionou um NavigableMap com funcionalidade paralela à do NavigableSet; ou seja, adiciona métodos para pesquisar nos elementos classificados um elemento maior ou menor que um valor alvo.

Por fim, devemos deixar claro que embora estejam relacionados, Map não é literalmente um tipo de Collection (Map não estende a interface de Collection). Você pode se perguntar por quê. Todos os métodos da interface Collection parecem fazer sentido para Map, exceto iterator(). Um Mapa, novamente, possui dois conjuntos de objetos: chaves e valores, e iteradores separados para cada um. É por isso que um Mapa não implementa uma Coleção. Se você deseja uma

visualização semelhante a uma coleção de um mapa com chaves e valores, você pode usar o método entrySet().

Mais uma observação sobre mapas: algumas implementações de mapas (incluindo o HashMap padrão do Java) permitem que null seja usado como chave ou valor, mas outras não.

Limitações de tipo

Genéricos são sobre abstração. Os genéricos permitem criar classes e métodos que funcionam da mesma maneira em diferentes tipos de objetos. O termo genérico vem da ideia de que gostaríamos de poder escrever algoritmos gerais que pudessem ser amplamente reutilizados para muitos tipos de objetos, em vez de ter que adaptar nosso código para se adequar a cada circunstância. Este conceito não é novo; é o ímpeto por trás da própria programação orientada a objetos. Os genéricos Java não adicionam novos recursos à linguagem, mas tornam o código Java reutilizável mais fácil de escrever e ler.

Os genéricos levam a reutilização para o próximo nível, tornando o tipo dos objetos com os quais trabalhamos um parâmetro explícito do código genérico. Por esse motivo, os genéricos também são chamados de tipos parametrizados. No caso de uma classe genérica, o desenvolvedor especifica um tipo como parâmetro (um argumento) sempre que usar o tipo genérico. A classe é parametrizada pelo tipo fornecido ao qual o código se adapta.

Em outras linguagens, os genéricos são às vezes chamados de modelos, que é mais um termo de implementação. Os templates são como classes

intermediárias, aguardando seus parâmetros de tipo para que possam ser utilizados. Java segue um caminho diferente, que traz vantagens e desvantagens que descreveremos em detalhes neste capítulo.

Há muito a dizer sobre os genéricos Java. Alguns dos pontos delicados podem parecer um pouco obscuros no início, mas não desanime. A grande maioria do que você fará com genéricos — usando classes existentes como List e Set, por exemplo — é fácil e intuitivo. Projetar e criar seus próprios genéricos requer uma compreensão mais cuidadosa e exigirá um pouco de paciência e ajustes.

Começamos nossa discussão nesse espaço intuitivo com o argumento mais convincente para genéricos: as classes e coleções de contêineres que acabamos de abordar. A seguir, daremos um passo para trás e veremos o que há de bom, de ruim e de feio em como os genéricos Java funcionam. Concluímos examinando algumas classes genéricas do mundo real em Java.

Contêineres: Construindo uma Ratoeira Melhor

Lembre-se de que em uma linguagem de programação orientada a objetos como Java, polimorfismo significa que os objetos são sempre intercambiáveis até certo ponto.

Qualquer filho de um tipo de objeto pode servir no lugar de seu tipo pai e, em última análise, todo objeto é filho de java.lang.Object: a “Eve” orientada a objetos, por assim dizer.

É natural que os tipos mais gerais de contêineres em Java funcionem com o tipo Object para que possam conter praticamente qualquer coisa. Por contêineres, queremos dizer classes que contêm instâncias de outras

classes de alguma forma. A estrutura de coleções Java que vimos na seção anterior é o melhor exemplo de contêineres. List, para recapitular, contém uma coleção ordenada de elementos do tipo Object. E Map mantém uma associação de pares chave/valor, sendo as chaves e valores também do tipo mais geral, Object. Com uma pequena ajuda de invólucros para tipos primitivos, esse arranjo nos serviu bem. Mas (para não ficar muito zen para você), uma “coleção de qualquer tipo” também é uma “coleção de nenhum tipo”, e trabalhar com Objetos impõe uma grande responsabilidade ao desenvolvedor.

É como uma festa à fantasia de objetos onde todos usam a mesma máscara e desaparecem na multidão da coleção. Depois que os objetos são vestidos como o tipo Object, o compilador não consegue mais ver os tipos reais e os perde de vista. Cabe ao usuário furar o anonimato dos objetos posteriormente usando uma conversão de tipo.

E, como tentar arrancar a barba falsa de um festeiro, é melhor você acertar o elenco ou terá uma surpresa indesejável:

```
Data data = new Data();

Lista lista = new ArrayList();

lista.add(data);

// outro código que pode adicionar ou remover elementos...

Data primeiroElemento = (Data)list.get(0); // O elenco está correto? Talv ez.
```

A interface List possui um método add() que aceita qualquer tipo de objeto. Aqui, atribuímos uma instância de ArrayList, que é simplesmente uma implementação da interface List, e adicionamos um objeto Date. A conversão neste exemplo está correta?

Depende do que acontece no período de tempo “outro código” elidido.

O compilador Java sabe que esse tipo de atividade é complicado e atualmente emite avisos quando você adiciona elementos a um ArrayList simples, como acima. Podemos ver isso com um pequeno desvio do jshell. Após importar dos pacotes java.util e javax.swing, tente criar um ArrayList e adicione alguns elementos diferentes:

```
jshell> importar java.util.ArrayList;

jshell> importar javax.swing.JLabel;

jshell> ArrayList coisas = new ArrayList();
```

coisas ==> []

jshell> coisas.add("Olá");

| Aviso:

| chamada não verificada para add(E) como um membro do tipo bruto java.ut il.ArrayList

| coisas.add("Olá");

| ^--------------------^

$3 ==> verdadeiro

jshell> coisas.add(new JLabel("Olá"));

| Aviso:

| chamada não verificada para add(E) como um membro do tipo bruto java.ut il.ArrayList

| coisas.add(new JLabel("Olá"));

| ^------------------------------^

$ 5 ==> verdadeiro

jshell> coisas

coisas ==> [Olá, javax.swing.JLabel[...,text=Olá,...]]

Você pode ver que o aviso é o mesmo, não importa o tipo de objeto adicionado (). Na última etapa, onde exibimos o conteúdo das coisas, tanto o objeto String simples quanto o objeto JLabel estão felizes na lista. O compilador não está preocupado em

usar tipos diferentes; é útil avisar que não saberá se conversões como a (Data) acima funcionarão em tempo de execução.

Os contêineres podem ser consertados?

É natural perguntar se existe uma maneira de melhorar esta situação. E se soubermos que só colocaremos Datas em nossa lista? Não podemos simplesmente criar nossa própria lista que aceita apenas objetos Date, nos livrar da conversão e deixar o compilador nos ajudar novamente? A resposta, talvez surpreendentemente, é não.

Pelo menos, não de uma forma muito satisfatória.

Nosso primeiro instinto pode ser tentar “substituir” os métodos de ArrayList em uma subclasse. Mas é claro que reescrever o método add() em uma subclasse não substituiria nada; adicionaria um novo método sobrecarregado:

```
public void add(Object o) { ... } // ainda aqui

public void add(Date d) { ... } // método sobrecarregado
```

O objeto resultante ainda aceita qualquer tipo de objeto — ele apenas invoca métodos diferentes para chegar lá.

Seguindo em frente, podemos assumir uma tarefa maior. Por exemplo, podemos escrever nossa própria classe DateList que não estende ArrayList, mas delega a essência de seus métodos à implementação de ArrayList. Com uma quantidade razoável de trabalho tedioso, isso nos daria um objeto que faz tudo o que uma Lista faz, mas funciona com Datas de uma forma que tanto o compilador quanto o ambiente de tempo de execução possam entender e aplicar. No entanto, agora demos um

tiro no pé porque nosso contêiner não é mais uma implementação de List. Isso significa que não podemos usá-lo de forma interoperável com todos os utilitários que lidam com coleções, como Collections.sort(), ou adicioná-lo a outra coleção com o método Collection addAll().

Generalizando, o problema é que em vez de refinar o comportamento dos nossos objetos, o que realmente queremos fazer é alterar o contrato deles com o usuário.

Queremos adaptar as assinaturas de seus métodos para um tipo mais específico, e o polimorfismo não permite isso. Então estamos presos aos objetos para nossas coleções? É aí que entram os genéricos.

Insira genéricos

Como observamos ao apresentar as limitações de tipo na seção anterior, os genéricos aprimoram a sintaxe das classes que nos permitem especializar a classe para um determinado tipo ou conjunto de tipos. Uma classe genérica requer um ou mais parâmetros de tipo sempre que nos referirmos ao tipo de classe. Ele os usa para se personalizar.

Se você olhar o código-fonte ou Javadoc da classe List, por exemplo, verá que ela define algo assim:

lista de classe pública< E > {

// ...

público void adicionar (elemento E) { ... }

público E get(int i) { ... }

}

O identificador E entre colchetes angulares (<>) é um parâmetro de tipo.1Indica que a classe List é genérica e requer um tipo Java como argumento para torná-la completa. O

nome E é arbitrário, mas existem convenções que veremos à medida que

prosseguirmos. Neste caso, o parâmetro de tipo E representa o tipo de elementos que queremos armazenar na lista. A classe List refere-se ao parâmetro de tipo dentro de seu corpo e métodos como se fosse um tipo real, a ser substituído posteriormente. O

parâmetro type pode ser usado para declarar variáveis de instância, argumentos para métodos e o tipo de retorno de métodos. Nesse caso, E é usado como o tipo dos elementos que adicionaremos por meio do método add() e para o tipo de retorno do método get(). Vamos ver como usá-lo.

A mesma sintaxe de colchetes angulares fornece o parâmetro type quando queremos usar o tipo List:

List<String> listOfStrings;

Neste trecho, declaramos uma variável chamada listOfStrings usando o tipo genérico List com um parâmetro de tipo String. String refere-se à classe String, mas poderíamos ter uma Lista especializada com qualquer tipo de classe Java. Por exemplo: Lista<Data> datas;

Lista<java.math.BigDecimal> decimais;

List<HelloJava> saudações;

Completar o tipo fornecendo seu parâmetro de tipo é chamado de instanciação do tipo. Às vezes também é chamado de invocação do tipo, por analogia com a invocação de um método e o fornecimento de seus argumentos. Enquanto com um tipo Java regular simplesmente nos referimos ao tipo pelo nome, um tipo genérico como List<> deve ser instanciado com parâmetros onde quer que seja usado.2Especificamente, isso significa que devemos instanciar o tipo em todos os lugares onde os tipos podem aparecer: como o tipo declarado de uma variável (como mostrado neste trecho de código), como o tipo de um parâmetro de método, como o tipo de retorno de um método, ou em um expressão de alocação de objetos usando a palavra-chave new.

Voltando ao nosso listOfStrings, o que temos agora é efetivamente uma List na qual o tipo String foi substituído pela variável de tipo E no corpo da classe: lista de classe pública<String> {

// ...

public void add (elemento String) { ... }

String pública get(int i) { ... }

}

Especializamos a classe List para trabalhar com elementos do tipo String e apenas elementos do tipo String. Esta assinatura de método não é mais capaz de aceitar um tipo Object arbitrário.

Listaé apenas uma interface. Para usar a variável, precisaremos criar uma instância de alguma implementação real de List. Como fizemos em nossa

introdução, usaremos ArrayList. Como antes, ArrayList é uma classe que implementa a interface List, mas neste caso, tanto List quanto ArrayList são classes genéricas. Como tal, eles exigem parâmetros de tipo para instanciá-los onde são usados. Claro, criaremos nosso ArrayList para conter elementos String que correspondam à nossa Lista de Strings: List<String> listOfStrings = new ArrayList<String>();

// Ou abreviação em Java 7.0 e posterior

List<String> listOfStrings = new ArrayList<>();

Como sempre, a palavra-chave new recebe um tipo Java e coloca parênteses com possíveis argumentos para o construtor da classe. Nesse caso, o tipo é ArrayList<String> — o tipo genérico ArrayList instanciado com o tipo String.

Declarar variáveis (como mostrado na primeira linha do exemplo anterior) é um pouco complicado porque exige que forneçamos o tipo de parâmetro genérico duas vezes: uma vez no lado esquerdo no tipo de variável e uma vez no lado direito na expressão inicial. E em casos complicados, os tipos genéricos podem ficar muito longos e aninhados uns nos outros.

O compilador é inteligente o suficiente para inferir o tipo da expressão de inicialização a partir do tipo da variável à qual você a está atribuindo. Isso é chamado de inferência de tipo genérico e se resume ao fato de que você pode usar abreviações no lado direito de suas declarações de variáveis, deixando de fora o conteúdo da notação <>, conforme mostrado na segunda versão do exemplo.

Agora podemos usar nossa lista especializada com strings. O compilador nos impede de tentar colocar qualquer coisa diferente de um objeto String (ou um subtipo de String, se houver) na lista. Também nos permite buscar objetos String com o método get() sem exigir qualquer conversão:

jshell> ArrayList<String> listOfStrings = new ArrayList<>(); listaDeStrings ==> []

jshell> listOfStrings.add("Ei!");

$ 8 ==> verdadeiro

jshell> listOfStrings.add(new JLabel("Olá"));

| Erro:

| tipos incompatíveis: javax.swing.JLabel não pode ser convertido em java

.lang.String

| listOfStrings.add(new JLabel("Olá"));

| ^--------------------^

jshell> String s = strings.get(0);

s ==> "Ei!"

Vejamos outro exemplo da API de coleções. A interface Map fornece um mapeamento semelhante a um dicionário que associa objetos-chave a objetos de valor. Chaves e valores não precisam ser do mesmo tipo. A interface genérica do Mapa requer dois parâmetros de tipo: um para o tipo de chave e outro para o tipo de valor. O Javadoc fica assim:

classe pública Mapa< K, V > {

// ...

public V put(K key, V value) { ... } // retorna qualquer valor antigo público V get (chave K) { ... }

}

Podemos fazer um mapa que armazene objetos Employee por números inteiros de “ID

de funcionário”, assim:

Mapa<Inteiro, Funcionário> funcionários = new HashMap<Inteiro, Funcionári o>();

BobsId inteiro = 314; // viva para o autoboxing!

Funcionário bob = new Funcionário("Bob", ...);

funcionários.put(bobsId, bob);

Funcionário funcionário = funcionários.get(bobsId);

Aqui usamos HashMap, que é uma classe genérica que implementa a interface Map.

Instanciamos ambos os tipos com os parâmetros de tipo Integer e Employee. O Mapa agora funciona apenas com chaves do tipo Inteiro e contém valores do tipo Funcionário.

A razão pela qual usamos Integer aqui para armazenar nosso número é que os parâmetros de tipo para uma classe genérica devem ser tipos de classe. Não podemos parametrizar uma classe genérica com um tipo primitivo, como int ou boolean.

Felizmente, o autoboxing de primitivas em Java (veja“Invólucros para tipos

primitivos”) quase faz parecer que podemos, permitindo-nos usar tipos primitivos como se fossem tipos de wrapper.

Dezenas de outras APIs além de Coleções usam genéricos para permitir adaptá-los a tipos específicos. Falaremos sobre eles à medida que ocorrerem ao longo do livro.

Falando sobre tipos

Antes de passarmos para coisas mais importantes, devemos dizer algumas palavras sobre a forma como descrevemos uma parametrização específica de uma classe genérica. Como o caso mais comum e convincente para genéricos é para objetos semelhantes a contêineres, é comum pensar em termos de um tipo genérico

“mantendo” um tipo de parâmetro. Em nosso exemplo, chamamos nossa List<String> de “lista de strings” porque, com certeza, era isso mesmo. Da mesma forma, poderíamos ter chamado nosso mapa de funcionários de “Mapa de IDs de funcionários

para objetos de funcionários”. Entretanto, essas descrições focam um pouco mais no que as classes fazem do que no tipo em si.

Em vez disso, pegue um contêiner de objeto único chamado Trap< E > que pode ser instanciado em um objeto do tipo Mouse ou do tipo Bear; isto é, Trap<Mouse> ou Trap<Bear>. Nosso instinto é chamar o novo tipo de “armadilha para ratos” ou

“armadilha para ursos”. Também podemos pensar em nossa lista de strings como um novo tipo. Poderíamos falar sobre uma “lista de strings” ou descrever nosso mapa de funcionários como um novo tipo de “mapa de objeto de funcionário inteiro”. Você pode usar qualquer palavreado que preferir, mas essas últimas descrições focam mais na noção do genérico como um tipo e podem ajudá-lo a manter os termos corretos quando discutirmos como os tipos genéricos estão relacionados no sistema de tipos.

Lá veremos que a terminologia do contêiner acaba sendo um pouco contra-intuitiva.

Na seção a seguir, discutiremos os tipos genéricos em Java de uma perspectiva diferente. Vimos um pouco do que eles podem fazer; agora precisamos conversar sobre como eles fazem isso.

"Não tem colher"

No filme Matrix,3ao herói Neo é oferecida uma escolha: tomar a pílula azul e permanecer no mundo da fantasia, ou tomar a pílula vermelha e ver as coisas como elas realmente são. Ao lidar com genéricos em Java, nos deparamos com um dilema ontológico semelhante. Só podemos ir até certo ponto em qualquer discussão sobre genéricos antes de sermos forçados a confrontar a realidade de como eles são implementados. Nosso mundo de fantasia é criado pelo compilador para tornar nossa vida escrevendo código mais fácil de aceitar. Nossa realidade (embora não seja exatamente o pesadelo distópico do filme) é um lugar mais difícil, cheio de perigos e questões invisíveis. Por que as conversões e testes não funcionam corretamente com genéricos? Por que não consigo implementar o que parecem ser duas

interfaces genéricas diferentes em uma classe? Por que posso declarar um array de tipos genéricos, mesmo que não haja nenhuma maneira em Java de criar tal array?!?

Responderemos a essas perguntas e muito mais no restante deste capítulo, e você nem precisará esperar pela sequência. Você estará dobrando colheres (bem, tipos) em pouco tempo. Vamos começar.

Apagamento

Os objetivos de design para genéricos Java eram formidáveis: adicionar uma nova sintaxe radical à linguagem que introduza com segurança tipos parametrizados sem impacto no desempenho e, ah, a propósito, torná-la compatível com versões anteriores de todo o código Java existente e não alterar o classes compiladas de qualquer maneira séria. É incrível que eles realmente satisfizessem essas condições e não é surpresa que tenha demorado um pouco. Mas, como sempre, alguns

compromissos necessários geraram algumas dores de cabeça.

Para realizar essa façanha, Java emprega uma técnica chamada apagamento. O

apagamento está relacionado à ideia de que, como quase tudo que fazemos com genéricos se aplica estaticamente em tempo de compilação, as informações genéricas não precisam ser transportadas para as classes compiladas. A natureza genérica das classes, imposta pelo compilador, pode ser “apagada” nas classes binárias, mantendo a compatibilidade com código não genérico.

Embora Java retenha informações sobre os recursos genéricos das classes na forma compilada, essas informações são usadas principalmente pelo compilador. O tempo de execução Java não sabe nada sobre genéricos (e não desperdiça recursos com eles).

Podemos usar jshell para confirmar a noção de tempo de execução de um List<E> parametrizado ainda sendo um List:

jshell> importar java.util.*;

jshell> List<Data> dateList = new ArrayList<Data>();

lista de datas ==> []

jshell> dateList instância da lista

$3 ==> verdadeiro

Mas nosso dateList genérico claramente não implementa os métodos List que acabamos de discutir:

jshell> dateList.add(novo Objeto())

| Erro:

| tipos incompatíveis: java.lang.Object não pode ser convertido em java.u til.Date

| dateList.add(novo objeto())

| ^----------^

Isso ilustra a natureza um tanto eclética dos genéricos Java. O compilador acredita neles, mas o tempo de execução diz que são uma ilusão. E se tentarmos algo

um pouco mais simples e verificarmos se nosso dateList é um List<Date>:

jshell> dateList instanceof List<Data>;

| Erro:

| tipo genérico ilegal, por exemplo,

| dateList instanceof List<Data>;

| ^--------^

Desta vez, o compilador simplesmente bate o pé e diz: “Não”. Você não pode testar um tipo genérico em uma operação instanceof. Como não existem classes discerníveis para diferentes parametrizações de List em tempo de execução (cada List ainda é uma List), não há como o operador instanceof saber a diferença entre uma encarnação de List e outra. Toda a verificação genérica de segurança foi feita em tempo de compilação, portanto, em tempo de execução, estamos lidando apenas com um único tipo de List real.

Aqui está o que realmente aconteceu: o compilador apagou toda a sintaxe dos colchetes angulares e substituiu os parâmetros de tipo em nossa classe List por um tipo que pode funcionar em tempo de execução com qualquer tipo permitido: neste caso, Object. Pareceríamos estar de volta ao ponto de partida, exceto que o compilador ainda tem o conhecimento para impor nosso uso dos genéricos no código em tempo de compilação e pode, portanto, lidar com a conversão para nós. Se você descompilar uma classe usando List<Date> (o comando javap com a opção -c mostra o bytecode, se você ousar), você verá que o código compilado na

verdade contém a conversão para Date, mesmo que não tenhamos feito isso. escreva nós mesmos.

Podemos agora responder a uma das questões colocadas no início da seção: “Por que não consigo implementar o que parecem ser duas interfaces genéricas diferentes em uma classe?” Não podemos ter uma classe que implemente duas instanciações genéricas diferentes de List porque elas são realmente do mesmo tipo em tempo de execução e não há como diferenciá-las:

classe abstrata pública DualList implementa List<String>, List<Date> { }

// Erro: java.util.List não pode ser herdado com argumentos diferentes:

// <java.lang.String> e <java.util.Date>

Felizmente, sempre há soluções alternativas. Neste caso, por exemplo, você pode usar uma superclasse comum ou criar múltiplas classes. As alternativas podem não ser tão elegantes, mas quase sempre você pode chegar a uma resposta clara – mesmo que seja um pouco detalhada.

Tipos brutos

Embora o compilador trate diferentes parametrizações de um tipo genérico como tipos diferentes (com APIs diferentes) em tempo de compilação, vimos que existe apenas um tipo real em tempo de execução. Por exemplo, tanto List<Date> quanto List<String> compartilham a antiga classe Java List. List é chamado de tipo bruto da classe genérica. Todo genérico possui um tipo bruto. É o formato Java básico “simples”

do qual todas as informações de tipo genérico foram removidas e as variáveis de tipo substituídas por um tipo Java geral como Object.

É possível usar tipos brutos em Java. Porém, o compilador Java gera um aviso sempre que são usados de forma “insegura”. Fora do jshell, o compilador ainda percebe estes problemas:

// código Java não genérico usando o tipo bruto

Lista lista = new ArrayList(); //tarefa ok

lista.add("foo"); // Aviso do compilador sobre uso do tipo bruto Este trecho usa o tipo List bruto da mesma forma que o código Java antigo anterior ao Java 5 faria. A diferença é que agora o compilador Java emite um aviso não verificado sobre o código se tentarmos inserir um objeto na lista:

% javac RawType.java

Nota: RawType.java usa operações não verificadas ou inseguras.

Nota: Recompile com -Xlint:unchecked para obter detalhes.

O compilador nos instrui a usar a opção -Xlint:unchecked para obter informações mais específicas sobre os locais de operações inseguras:

% javac -Xlint: desmarcado MyClass.java

RawType.java:6: aviso: [desmarcado] chamada desmarcada para add(E)

como membro da lista de tipo bruto

lista.add("foo");

^

onde E é uma variável de tipo:

E estende o objeto declarado na lista de interfaces

Observe que criar e atribuir o ArrayList bruto não gera um aviso. Somente quando tentamos usar um método “inseguro” (aquele que se refere a uma variável de tipo) é que recebemos o aviso. Isso significa que ainda é aceitável usar APIs Java não genéricas e de estilo antigo que funcionam com tipos brutos. Recebemos avisos apenas quando fazemos algo inseguro em nosso próprio código.

Mais uma coisa sobre o apagamento antes de prosseguirmos. Nos exemplos anteriores, as variáveis de tipo foram substituídas pelo tipo Object, que poderia representar qualquer tipo aplicável à variável de tipo E. Mais tarde, veremos que nem sempre é esse o caso. Podemos colocar limitações ou limites nos tipos de parâmetros e, quando o fazemos, o compilador pode ser mais restritivo quanto ao apagamento do tipo, por exemplo:

classe Limitada<E estende Data> {

public void addElement (elemento E) { ... }

}

Esta declaração de tipo de parâmetro diz que o tipo de elemento E deve ser um subtipo do tipo Date. Neste

caso, o apagamento do método addElement() é, portanto, mais restritivo que Object, e o compilador usa Date:

public void addElement (elemento de data) { ... }

Dataé chamado de limite superior deste tipo, o que significa que é o topo da hierarquia de objetos aqui. Você só pode instanciar o tipo parametrizado em um tipo Date ou em um tipo “inferior” (mais derivado ou subclasse).

Agora que sabemos o que realmente são os tipos genéricos, podemos entrar em mais detalhes sobre como eles se comportam.

Relacionamentos de tipo parametrizado

Sabemos agora que os tipos parametrizados compartilham um tipo bruto comum. É

por isso que nosso List<Date> parametrizado é apenas um List em tempo de execução.

Na verdade, podemos atribuir qualquer instanciação de List ao tipo bruto se quisermos:

Lista lista = new ArrayList<Data>();

Podemos até ir por outro caminho e atribuir um tipo bruto a uma instanciação específica do tipo genérico:

List<Data> datas = new ArrayList(); // aviso desmarcado

Esta instrução gera um aviso não verificado na atribuição, mas depois disso, o compilador confia que a lista continha apenas datas anteriores à atribuição. Você

pode tentar converter new ArrayList() para List<Date>, mas isso não resolverá o aviso.

Falaremos sobre conversão para tipos genéricos em“Elencos”.

Quaisquer que sejam os tipos de tempo de execução, o compilador está executando o programa. Não nos permite atribuir coisas claramente incompatíveis:

List<Data> datas = new ArrayList<String>(); // Erro em tempo de compilaçã o!

Obviamente, ArrayList<String> não implementa os métodos de List<Date> exigidos pelo compilador, portanto, esses tipos são incompatíveis.

Mas e quanto a relacionamentos de tipo mais interessante? A interface List, por exemplo, é um subtipo da interface Collection mais geral. Você pode pegar uma instanciação específica da Lista genérica e atribuí-la a alguma instanciação da Coleção genérica? Depende dos parâmetros de tipo e de seus relacionamentos? Claramente, um List<Date> não é um Collection<String>. Mas uma List<Date> é uma Collection<Date>? Uma List<Date> pode ser uma Collection<Object>?

Vamos apenas deixar escapar a resposta aqui primeiro, depois analisá-la e explicar. A regra para os tipos simples de instanciações genéricas que discutimos até agora é que a herança se aplica apenas ao tipo genérico “base” e não aos tipos de parâmetros.

Além disso, a atribuibilidade se aplica somente quando os dois tipos genéricos são instanciados exatamente no mesmo tipo de parâmetro. Em outras palavras, ainda

existe herança unidimensional, seguindo o tipo de classe genérica base, mas com a restrição adicional de que os tipos de parâmetros devem ser idênticos.

Por exemplo, como List é um tipo de Collection, podemos atribuir instanciações de List a instanciações de Collection quando o parâmetro de tipo é exatamente o mesmo: Coleção<Data> cd;

List<Data> ld = new ArrayList<Data>();

cd=ld; // OK!

Este trecho de código diz que List<Date> é Collection<Date> — bastante intuitivo. Mas tentar a mesma lógica em uma variação nos tipos de parâmetros falha:

List<Objeto> lo;

List<Data> ld = new ArrayList<Data>();

el = ld; // Erro em tempo de compilação! Tipos incompatíveis.

Embora a nossa intuição nos diga que as Datas nessa Lista poderiam viver todas felizes como Objetos numa Lista, a atribuição é um erro. Explicaremos exatamente o porquê na próxima seção, mas por enquanto apenas observe que os parâmetros de tipo não são exatamente os mesmos e que não há relacionamento de herança entre tipos de parâmetros em genéricos.

Este é um caso em que ajuda pensar na instanciação em termos de tipos e não em termos do que os objetos instanciados fazem. Estas não são realmente uma “lista de datas” e uma “lista de objetos” – são mais como uma

DateList e uma ObjectList, cuja relação não é imediatamente óbvia.

Tente escolher o que está bem e o que não está bem no exemplo a seguir: Coleção<Número> cn;

List<Integer> li = new ArrayList<Integer>();

cn = li;

É possível que uma instanciação de List seja uma instanciação de Collection, mas apenas se os tipos de parâmetros forem exatamente os mesmos. A herança não segue os tipos de parâmetros, portanto a atribuição final neste exemplo falha.

Mencionamos anteriormente que esta regra se aplica aos tipos simples de instanciações que discutimos até agora neste capítulo. Que outros tipos existem? Bem, os tipos de instanciações que vimos até agora, onde inserimos um tipo Java real como parâmetro, são chamados de instanciações de tipo concreto. Mais tarde, falaremos sobre instanciações curinga, que são como operações matemáticas de conjuntos em tipos (pense em uniões e interseções). É possível fazer instanciações mais exóticas de genéricos onde os relacionamentos de tipo são na verdade bidimensionais, dependendo tanto do tipo base quanto da parametrização. Mas não se preocupe: isso não acontece com muita frequência e não é tão assustador quanto parece.

Por que uma Lista<Data> não é uma Lista<Objeto>?

É uma pergunta razoável. Por que não deveríamos ser capazes de atribuir nosso List<Date> a um List<Object>

e trabalhar com os elementos Date como tipos de objetos?

A razão volta ao cerne da lógica dos genéricos: alterar os contratos de programação.

No caso mais simples, supondo que um tipo DateList estenda um tipo ObjectList, o DateList teria todos os métodos de ObjectList e poderíamos inserir Objects nele.

Agora, você pode objetar que os genéricos nos permitem alterar as assinaturas dos métodos, então isso não se aplica mais. Isso é verdade, mas há um problema maior. Se pudéssemos atribuir nosso DateList a uma variável ObjectList, poderíamos usar métodos Object para inserir elementos de tipos diferentes de Date nele.

Poderíamos alias (fornecer um tipo alternativo e mais amplo) de DateList como ObjectList. Usando o objeto com alias, poderíamos tentar induzi-lo a aceitar algum outro tipo:

ListaDataListaData = new ListaData();

ObjectList objectList = dataList; // Realmente não posso fazer isso

objectList.add(new Foo()); // deve ser um erro de execução!

Esperaríamos obter um erro de tempo de execução quando a implementação real de DateList fosse apresentada com o tipo errado de objeto.

E é aí que reside o problema. Os genéricos Java não têm representação em tempo de execução. Mesmo que essa funcionalidade fosse útil, não há como o Java saber o que

fazer em tempo de execução. Esse recurso é simplesmente perigoso – ele permite erros em tempo de execução que não podem ser detectados em tempo de compilação.

Em geral, gostaríamos de detectar erros de tipo em tempo de compilação.

Você pode pensar que Java poderia garantir a segurança de tipo do seu código se ele compilar sem avisos não verificados, proibindo essas atribuições. Infelizmente não pode, mas essa limitação não tem nada a ver com genéricos; tem a ver com matrizes.

(Se tudo isso lhe parece familiar, é porque mencionamos esse problema emCapítulo

4em relação aos arrays Java.) Os tipos de array têm um relacionamento de herança que permite que esse tipo de alias ocorra:

```
Data [] datas = nova Data[10];

Objeto [] objetos = datas;

objetos[0] = "não é uma data"; //Tempo de execução ArrayStoreException!
```

Matrizes têm representações de tempo de execução como classes diferentes. Eles se verificam em tempo de execução, lançando uma ArrayStoreException em situações como essa. O compilador Java não pode garantir a segurança do tipo do seu código se você usar arrays dessa forma.

Elencos

Já falamos sobre relacionamentos entre tipos genéricos e até mesmo entre tipos genéricos e tipos brutos. Mas ainda não exploramos realmente o conceito de elencos no mundo dos genéricos.

Nenhuma conversão foi necessária quando trocamos genéricos por seus tipos brutos.

Mas acionamos avisos não verificados do compilador:

Lista lista = new ArrayList<Data>();

Lista<Data> dl = lista; // aviso desmarcado

Normalmente, usamos uma conversão em Java para trabalhar com dois tipos que podem ser atribuíveis. Por exemplo, poderíamos tentar converter um Objeto em uma Data porque é plausível que o Objeto possa ser um valor de Data. O elenco então realiza a verificação em tempo de execução para ver se estamos corretos.

A conversão entre tipos não relacionados é um erro em tempo de compilação. Por exemplo, não podemos nem tentar converter um número inteiro em uma string. Esses tipos não têm relacionamento de herança. E quanto às conversões entre tipos genéricos compatíveis?

Coleção<Data> cd = new ArrayList<Data>();

Lista<Data> ld = (Lista<Data>)cd; // OK!

Este trecho de código mostra uma conversão válida de uma Collection<Date> mais geral para uma List<Date>. A conversão é plausível aqui porque um Collection<Date> pode ser atribuído e pode realmente ser um List<Date>.

Da mesma forma, a conversão a seguir detecta nosso erro: colocamos o alias de TreeSet<Date> como Collection<Date> e tentamos convertê-lo em List<Date>: Coleção<Data> cd = new TreeSet<Data>();

Lista<Data> ld = (Lista<Data>)cd; //Tempo de execução ClassCastException!

ld.add(nova data());

No entanto, há um caso em que as conversões não são eficazes com genéricos: quando se tenta diferenciar os tipos com base em seus tipos de parâmetros:

Objeto o = new ArrayList<String>();

Lista<Data> ld = (Lista<Data>)o; // aviso desmarcado, ineficaz Data d = ld.get(0); // inseguro em tempo de execução, a conversão implíci ta pode falhar

Aqui, criamos o alias de ArrayList<String> como um objeto simples. Em seguida, lançamos para List<Date>. Infelizmente, Java não sabe a diferença entre List<String> e List<Date> em tempo de execução, portanto a conversão é infrutífera. O compilador nos avisa gerando um aviso não verificado no local da conversão. Quando tentamos usar o objeto cast, ld, podemos descobrir que ele está incorreto. As conversões em tipos genéricos são ineficazes em tempo de execução devido ao apagamento e à falta de informações de tipo.

Convertendo entre coleções e matrizes

Embora não esteja relacionado por herança direta ou interfaces compartilhadas, a conversão entre coleções e arrays ainda é simples. Por conveniência, você pode

recuperar os elementos de uma coleção como um array usando os seguintes métodos: objeto público[] toArray()

público <E> E[] toArray(E[] a)

O primeiro método retorna um array Object simples. Com a segunda forma, podemos ser mais específicos e recuperar um array do tipo de elemento correto. Se fornecermos um array de tamanho suficiente, ele será preenchido com os valores. Mas se o array for muito curto (por exemplo, comprimento zero), Java criará um novo array do mesmo tipo, mas com o comprimento necessário, e o retornará. Então você pode simplesmente passar um array vazio do tipo correto assim:

Coleção<String> minhaColeção = ...;

String [] minhasStrings = minhaCollection.toArray(new String[0]);

Este truque é um pouco estranho. Seria bom se Java nos permitisse especificar explicitamente o tipo usando uma referência de classe, mas por algum motivo isso não acontece.

Indo na direção oposta, você pode converter um array de objetos em uma coleção List com o método estático asList() da classe auxiliar java.util.Arrays:

String [] minhasStrings = { "a", "b", "c" }; Lista lista = Arrays.asList(myStrings);

O compilador também é inteligente o suficiente para reconhecer uma atribuição válida a uma variável List<String>.

Iterador

Um iterador é um objeto que permite percorrer uma sequência de valores. Esse tipo de operação surge com tanta frequência que possui uma interface padrão: java.util.Iterator. A interface do Iterator possui três métodos interessantes: **público E próximo()**

Este método retorna o próximo elemento (um elemento do tipo genérico E) da coleção associada.

público booleano hasNext()

Este método retorna verdadeiro se você ainda não percorreu todos os elementos da Coleção. Em outras palavras, retorna verdadeiro se você puder chamar next() para obter o próximo elemento.

remoção de vazio público()

Este método remove o objeto mais recente retornado de next() da coleção associada.

O exemplo a seguir mostra como usar um Iterator para imprimir cada elemento de uma coleção:

public void printElements(Coleção c, PrintStream out) {

Iterador iterador = c.iterator();

enquanto (iterator.hasNext()) {

out.println(iterator.next());

}

}

Depois de usar next() para obter o próximo elemento, às vezes você pode removê-lo().

Trabalhar em uma lista de tarefas, por exemplo, pode seguir um padrão: “pegue um item, processe o item, remova o item”. Mas o recurso de remoção de iteradores nem sempre é apropriado e nem todos os iteradores implementam remove(). Não faz sentido remover um elemento de uma coleção somente leitura, por exemplo.

Se a remoção do elemento não for permitida, uma UnsupportedOperationException será lançada desse método. Se você chamar remove() antes de chamar next() pela primeira vez, ou se chamar remove() duas vezes seguidas, você obterá uma IllegalStateException.

Looping sobre coleções

Uma forma do loop for, descrita em“O loop for”, pode operar em todos os tipos Iterable, o que significa que pode iterar em todos os tipos de objetos Collection à medida que essa interface estende Iterable. Por exemplo, agora ele pode percorrer todos os elementos de uma coleção digitada de objetos Date, assim:

```
Coleção<Data> col = ...

for (Data data: col) {

System.out.println(data);

}
```

Esse recurso do loop for integrado do Java é chamado de loop for “aprimorado” (em oposição ao loop for pré-genérico, somente numérico). O loop for aprimorado se aplica apenas a coleções do tipo Coleção, não a Mapas. Mas fazer um loop em um mapa pode ser útil em algumas situações. Você pode usar os métodos keySet() ouvalues() do mapa (ou até mesmo entrySet() se quiser

que cada par chave/valor seja uma entidade única) para obter uma coleção do seu mapa que funcione com este loop for aprimorado:

```
Map<Inteiro, Funcionário> funcionários = new HashMap<>();

// ...

for (ID inteiro: funcionários.keySet()) {

System.out.print("Funcionário " + id);

System.out.println(" => " + funcionários.get(id));

}
```

A coleção de chaves é um conjunto simples e não ordenado. O loop for aprimorado acima mostrará todos os seus funcionários, mas o pedido impresso pode parecer um tanto aleatório. Se você quisesse que eles fossem listados na ordem de seus IDs ou talvez de seus nomes, você precisaria classificar as chaves ou valores primeiro.

Felizmente, a classificação é uma tarefa muito comum – e a estrutura de coleções pode ajudar.

Uma análise mais detalhada: o método sort()

Procurando na classe java.util.Collections, encontramos todos os tipos de métodos utilitários estáticos para trabalhar com coleções. Entre eles está este presente - o método genérico estático sort():

```
<T estende Comparável<? super T >> void sort(Lista<T> lista) { ... }
```

Outra noz para quebrarmos. Vamos nos concentrar na última parte do limite:

Comparável<? ótimoT>

Esta é uma instanciação curinga que menci[onamos em“Relacionamentos de tipo

parametrizado”.](#) Nesse caso, é uma interface, então podemos ler as extensões no tipo de retorno sort() como implementos, se isso ajudar.

Comparávelcontém um método compareTo() para algum tipo de parâmetro. Um Comparable<String> significa que o método compareTo() usa o tipo String. Portanto, Comparável<? super T> é o conjunto de instanciações de Comparable em T e todas as suas superclasses. Um Comparable<T> é suficiente e, na outra extremidade, um Comparable<Object> também.

O que isso significa em inglês é que os elementos devem ser comparáveis ao seu próprio tipo, ou a algum supertipo de seu próprio tipo para que o método sort() possa usá-los. Isso garante que todos os elementos possam ser comparados entre si, mas não é tão restritivo quanto dizer que todos eles devem implementar o método compareTo() por conta própria. Alguns dos elementos podem herdar a interface Comparable de uma classe pai que sabe comparar apenas com um supertipo de T, e é exatamente isso que é permitido.

Aplicação: Árvores no Campo

Há muita teoria neste capítulo. Não tenha medo da teoria – ela pode ajudá-lo a prever o comportamento em novos cenários e inspirar soluções para novos problemas. Mas a

prática é igualmente importante, então vamos revisitar o jogo em que

começamos"Aulas". Em particular, é hora de armazenar mais de um objeto de cada tipo.

EmCapítulo 13abordaremos redes e veremos a criação de uma configuração para dois jogadores que exigiria o armazenamento de vários físicos. Por enquanto, ainda temos um físico que consegue lançar uma maçã de cada vez. Mas podemos povoar nosso campo com diversas árvores para prática de tiro ao alvo.

Vamos adicionar seis árvores. Usaremos um par de loops para que você possa aumentar facilmente a contagem de árvores, se desejar. Nosso campo atualmente armazena uma instância de árvore solitária. Podemos atualizar esse armazenamento para uma lista digitada (chamaremos de árvores). A partir daí podemos abordar a adição e remoção de árvores de várias maneiras:

•

Poderíamos criar alguns métodos para Field que funcionassem com a lista e talvez impor algumas outras regras do jogo (como gerenciar um número

máximo de árvores).

•

Poderíamos simplesmente usar a lista diretamente, pois a classe List já possui bons métodos para a maioria das coisas que queremos fazer.

•

Poderíamos usar alguma combinação dessas abordagens: métodos especiais onde fizer sentido para o nosso jogo e manipulação direta em todos os outros lugares.

Como temos algumas regras de jogo que são peculiares ao nosso Campo, faremos aqui a primeira abordagem. (Mas observe os exemplos e pense em como você pode alterá-los para usar a lista de árvores diretamente.) Começaremos com um método addTree(). Um benefício dessa abordagem é que também podemos realocar a criação da instância da árvore para o nosso método, em vez de criar e manipular a árvore separadamente. Esta é uma maneira de adicionar uma árvore em um ponto desejado do campo:

```
List<Árvore> árvores = new ArrayList<>();

// outro estado do campo

public void addTree(int x, int y) {

Árvore árvore = nova Árvore();

árvore.setPosition(x,y);

árvores.add(árvore);

}
```

Com esse método implementado, poderíamos adicionar algumas árvores

rapidamente:

```
Campo campo = new Campo();

// outro código de configuração
```

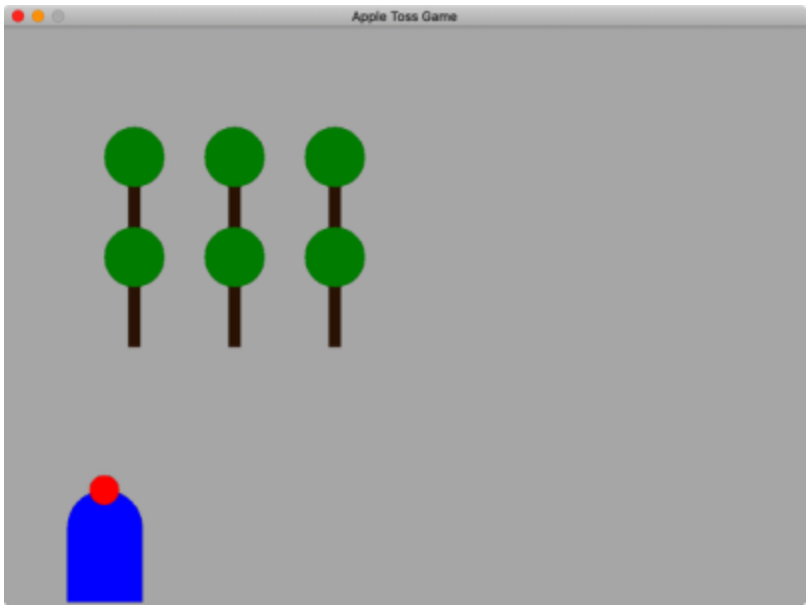
```
campo.addTree(100,100);

campo.addTree(200.100);
```

Essas duas linhas adicionam um par de árvores lado a lado. Vamos em frente e escrever os loops necessários para criar nossas seis árvores:

```
Campo campo = new Campo();

// outro código de configuração

for (int linha = 1; linha <= 2; linha++) {

for (int col = 1; col <=3; col++) {

field.addTree(col * 100, linha * 100);

}

}
```

Você consegue ver agora como seria fácil adicionar oito, nove ou cem árvores? Os computadores são realmente bons em repetição.

Viva por criar nossa floresta de alvos de maçã! No entanto, deixamos de fora alguns detalhes críticos. Mais importante ainda, precisamos mostrar a nossa nova floresta na tela. Também precisamos atualizar nosso método de desenho para a classe Field para que ele entenda e use nossa lista de árvores corretamente. Eventualmente faremos o mesmo com nossos físicos e maçãs à medida que adicionarmos mais funcionalidades ao nosso jogo. Também precisaremos de uma forma de remover elementos que não estão mais ativos. Mas primeiro vamos ver nossa floresta!




```
//Arquivo: Field.java

protegido void paintComponent(Gráficos g) {

g.setColor(fieldColor);

g.fillRect(0,0, getWidth(), getHeight());

for (Árvore t: árvores) {

t.draw(g);

}

físico.draw(g);

maçã.draw(g);

}
```

Como já estamos na classe Field onde nossas árvores estão armazenadas, não há necessidade de escrever uma função separada para extrair uma árvore individual e pintá-la. Podemos usar a elegante estrutura de loop for

aprimorada e colocar rapidamente todas as nossas árvores no campo, como mostrado em Figura 7-1.

Figura 7-1. Renderizando todas as árvores da nossa lista

Recursos úteis

Coleções e genéricos Java são adições muito poderosas e úteis à linguagem. Embora alguns dos detalhes que analisamos na segunda metade deste capítulo possam parecer assustadores, o uso comum é muito simples e convincente: os genéricos tornam as coleções melhores. À medida que você começar a usar mais genéricos, descobrirá que seu código se tornará mais legível e mais fácil de manter. As coleções permitem um armazenamento elegante e eficiente. Os genéricos tornam explícito o que você tinha que inferir anteriormente do uso.

Perguntas de revisão

1. Se você quiser armazenar uma lista de contatos com nomes e números de telefone, que tipo de coleção funcionaria melhor?

2. Qual método você usa para obter um iterador para os itens de um conjunto?

3. Como você pode transformar uma lista em um array?

4. Como você pode transformar um array em uma lista?

5. Qual interface você deve implementar para classificar uma lista usando o método Collections.sort()?

Exercícios de código

1. A classe EmployeeList na pasta ch07exercises contém alguns funcionários carregados em um mapa de IDs de funcionários e objetos Employee,

semelhante ao usado em vários exemplos acima. Mencionamos a impressão desses funcionários de forma ordenada, mas não mostramos nenhum código.

Tente classificar os funcionários por seus números de identificação. Você provavelmente precisará usar o método keySet() e então criar uma lista temporária — mas classificável — desse conjunto.

2. EmExercício Avançado 5.1, você criou uma nova classe de obstáculos, Hedge.

Atualize o jogo para que você possa ter várias sebes semelhantes às múltiplas árvores. Certifique-se de que todas as suas árvores e sebes sejam pintadas corretamente ao executar o programa.

Exercícios Avançados

1. No Exercício de Código 1 acima, você provavelmente classificou as chaves do mapa e depois usou as chaves classificadas corretamente para obter o

funcionário correspondente. Para um desafio um pouco maior, implemente a interface Comparable em sua classe Employee. Você pode decidir como deseja organizar os funcionários: por ID, por sobrenome, por nome completo ou talvez alguma combinação desses atributos. Em vez de classificar a coleção keySet(), tente classificar seus funcionários recentemente comparáveis diretamente, criando uma lista temporária a partir dos valores() do seu mapa.

1Você também pode ver o termo variável de tipo usado. A especificação da linguagem Java usa principalmente parâmetros, então é isso que tentamos manter, mas você pode ver os dois nomes usados livremente.

2Isto é, a menos que você queira usar um tipo genérico de maneira não genérica.

Falaremos sobre tipos “brutos” posteriormente neste capítulo.

3Para aqueles que desejam algum contexto para o título desta seção, é aqui que ele vem. Nosso herói, Neo, está aprendendo sobre seus poderes. Menino: Não tente entortar a colher. Isso é impossível. Em vez disso, apenas tente perceber a verdade.

Neo: Que verdade? Menino: Não há colher.

Neo: Não tem colher? Menino: Então você verá que não é a colher que entorta, é só você mesmo. — Os Wachowskis. O Matrix. 136 minutos. Warner Brothers, 1999.

Capítulo 8. Texto e utilitários principais

Se você leu este livro sequencialmente, você leu tudo sobre as principais construções da linguagem Java, incluindo os aspectos orientados a objetos da linguagem e o uso de

threads. Agora é hora de mudar de assunto e começar a falar sobre a coleção de classes que compõem os pacotes Java padrão e que acompanham cada implementação Java. Os pacotes principais do Java são um de seus recursos mais diferenciados. Muitas outras linguagens orientadas a objetos possuem recursos

semelhantes, mas nenhuma possui um conjunto tão extenso de classes e ferramentas padronizadas quanto o Java.

Isso é tanto um reflexo quanto uma razão para o sucesso do Java.

Cordas

Começaremos examinando mais de perto a classe Java String (ou, mais

especificamente, java.lang.String). Como trabalhar com Strings é tão fundamental, é importante entender como elas são implementadas e o que você pode fazer com elas.

Um objeto String encapsula uma sequência de caracteres Unicode. Internamente, esses caracteres são armazenados em um array Java regular, mas o objeto String protege esse array zelosamente e dá acesso a ele apenas por meio de sua própria API.

Isto é para apoiar a ideia de que Strings são imutáveis; depois de criar um objeto String, você não poderá alterar seu valor. Muitas operações em um objeto String parecem alterar os caracteres ou o comprimento de uma string, mas o que elas realmente fazem é retornar um novo objeto String que copia ou faz referência interna aos caracteres necessários do original. As implementações Java fazem um esforço para consolidar strings idênticas usadas na mesma classe em um conjunto de strings compartilhadas e para compartilhar partes de Strings sempre que possível.

A motivação original para tudo isso foi o desempenho. Strings imutáveis podem economizar memória e o Java

VM pode otimizar seu uso para velocidade. Mas eles não são mágicos. Você deve ter um conhecimento básico da classe String para evitar a criação de um número excessivo de objetos String em locais onde o desempenho é um problema.1

Construindo Strings

Strings literais, definidas em seu código-fonte, são declaradas entre aspas duplas e podem ser atribuídas a uma variável String:

String quote = "Ser ou não ser";

Java converte automaticamente a string literal em um objeto String e a atribui à variável.

Cordas controlam seu próprio comprimento, portanto, objetos String em Java não requerem terminadores especiais. Você pode obter o comprimento de uma String com o método length(). Você também pode testar uma string de comprimento zero usando isEmpty():

comprimento interno = citação.comprimento();

booleano vazio = quote.isEmpty();

Cordas pode aproveitar o único operador sobrecarregado em Java, o operador +, para concatenação de strings. As duas linhas a seguir produzem strings equivalentes:
String nome = "John" + "Smith";

// ou equivalente:

String nome = "John ".concat("Smith");

Para pedaços de texto maiores que um nome, o Java 13 introduziu blocos de texto.

Podemos armazenar um poema com pouco esforço usando três aspas duplas para marcar o início e o fim do bloco de múltiplas linhas. Esse recurso preserva até mesmo o espaço inicial de uma maneira inteligente: o caractere não espacial mais à esquerda torna-se a “borda” esquerda. Os espaços à esquerda dessa borda nas linhas subsequentes são ignorados, mas os espaços após essa borda são mantidos. Considere refazer nosso poema em jshell:

```
jshell> String poema = """

...> Foi brilhante, e os toves escorregadios

...> Girou e girou no wabe:

...> Todos mimsy eram os borogoves,

...> E os mome rats superam.

...> """;

poema ==> "Foi brilhante, e... o mome raths supera.\n"

jshell> System.out.print(poema);

Foi brilhante, e os toves escorregadios

Girou e girou no wabe:

Todos os mimesy eram os borogoves,

E os mome rats superam.

jshell>
```

Incorporar texto extenso no código-fonte normalmente não é algo que você deseja fazer. Para textos com mais de algumas dezenas de linhas,Capítulo 10analisa maneiras de carregar Strings de arquivos.

Além de criar strings a partir de expressões literais, você pode construir uma String diretamente a partir de um array de caracteres:

```
char [] dados = novo char [] { 'L', 'e', 'm', 'm', 'i', 'n', 'g' };

String lemingue = new String(dados);
```

Você também pode construir uma String a partir de um array de bytes:

```
byte [] dados = novo byte [] { (byte)97, (byte)98, (byte)99 };

String abc = new String(dados, "ISO8859_1");
```

Nesse caso, o segundo argumento do construtor String é o nome de um esquema de codificação de caracteres. O construtor String o utiliza para converter os bytes brutos na codificação especificada para a codificação interna escolhida pelo tempo de

execução. Se você não especificar uma codificação de caracteres, o esquema de codificação padrão do seu sistema será usado.2

Por outro lado, o método charAt() da classe String permite acessar os caracteres de uma String de maneira semelhante a um array:

```
Strings = "Newton";
```

for (int i = 0; i < s.length(); i++)

System.out.println(s.charAt(i));

Este código imprime os caracteres da string, um de cada vez.

A noção de que uma String é uma sequência de caracteres também é codificada pela classe String que implementa a interface java.lang.CharSequence, que prescreve os métodos length() e charAt() como forma de obter um subconjunto dos caracteres.

Cordas de coisas

Objetos e tipos primitivos em Java podem ser transformados em uma representação textual padrão como String. Para tipos primitivos como números, a string deve ser bastante óbvia; para tipos de objetos, está sob o controle do próprio objeto. Podemos obter a representação de string de um item com o método estático String.valueOf().

Várias versões sobrecarregadas deste método aceitam cada um dos tipos primitivos: String um = String.valueOf(1); // inteiro, "1"

String dois = String.valueOf(2.384f); //flutuar, "2.384"

String notTrue = String.valueOf(false); // booleano, "falso"

Todos os objetos em Java possuem um método toString() que é herdado da classe Object. Para muitos objetos, este método retorna um resultado útil que exibe o conteúdo do objeto. Por exemplo, o método toString() de um objeto java.util.Date retorna a data que ele representa formatada como uma string. Para objetos que

não fornecem uma representação, o resultado da string é apenas um identificador exclusivo que pode ser usado para depuração. O método String.valueOf(), quando chamado para um objeto, invoca o método toString() do objeto e retorna o resultado.

A única diferença real no uso deste método é que se você passar uma referência de objeto nula, ele retornará a String “null” para você, em vez de produzir uma NullPointerException:

Data data = new Data();

// Equivalente, por exemplo, "Sexta-feira, 19 de dezembro, 05:45:34 CST 1

969"

String d1 = String.valueOf(data);

String d2 = data.toString();

data = nulo;

d1 = String.valueOf(data); // "nulo"

d2 = data.toString(); // Null Pointer Exception!

A concatenação de strings usa o método valueOf() internamente, portanto, se você

“adicionar” um objeto ou primitivo usando o operador mais (+), obterá uma String: String hoje = "A data de hoje é:" + data;

Às vezes, você verá pessoas usando a string vazia e o operador de adição (+) como uma abreviação para obter

o valor da string de um objeto. Por exemplo: String dois = "" + 2.384f;

String hoje = "" + new Date();

É um pouco trapaceiro, mas funciona e é visualmente sucinto.

Comparando Strings

O método equals() padrão pode comparar strings quanto à igualdade; eles devem conter exatamente os mesmos caracteres na mesma ordem. Você pode usar um método diferente, equalsIgnoreCase(), para verificar a equivalência de strings sem diferenciar maiúsculas de minúsculas:

String um = "FOO";

String dois = "foo";

um.igual(dois); // falso

one.equalsIgnoreCase(dois); // verdadeiro

Um erro comum para programadores novatos em Java é comparar strings com o operador == quando eles realmente precisam do método equals(). Lembre-se de que strings são objetos em Java e == testa a identidade do objeto: isto é, se os dois argumentos testados são o mesmo objeto. Em Java, é fácil criar duas strings que tenham os mesmos caracteres, mas não sejam o mesmo objeto string. Por exemplo: String foo1 = "foo";

String foo2 = String.valueOf(new char [] { 'f', 'o', 'o' });

foo1 == foo2 // falso!

foo1.equals(foo2) // verdadeiro

Esse erro é particularmente perigoso porque geralmente funciona no caso comum em que você está comparando strings literais (strings declaradas com aspas duplas diretamente no código). A razão para isso é que Java tenta gerenciar strings de forma eficiente, combinando-as. Em tempo de compilação, Java encontra todas as strings idênticas dentro de uma determinada classe e cria apenas um objeto para elas. Isso é seguro porque as strings são imutáveis e não podem ser alteradas, mas deixa espaço para esse problema de comparação.

O método compareTo() compara o valor léxico da String com outra String, usando a especificação Unicode para comparar as posições relativas de duas strings dentro do

“alfabeto”. (Usamos aspas porque o Unicode tem muito mais caracteres do que apenas

letras do alfabeto inglês.) Ele retorna um número inteiro menor, igual ou maior que zero:

String abc = "abc";

String def = "def";

String num = "123";

if (abc.compareTo(def) < 0) { ... } // verdadeiro

if (abc.compareTo(abc) == 0) { ... } // verdadeiro

if (abc.compareTo(num) > 0) { ... } // verdadeiro

Você realmente não pode usar o valor real retornado por compareTo() além dessas três possibilidades. Qualquer

número negativo, seja -1, -5 ou -1.000, significa simplesmente que a primeira string é “menor que” a segunda string. O método compareTo() compara strings estritamente pelas posições de seus caracteres na especificação Unicode. Isso funciona para texto simples, mas não lida bem com todas as variações de idioma. Se você precisar de comparações mais sofisticadas com suporte mais amplo à internacionalização, verifique a documentação paraa classe

java.text.Collator.

Procurando

A classe String fornece vários métodos simples para localizar substrings fixas dentro de uma string. Os métodos startWith() e endsWith() comparam uma string de argumento com o início e o fim da String, respectivamente:

String url = "http://foo.bar.com/";

if (url.startsWith("http:")) // verdadeiro

O método indexOf() procura a primeira ocorrência de um caractere ou substring e retorna a posição inicial do caractere, ou -1 se a substring não for encontrada: String abcs = "abcdefghijklmnopqrstuvwxyz";

int i = abcs.indexOf('p'); // 15

int i = abcs.indexOf("def"); //3

int I = abcs.indexOf("Fang"); // -1

Da mesma forma, lastIndexOf() pesquisa para trás na string a última ocorrência de um caractere ou substring.

O método contains() lida com a tarefa muito comum de verificar se uma determinada substring está contida na string de destino:

String log = "Há uma emergência no setor 7!";

if (log.contains("emergência")) pageSomeone();

// equivalente a

if (log.indexOf("emergência") != -1) ...

Para pesquisas mais complexas, você pode usar a API de Expressão Regular, que permite procurar e analisar padrões complexos. Falaremos sobre expressões regulares posteriormente neste capítulo.

Resumo do método String

Tabela 8-1resume os métodos fornecidos pela classe String. Incluímos vários métodos não discutidos neste capítulo. Sinta-se à vontade para experimentar esses métodos em jshell ou procurar odocumentação on-line.

Tabela 8-1. Métodos de string

Método

Funcionalidade

charAt()

Obtém um caractere específico na string

comparado a()

Compara a string com outra string

concat()

Concatena a string com outra string

contém()

Verifica se a string contém outra string

copyValueOf()

Retorna uma string equivalente ao array de caracteres especificado

termina com()

Verifica se a string termina com um sufixo especificado

é igual a()

Compara a string com outra string

equalsIgnoreCase()

Compara a string com outra string, ignorando maiúsculas e minúsculas

getBytes()

Copia caracteres da string em uma matriz de bytes

getChars()

Copia caracteres da string em uma matriz de caracteres

código hash()

Retorna um hashcode para a string

índice de()

Procura a primeira ocorrência de um caractere ou

substring na string

estagiário()

Busca uma instância exclusiva da string de um pool global

de strings compartilhadas

está em branco()

Retorna verdadeiro se a string tiver comprimento zero ou

contiver apenas espaços em branco

está vazia()

Retorna verdadeiro se a string tiver comprimento zero

lastIndexOf()

Pesquisa a última ocorrência de um caractere ou substring

em uma string

Método

Funcionalidade

comprimento()

Retorna o comprimento da string

linhas()

Retorna um fluxo de linhas separadas por terminadores

de linha

partidas()

Determina se a string inteira corresponde a um padrão de

expressão regular

regiãoMatches()

Verifica se uma região da string corresponde à região

especificada de outra string

repita()

Retorna uma concatenação desta string, repetida um

determinado número de vezes

substituir()

Substitui todas as ocorrências de um caractere na string

por outro caractere

substitua tudo()

Substitui todas as ocorrências de um padrão de expressão

regular por um padrão

substituirPrimeiro() Substitui a primeira ocorrência de um padrão de expressão regular por um padrão

dividir()

Divide a string em uma matriz de strings usando um padrão de expressão regular como delimitador

começa com()

Verifica se a string começa com um prefixo especificado

faixa()

Remove espaços em branco à esquerda e à direita, conforme definido porCharacter.isWhitespace()

stripLeading()

Remove os espaços em branco iniciais, semelhante a strip()

stripTrailing()

Remove espaços em branco à direita, semelhante a strip()

substring()

Retorna uma substring da string

toCharArray()

Retorna o array de caracteres da string

paraLowerCase()

Converte a string para minúscula

para sequenciar()

Retorna o valor da string de um objeto

paraUpperCase()

Converte a string para maiúscula

aparar()

Remove espaços em branco à esquerda e à direita,

definidos aqui como qualquer caractere com uma posição

Método

Funcionalidade

Unicode (chamada de ponto de código) menor ou igual a

32 (o caractere “espaço”)

valor de()

Retorna uma representação em string de um valor

Coisas de cordas

A análise e formatação de texto é um tópico amplo e aberto. Até agora neste capítulo, examinamos apenas operações primitivas em strings — criação, pesquisa e transformação de valores simples em strings. Agora gostaríamos de passar para formas de texto mais estruturadas. Java possui um rico conjunto de APIs para analisar e imprimir strings formatadas, incluindo números, datas, horas e valores monetários.

Abordaremos a maioria desses tópicos neste capítulo, mas aguardaremos para discutir a formatação de data e hora no“Datas e horários locais”.

Começaremos com a análise – lendo números e valores primitivos de strings e dividindo strings longas em tokens. Em seguida, daremos uma olhada nas expressões regulares, a ferramenta de análise de texto mais poderosa que o Java oferece. As expressões regulares permitem definir seus próprios padrões de complexidade arbitrária, pesquisá-los e analisá-los no texto.

Analisando Números Primitivos

Em Java, números, caracteres e booleanos são tipos primitivos – não objetos. Mas para cada tipo primitivo, Java também define uma classe wrapper primitiva.

Especificamente, o pacote java.lang inclui as seguintes classes: Byte, Short, Integer, Long, Float, Double, Character e Boolean. Falamos sobre isso em“Invólucros para tipos

primitivos”, mas vamos trazê-los à tona agora porque essas classes contêm métodos utilitários estáticos que analisam seus respectivos tipos a partir de strings. Cada uma dessas classes wrapper primitivas possui um método estático de “análise” que lê uma String e retorna o tipo primitivo correspondente. Por exemplo:

```
byte b = Byte.parseByte("16");

int n = Integer.parseInt("42");

longo l = Long.parseLong("99999999999");

float f = Float.parseFloat("4.2");
```

duplo d = Double.parseDouble("99.99999999");

boolean b = Boolean.parseBoolean("verdadeiro");

Você pode encontrar outras maneiras de converter strings em tipos base e vice-versa, mas esses métodos de classe wrapper são diretos e fáceis de ler. E no caso de Integer e Long, você também pode fornecer um argumento de base opcional (a base de um sistema numérico; os números decimais têm uma base de 10, por exemplo) para converter strings com números octais ou hexadecimais. (Dados não decimais às vezes aparecem ao lidar com itens como assinaturas criptográficas ou anexos de e-mail.)

Texto de tokenização

Você raramente encontrará strings com apenas um número para analisar ou apenas com a palavra necessária. É uma tarefa de programação mais comum analisar uma sequência mais longa de texto em palavras individuais, ou tokens, separados por algum conjunto de caracteres delimitadores, como espaços ou vírgulas.

Os programadores falam sobre tokens como uma forma genérica de discutir diferentes valores ou tipos presentes em um trecho de texto. Um token pode ser uma simples palavra, um nome de usuário, um endereço de e-mail ou um número. Vejamos alguns exemplos.

Considere o texto de exemplo abaixo. A primeira linha contém palavras separadas por espaços simples. O par de linhas restante envolve campos delimitados por vírgulas: Agora é a hora de todas as pessoas boas

Número do cheque, descrição, valor

4231, Programação Java, 1000,00

Java possui vários métodos e classes (infelizmente sobrepostos) para lidar com situações como essa. Usaremos o poderoso método split() da classe String. Ele utiliza expressões regulares para permitir quebrar uma string com base em padrões arbitrários. Falaremos sobre expressões regulares em breve, mas para mostrar como isso funciona, vamos apenas dar a mágica necessária agora.

O método split() aceita uma expressão regular que descreve um delimitador. Ele usa essa expressão para dividir a string em uma matriz de Strings menores: String text1 = "Agora é a hora de todas as pessoas boas";

String [] palavras = text1.split("\\s");

// palavras = "Agora", "é", "o", "tempo", ...

String text2 = "4231, Programação Java, 1000,00";

String [] campos = text2.split("\\s*,\\s*");

// campos = "4231", "Programação Java", "1000.00"

No primeiro exemplo, usamos a expressão regular \\s, que corresponde a um único caractere de espaço em branco (espaço, tabulação ou retorno de carro). Chamar split() em nossa variável text1 retorna um array de oito strings. No segundo exemplo, usamos uma expressão regular mais complicada, \\s*,\\s*, que corresponde a uma vírgula cercada por qualquer quantidade de espaços em branco opcionais. Isso reduziu nosso texto a três campos bonitos e organizados.

Expressões regulares

Agora é hora de fazer um breve desvio em nossa viagem por Java e entrar na terra das expressões regulares. Uma expressão regular, ou regex, descreve um padrão de texto.

Expressões regulares são usadas com muitas ferramentas — incluindo o pacote java.util.regex, editores de texto e muitas linguagens de script — para fornecer recursos sofisticados de pesquisa de texto e manipulação de strings.

Expressões regulares podem ajudá-lo a encontrar todos os números de telefone em um arquivo grande. Eles podem ajudá-lo a encontrar todos os números de telefone com um código de área específico. Eles podem ajudá-lo a encontrar todos os números de telefone que não possuem um código de área específico. Você pode usar uma expressão regular para localizar links no código-fonte de uma página da web. Você pode até usar expressões regulares para fazer algumas edições em um arquivo de texto. Você pode procurar números de telefone com o código de área entre parênteses, digamos (123) 456-7890, e substituí-lo pelo formato mais simples 123-456-7890, por exemplo. E a chave para o poder das expressões regulares é que você pode encontrar todos os números de telefone em seu bloco de texto entre parênteses e convertê-los –

não apenas um número de telefone específico.

Se você já está familiarizado com o conceito de expressões regulares e como elas são usadas em outras linguagens, talvez você queira dar uma olhada nesta seção, mas não a ignore completamente. No mínimo, você precisará olhar“A API java.util.regex”mais

adiante neste capítulo, que aborda as classes Java necessárias para usá-los. Se você está se perguntando o que são exatamente expressões regulares, pegue uma lata ou xícara de sua bebida favorita e prepare-se. Você está prestes a aprender sobre a ferramenta mais poderosa do arsenal de manipulação de texto, bem como sobre uma pequena linguagem dentro de uma linguagem, tudo em poucas páginas.

Notação Regex

Uma expressão regular (regex) descreve um padrão no texto. Por padrão, queremos dizer praticamente qualquer recurso que você possa imaginar identificar no texto apenas a partir dos caracteres literais, sem realmente compreender seu significado.

Isso inclui recursos como palavras, agrupamentos de palavras, linhas e parágrafos, pontuação, letras maiúsculas ou minúsculas e, mais geralmente, strings e números com uma estrutura específica. (Pense em coisas como números de telefone, endereços de e-mail ou códigos postais.) Com expressões regulares, você pode pesquisar no dicionário todas as palavras que têm a letra “q” sem o amigo “u” próximo a ela, ou palavras que começam e termine com a mesma letra. Depois de construir um padrão, você pode usar ferramentas simples para procurá-lo no texto ou para determinar se uma determinada string corresponde a ele.

Escreva uma vez, fuja

Expressões regulares constituem uma forma simples de linguagem de programação.

Pense por um momento nos exemplos que citamos anteriormente. Precisaríamos de algo como uma

linguagem para descrever até mesmo padrões simples – como endereços de e-mail – que tenham elementos comuns, mas também alguma variação na forma.

Um livro didático de ciência da computação classificaria as expressões regulares na parte inferior da hierarquia das linguagens de computação, tanto em termos do que elas podem descrever quanto do que você pode fazer com elas. Eles ainda são capazes de ser bastante sofisticados, no entanto. Tal como acontece com a maioria das linguagens de programação, os elementos das expressões regulares são simples, mas você pode combiná-los para criar algo bastante complexo. E é nessa complexidade potencial que as coisas começam a ficar complicadas.

Como as expressões regulares funcionam em strings, que podem ser encontradas em qualquer lugar no código Java, é conveniente ter uma notação muito compacta. Mas a notação compacta pode ser enigmática, e a experiência mostra que é muito mais fácil escrever uma declaração complexa do que lê-la novamente mais tarde. Essa é a maldição da expressão regular. Você pode se encontrar em um momento de inspiração movida a cafeína, escrevendo um único padrão glorioso para simplificar o resto do seu programa em uma linha. Quando você voltar a ler essa linha no dia seguinte, porém, ela poderá parecer-lhe hieróglifos egípcios. Geralmente, mais simples é melhor, mas se você puder dividir seu problema de forma mais clara em várias etapas, talvez deva.

Personagens escapados

Agora que você foi devidamente avisado, temos que lançar mais uma coisa em você antes de reconstruí-lo. A

notação regex não apenas pode ficar um pouco complicada, mas também é um tanto ambígua quando usada com strings Java comuns. Uma parte importante da notação é o caractere de escape – um caractere com uma barra invertida na frente dele. Por exemplo, na notação regex, o caractere d de escape, \d, (barra invertida “d”) é uma abreviação para qualquer caractere de um único dígito (0–

9). Entretanto, você não pode simplesmente escrever \d como parte de uma string Java, porque Java usa a barra invertida para seus próprios caracteres especiais e para especificar sequências de caracteres Unicode (\uxxxx). Felizmente, Java nos oferece um substituto: uma barra invertida com escape: duas barras invertidas (\\).

Representa uma barra invertida literal. A regra é que, quando você quiser que uma barra invertida apareça em sua regex, você deve escapar dela com uma barra extra:

"\\d" // string Java que produz \d em uma regex

Fica mais estranho! Como a própria notação regex usa uma barra invertida para denotar caracteres especiais, ela deve fornecer a mesma “saída de escape” para si mesma. Você precisa dobrar as barras invertidas se quiser que sua regex corresponda a uma barra invertida literal. Se parece com isso:

"\\\\" // String Java produz duas barras invertidas; regex produz um A maioria dos caracteres do operador “mágico” nesta seção operam no caractere que os precede, portanto, você deve escapá-los se quiser seu significado literal. Isso inclui caracteres como ., *, +, {} e (). Uma expressão que pode corresponder a números de

telefone formais dos EUA (com o código de área entre parênteses) é semelhante a esta:

"\\(\\d\\d\dd\\) \\d\\d\\d-\\d\\d\\d\\d"

Se você precisar criar parte de uma expressão que contenha muitos caracteres literais, poderá usar os delimitadores especiais \Q e \E para ajudá-lo. Qualquer texto que apareça entre \Q e \E será escapado automaticamente. (Você ainda precisa dos escapes Java String - barras invertidas duplas para barra invertida, mas não quádruplas.) Há também um método estático chamado Pattern.quote() que faz a mesma coisa, retornando uma versão com escape adequado de qualquer string que você fornecer.

Temos mais uma sugestão para ajudar a manter a calma ao trabalhar com esses exemplos. Escreva o regex simples usando uma linha de comentário acima da string Java real (onde você deve duplicar todas as barras invertidas). Também tendemos a incluir um comentário com um exemplo do texto que esperamos corresponder. Aqui está aquele exemplo de número de telefone dos EUA novamente com esta abordagem de comentários:

// Número de telefone dos EUA: (123) 456-7890

//regex: \(\d\d\d\) \d\d\d-\d\d\d\d

"\\(\\d\\d\dd\\) \\d\\d\\d-\\d\\d\\d\\d"

E não se esqueça do jshell! Pode ser um playground muito poderoso para testar e ajustar seus padrões. Veremos vários exemplos de padrões de teste com jshell em“A

API java.util.regex”. Mas primeiro, vamos dar uma olhada em mais elementos que você pode usar para construir padrões.

Personagens e classes de personagens

Agora, vamos mergulhar na sintaxe real da regex. A forma mais simples de uma expressão regular é um texto simples e literal, que não tem nenhum significado especial e é correspondido diretamente (caractere por caractere) na entrada. Pode ser um único caractere ou mais. Por exemplo, na string a seguir, o padrão s pode corresponder ao caractere “s” nas palavras “rosa” e “é”:

"Uma rosa custa US$ 1,99."

O padrão “rosa” pode corresponder apenas à palavra literal rosa. Mas isto não é muito interessante. Vamos aumentar um pouco as coisas introduzindo alguns personagens especiais e a noção de “classes” de personagens:

Qualquer caractere: ponto(.)

O caractere especial ponto (.) corresponde a qualquer caractere único. O padrão .ose corresponde a “rosa”, “nariz”, “_ose” (espaço seguido de “ose”) ou qualquer outro caractere seguido pela sequência “ose”. Dois pontos correspondem a quaisquer dois caracteres (“prosa”, “fechar”) e assim por diante. O operador ponto é muito amplo; normalmente para apenas para um terminador de linha (uma nova linha, retorno de carro ou uma combinação de ambos). Imagine . como representando a classe de todos os personagens.

Caractere de espaço em branco ou não-espaço em branco:\s, \S

O caractere especial \s corresponde ao espaço em branco. Espaço em branco inclui qualquer caractere relacionado ao espaço visual no texto ou que marque o final de uma linha. Caracteres de espaço em branco comuns incluem o caractere de espaço literal (o que você obtém quando pressiona a barra de espaço no teclado), \t (tab), \r (retorno de carro), \n (nova linha) e \f (formfeed). O caractere especial correspondente \S faz o inverso, correspondendo a qualquer caractere que não seja um espaço em branco.

***Caractere de dígito ou não dígito*: \d, \D**

\dcorresponde a qualquer um dos dígitos de 0 a 9. \D faz o inverso, combinando todos os caracteres, exceto os dígitos.

***Caractere de palavra ou não palavra*: \w, \W**

\ccorresponde a caracteres normalmente encontrados em “palavras”, como letras maiúsculas e minúsculas A–Z, a–z, os dígitos 0–9 e o caractere sublinhado (_). \W

corresponde a tudo, exceto a esses caracteres.

Classes de personagens personalizadas

Você pode definir suas próprias classes de caracteres usando colchetes ([ ]) ao redor dos caracteres desejados. aqui estão alguns exemplos:

[abcxyz] // corresponde a qualquer um de a, b, c, x, y ou z

[02468] // corresponde a qualquer dígito par

[aeiouAEIOU] // corresponde a qualquer vogal, maiúscula ou minúscula

[AaEeIiOoUu] // também corresponde a qualquer vogal

A notação especial de intervalo xy pode ser usada como abreviação para execuções consecutivas de caracteres alfanuméricos:

[LMNOPQ] // Classe explícita de L, M, N, O, P ou Q

[LQ] // Versão abreviada equivalente

[12345] // Classe explícita de 1, 2, 3, 4 ou 5

[1-5] // Versão abreviada equivalente

Colocar um sinal de intercalação (^) como o primeiro caractere entre colchetes inverte a classe de caracteres, correspondendo a qualquer caractere, exceto aqueles incluídos entre colchetes:

[^AF] // G, H, I, ..., a, b, c, 1, 2, $, #... etc.

[^aeiou] // Qualquer caractere que não seja uma vogal minúscula

Aninhar classes de caracteres simplesmente as concatena em uma única classe:

[AF[GZ]\s] // AZ mais espaço em branco

Você pode usar a notação lógica AND && (semelhante ao operador booleano que vimos em“Operadores”) para pegar o cruzamento (caracteres em comum):

[ap&&[lz]] // l, m, n, o, p

[AZ&&[^P]] // A a Z exceto P

Marcadores de posição

O padrão [Aa] rosa (incluindo um A maiúsculo ou minúsculo) corresponde três vezes na seguinte frase:

"Uma rosa é uma rosa é uma rosa"

Os caracteres de posição permitem designar a localização relativa de uma correspondência dentro de uma linha. Os mais importantes são ^ e $, que correspondem ao início e ao fim de uma linha, respectivamente:

^[Aa] rose // corresponde a "A rose" no início da linha

[Aa] rose$ // corresponde a "uma rosa" no final da linha

Para ser um pouco mais preciso, ^ e $ correspondem ao início e ao fim de “input”, que geralmente é uma única linha. Se você estiver trabalhando com múltiplas linhas de texto e desejar combinar o início e o fim das linhas em uma única string grande, poderá ativar o modo “multilinha” com um sinalizador, conforme descrito posteriormente em“Opções especiais”.

Os marcadores de posição \b e \B correspondem a um limite de palavra (espaço em branco, pontuação ou início ou fim de uma linha) ou a um limite que não seja de palavra (o meio de uma palavra), respectivamente. Por exemplo, o primeiro padrão corresponde a “rosa” e “alecrim”, mas não a “prímula”. O segundo padrão corresponde a “prímula” e “prosa”, mas não a “rosa” no início de uma palavra ou sozinho:

\brose // rosa, alecrim, rosas; NÃO prosa

\Brose // prosa, prímula; NÃO rosa ou alecrim

Você costuma usar esses marcadores de posição quando precisa procurar ou excluir prefixos ou sufixos.

Iteração (multiplicidade)

A simples correspondência de padrões de caracteres fixos não nos levará muito longe.

A seguir, veremos os operadores que contam o número de ocorrências de um caractere (ou, mais geralmente, de um padrão, como veremos em"Padrão"):

Qualquer (zero ou mais iterações): asterisco(*)

Colocar um asterisco (*) após um caractere ou classe de caractere significa “permitir qualquer número desse tipo de caractere” – em outras palavras, zero ou mais. Por exemplo, o padrão a seguir corresponde a um dígito com qualquer número de zeros à esquerda (possivelmente nenhum):

0*\d // corresponde a um dígito com qualquer número de zeros à esquerda

Algumas (uma ou mais iterações): sinal de mais(+)

O sinal de mais (+) significa “uma ou mais” iterações e é equivalente a XX* (padrão seguido de asterisco padrão). Por exemplo, o padrão a seguir corresponde a um número com um ou mais dígitos, além de zeros à esquerda opcionais:

0*\d+ // corresponde a um número (um ou mais dígitos) com opcional

// zeros à esquerda

Pode parecer redundante combinar os zeros no início de uma expressão porque zero é um dígito e, portanto, é correspondido pela porção \d+ da expressão de qualquer maneira. No entanto, mostraremos mais tarde como você pode separar a string usando uma regex e obter apenas as peças desejadas. Nesse caso, você pode querer retirar os zeros à esquerda e manter apenas os dígitos.

***Opcional (zero ou uma iteração): ponto de interrogação*(?)** O operador de ponto de interrogação (?) permite exatamente zero ou uma iteração.

Por exemplo, o padrão a seguir corresponde à data de vencimento de um cartão de crédito, que pode ou não ter uma barra no meio:

\d\d/?\d\d // corresponde a quatro dígitos com uma barra opcional no meio

***Intervalo (entre iterações x e y, inclusive)*: {x,y}**

O operador de intervalo de chaves {x,y} é o operador de iteração mais geral. Ele especifica um intervalo preciso para corresponder. Um intervalo leva dois argumentos: um limite inferior e um limite superior, separados por vírgula. Esta regex corresponde a qualquer palavra com cinco a sete caracteres, inclusive:

\b\w{5,7}\b // combina palavras com 5, 6 ou 7 caracteres

***Pelo menos x ou mais iterações (y é infinito)*: {x,}**

Se você omitir o limite superior, simplesmente deixando uma vírgula pendente no intervalo, o limite superior se tornará infinito. Esta é uma forma de especificar um mínimo de ocorrências sem máximo.

Alternância

O operador de barra vertical (|) denota a operação lógica OR, também chamada de alternância ou escolha. O | operador não opera em caracteres individuais, mas se aplica a tudo em ambos os lados dele. Ele divide a expressão em duas, a menos que seja restringido pelo agrupamento entre parênteses. Por exemplo, uma abordagem um pouco ingênua para analisar datas pode ser a seguinte:

\w+, \w+ \d+, \d+|\d\d/\d\d/\d\d // padrão 1 OU padrão 2

Nesta expressão, a esquerda corresponde a padrões como “Sexta-feira, 12 de outubro de 2001” e a direita corresponde a “12/10/01”.

O seguinte regex pode ser usado para combinar endereços de e-mail com um dos três domínios (net, edu e gov):

\w+@[\w.]+\.(rede|edu|gov)

// endereço de e-mail terminado em .net, .edu ou .gov

Este padrão não é de forma alguma completo em termos de endereços de e-mail verdadeiros e válidos. Mas destaca como você pode usar a alternância para ajudar a construir expressões regulares com algumas características úteis.

Opções especiais

Várias opções especiais afetam a forma como o mecanismo regex realiza sua correspondência. Essas opções podem ser aplicadas de duas maneiras:

•

Você pode fornecer um ou mais argumentos especiais (sinalizadores) para a etapa Pattern.compile() (discutida em“A API java.util.regex”).

•

Você pode incluir um bloco especial de código em seu regex.

Mostraremos a última abordagem aqui. Para fazer isso, inclua um ou mais sinalizadores em um bloco especial (?x), onde x é o sinalizador da opção que queremos ativar. Geralmente, você faz isso no início da regex. Você também pode desativar sinalizadores adicionando um sinal de menos (?-x), que permite aplicar sinalizadores para selecionar partes do seu padrão.

Os seguintes sinalizadores estão disponíveis:

***Não diferencia maiúsculas de minúsculas*: (?eu)**

O sinalizador (?i) diz ao mecanismo regex para ignorar maiúsculas e minúsculas durante a correspondência. Por exemplo:

(?i)yahoo // corresponde ao Yahoo, yahoo, yahOO, etc.

***Ponto tudo*: (?s)**

O sinalizador (?s) ativa o modo “ponto tudo”, permitindo que o caractere ponto corresponda a qualquer coisa, incluindo caracteres de fim de linha. É útil se você estiver combinando padrões que abrangem várias linhas. O s significa “modo de linha única”, um nome um tanto confuso derivado de Perl.

Multilinha: (?m)

Por padrão, ^ e $ não correspondem realmente ao início e ao fim das linhas (conforme definido por retorno de carro ou combinações de nova linha). Em vez disso, eles correspondem ao início ou ao final de todo o texto de entrada. Em muitos casos, “uma linha” é sinônimo de toda a entrada.

Se você tiver um grande bloco de texto para processar, muitas vezes dividirá esse bloco em linhas separadas por outros motivos. Se você fizer isso, verificar qualquer linha em busca de uma expressão regular será simples e ^ e $ se comportarão conforme o esperado. No entanto, se você quiser usar um regex com toda a string de entrada contendo múltiplas linhas (separadas por essas combinações de retorno de carro ou nova linha), você pode ativar o modo multilinha com (?m). Este sinalizador faz com que ^ e $ correspondam ao início e ao fim das linhas individuais dentro do

bloco de texto, bem como ao início e ao fim de todo o bloco. Especificamente, isso significa o ponto antes do primeiro caractere, o ponto após o último caractere e os pontos logo antes e depois dos terminadores de linha dentro da string.

Linhas Unix: (?d)

O sinalizador (?d) limita a definição do terminador de linha para ^, $ e . caracteres especiais apenas para nova linha no estilo Unix (\n). Por padrão, nova linha com retorno de carro (\r\n) também é permitida.

A API java.util.regex

Agora que cobrimos a teoria de como construir expressões regulares, a parte difícil acabou. Tudo o que resta é investigar a API Java para ver como aplicar essas expressões.

Padrão

Como dissemos, os padrões regex que escrevemos como strings são, na verdade, pequenos programas que descrevem como combinar texto. Em tempo de execução, o pacote Java regex compila esses pequenos programas em um formato que pode ser executado em algum texto de destino. Vários métodos simples de conveniência aceitam strings diretamente para serem usadas como padrões.

O método estático Pattern.matches() pega duas strings – uma regex e uma string de destino – e determina se o destino corresponde à regex. Isso é muito conveniente se você quiser fazer um teste rápido uma vez em seu aplicativo. Por exemplo: Correspondência booleana = Pattern.matches("\\d+\\.\\d+f?", meuTexto); Esta linha de código pode testar se a string myText contém um número de ponto flutuante no estilo Java, como “42.0f”. Observe que a string deve corresponder completamente para ser considerada uma correspondência. Se você quiser ver se um padrão pequeno está contido em uma string maior, mas não se importa com o resto da string, você terá que usar um Matcher, conforme descrito em“O Combinado”.

Vamos tentar outro padrão (simplificado) que poderíamos usar em nosso jogo quando começarmos a permitir que vários jogadores compitam entre si. Muitos sistemas de login usam endereços de e-mail como identificador de

usuário. Esses sistemas não são perfeitos, é claro, mas um endereço de e-mail atenderá às nossas necessidades.

Gostaríamos de convidar os usuários a inserirem seus endereços de e-mail, mas queremos ter certeza de que ele parece válido antes de usá-lo. Uma expressão regular pode ser uma maneira rápida de realizar essa validação.3

Assim como escrever algoritmos para resolver problemas de programação, projetar uma expressão regular exige que você divida seu problema de correspondência de padrões em pedaços pequenos. Se pensarmos em endereços de e-mail, alguns padrões se destacam imediatamente. O mais óbvio é o @ no meio de cada endereço. Um padrão ingênuo (mas melhor que nada!) baseado nesse fato poderia ser construído assim:

String sample = " meu.nome@algum.domínio "; Boolean validEmail = Pattern.matches(".*@.*", amostra);

Mas esse padrão é muito permissivo. Certamente reconhecerá endereços de e-mail válidos, mas também reconhecerá muitos endereços inválidos, como "bad.address@"

ou "@also.bad" ou mesmo "@@". Vamos testar isso em jshell: jshell> String sample = " meu.nome@algum.domínio ";

amostra ==> " meu.nome@algum.domínio "

jshell> Pattern.matches(".*@.*", amostra)

Pattern.matches(".*@.*", amostra)$2 ==> verdadeiro

jshell> Pattern.matches(".*@.*", "bad.address@") Pattern.matches(".*@.*", "bad.address@")$3 ==> verdadeiro jshell> Pattern.matches(".*@.*", "@@")

Pattern.matches(".*@.*", "@@")$4 ==> verdadeiro Tente inventar mais alguns exemplos ruins de sua autoria. Você verá rapidamente que nosso padrão de e-mail simples é definitivamente muito simples.

Como podemos fazer combinações melhores? Um ajuste rápido seria usar o modificador + em vez do *. O padrão atualizado agora requer pelo menos um caractere em cada lado do @. Mas sabemos algumas outras coisas sobre endereços de e-mail. Por exemplo, a “metade” esquerda do endereço (a parte do nome) não pode conter o caractere @. Aliás, a parte do domínio também não. Podemos usar uma classe de personagem personalizada para esta próxima atualização:

String sample = " meu.nome@algum.domínio ";

Boolean validEmail = Pattern.matches("[^@]+@[^@]+", amostra); Esse padrão é melhor, mas ainda permite vários endereços inválidos, como

"still@bad", já que os nomes de domínio têm pelo menos um nome seguido por um ponto (.) seguido por um domínio de nível superior (TLD), como “oreilly.com”. Então, talvez um padrão como este:

String sample = " meu.nome@algum.domínio ";

Boolean validEmail = Pattern.matches("[^@]+@[^@]+\\.(com|org)", amostra); Esse padrão corrige nosso problema com um endereço como "ainda@ruim", mas fomos longe demais no sentido contrário. Existem muitos, muitos TLDs – muitos para serem listados

razoavelmente, mesmo se ignorarmos o problema de manter essa lista à medida que novos TLDs são adicionados.4Então, vamos recuar um pouco.

Manteremos o “ponto” na parte do domínio, mas removeremos o TLD específico e aceitaremos apenas uma sequência simples de letras:

String sample = " meu.nome@algum.domínio ";

Boolean validEmail = Pattern.matches("[^@]+@[^@]+\\.[az]+", amostra);

Muito melhor. Podemos adicionar um último ajuste para garantir que não nos preocupemos com a caixa do endereço, já que todos os endereços de e-mail não diferenciam maiúsculas de minúsculas. Basta inserir o sinalizador (?i) no início de nossa string padrão:

String sample = " meu.nome@algum.domínio ";

Boolean validEmail = Pattern.matches("(?i)[^@]+@[^@]+\\.[az]+", amostra); Novamente, este não é de forma alguma um validador de e-mail perfeito, mas é definitivamente um bom começo e é suficiente para nosso hipotético sistema de login: jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[az]+", " good@some.domain ") $1 ==> verdadeiro

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[az]+", " good@oreilly.com ") $2 ==> verdadeiro

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[az]+", "oreilly.com") $3 ==> falso

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[az]+", "bad@oreilly@com") $4 ==> falso

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[az]+", " me@oreilly.COM ") $ 5 ==> verdadeiro

jshell> Pattern.matches("[^@]+@[^@]+\\.[az]+", " me@oreilly.COM ") $6 ==> falso

Nestes exemplos, digitamos a linha Pattern.matches(…) completa apenas uma vez.

Depois disso, foi uma simples seta para cima, edite e pressione Return para as cinco linhas subsequentes. Você consegue identificar a falha em nosso padrão final que faz com que a partida falhe?

Observação

Se você quiser mexer no padrão de validação e expandi-lo ou melhorá-lo, lembre-se de que você pode “reutilizar” linhas no jshell com as teclas de seta do teclado. Use a seta para cima para recuperar a linha anterior. Na verdade, você pode usar as setas para cima e para baixo para navegar por todas as suas linhas recentes. Dentro de uma linha, use a seta para a esquerda e a seta para a direita para mover e excluir/adicionar/editar seu comando. Em seguida, basta pressionar a tecla Return para executar o comando recém-alterado – você não precisa mover o cursor até o final da linha antes de pressionar Return.

O combinador

Um Matcher associa um padrão a uma string e fornece ferramentas para testar, localizar e iterar correspondências do padrão com ele. O Matcher é “com estado”. Por

exemplo, o método find() tenta encontrar a próxima correspondência cada vez que é chamado. Mas você pode limpar o Matcher e começar de novo chamando seu método reset().

Para criar um objeto Matcher, primeiro você precisa compilar sua string padrão em um objeto Pattern usando o método estático Pattern.compile(). Com esse objeto padrão em mãos, você pode usar o método matcher() para obter seu Matcher, assim: String myText = "Muito texto com hiperlinks e outras coisas..."; Padrão urlPattern = Pattern.compile("https?://[\\w./]*");

Correspondente correspondente = urlPattern.matcher(meuTexto);

Se você está interessado apenas em “uma grande correspondência” – isto é, espera que sua string corresponda ao padrão ou não – você pode usar matches() ou LookingAt(). Eles correspondem aproximadamente aos métodos equals() e startupsWith() da classe String. O método matches() pergunta se a string corresponde ao padrão em sua totalidade (sem nenhum caractere de string sobrando) e retorna verdadeiro ou falso. O método LookingAt() faz o mesmo, exceto que pergunta apenas se a string começa com o padrão e não se importa se o padrão usa todos os caracteres da string.

De maneira mais geral, você desejará pesquisar na string e encontrar uma ou mais correspondências. Para fazer isso, você pode usar o método find(). Cada chamada para find() retorna verdadeiro ou falso para a próxima correspondência do padrão e anota internamente a posição do texto correspondente. Você pode obter as posições dos caracteres inicial e final com os métodos

matcher start() e end() ou pode simplesmente recuperar o texto correspondente com o método group(). Por exemplo: importar java.util.regex.*;

// ...

String text="Um cavalo é um cavalo, claro, claro...";

String pattern="cavalo|curso";

Matcher matcher =
Pattern.compile(pattern).matcher(texto);

enquanto (matcher.find())

Sistema.out.println(

"Correspondido: '"+matcher.group()+"' na posição "+matcher.start()); O trecho anterior imprime o local inicial das palavras “cavalo” e “curso” (quatro no total):

Correspondido: 'cavalo' na posição 2

Correspondido: 'cavalo' na posição 13

Correspondido: 'curso' na posição 23

Correspondido: 'curso' na posição 33

Divisão de strings

Uma necessidade muito comum é analisar uma string em vários campos com base em algum delimitador, como uma vírgula. É um problema tão comum que a classe String contém um método para fazer exatamente isso. O método split() aceita uma expressão regular e retorna uma matriz de substrings quebradas em torno desse padrão.

Considere as seguintes chamadas de string e split():

String text = "Foo, bar, blá";

String[] badFields = text.split(",");

// { "Foo", "bar", "blá" }

String[] camposbons = text.split("\\s*,\\s*");

// { "Foo", "bar", "blá" }

O primeiro split() retorna um array String, mas o uso ingênuo de , para separar a string significa que os caracteres de espaço em nossa variável de texto permanecem presos aos caracteres mais interessantes. Obtemos “Foo” como uma única palavra, como esperado, mas depois obtemos “bar<space>” e finalmente

“<space><space><space>blah”. Caramba! O segundo split() também produz um array String, mas desta vez contendo o esperado “Foo”, “bar” (sem espaço à direita) e “blah”

(sem espaços à esquerda).

Se você for usar uma operação como essa mais de algumas vezes em seu código, provavelmente deverá compilar o padrão e usar seu método split(), que é idêntico à versão em String. O método String split() é equivalente a:

Padrão.compile(padrão).split(string);

Como observamos antes, há muito o que aprender sobre expressões regulares além dos poucos recursos de regex que abordamos aqui. Confira a documentação sobre

padrões. Experimente sozinho usando jshell. Modifique o arquivo ch08/examples/ValidEmail.java e veja se você consegue criar um validador de e-mail melhor! Este é definitivamente um tópico que se beneficia da prática.

Utilitários matemáticos

É claro que a manipulação de strings e a correspondência de padrões não são os únicos tipos de operações que o Java pode realizar. Java suporta aritmética inteira e de ponto flutuante diretamente na linguagem. Operações matemáticas de nível superior são suportadas pela classe java.lang.Math. Como você viu, as classes wrapper para tipos de dados primitivos permitem tratá-los como objetos. As classes wrapper também contêm alguns métodos para conversões básicas.

Vamos começar dando uma rápida olhada na aritmética integrada em Java. Java lida com erros na aritmética inteira lançando uma ArithmeticException:

```
interno zero = 0;

tentar {

int i = 72/zero;

} catch (ArithmeticException e) {

// divisão por zero

}
```

Para gerar o erro neste exemplo, criamos a variável intermediária zero. O compilador é um tanto astuto. Ter-nos-íamos apanhado se tivéssemos tentado dividir por 0

diretamente.

As expressões aritméticas de ponto flutuante, por outro lado, não lançam exceções.

Em vez disso, eles assumem os valores especiais fora da faixa mostrados emTabela 8-2.

Tabela 8-2. Valores especiais de ponto flutuante

Valor

Representação matemática

POSITIVO_INFINITY 1,0/0,0

NEGATIVE_INFINITY -1,0/0,0

NaN

0,0/0,0

O exemplo a seguir gera um resultado infinito:

zero duplo = 0,0;

duplo d = 1,0/zero;

se (d == Duplo.POSITIVE_INFINITY)

System.out.println("Divisão por zero");

O valor especial NaN (não um número) indica o resultado da divisão de zero por zero.

Este valor tem a distinção matemática especial de não ser igual a si mesmo (NaN !=

NaN é avaliado como verdadeiro). Use Float.isNaN() ou Double.isNaN() para testar NaN.

A classe java.lang.Math

A classe java.lang.Math é a biblioteca matemática do Java. Ele contém um conjunto de métodos estáticos que cobrem todas as operações matemáticas usuais como sin(), cos() e sqrt(). A classe Math não é muito orientada a objetos (você não pode criar uma instância de Math). Em vez disso, é apenas um suporte conveniente para métodos estáticos que são mais parecidos com funções globais. Como vimos emcapítulo 5, é possível usar a funcionalidade de importação estática para importar os nomes de métodos estáticos e constantes como esse diretamente para o escopo de nossa classe e usá-los por seus nomes simples e não qualificados.

Tabela 8-3resume os métodos em java.lang.Math.

Tabela 8-3. Métodos em java.lang.Math

Tipo(s) de

Método

argumento

Funcionalidade

Matemática.abs(a)

interno, longo,

Valor absoluto

flutuante, duplo

Math.acos(a)

dobro

Arco cosseno

Matemática.asin(a)

dobro

Arco senoidal

Matemática.atan(a)

dobro

Tangente de arco

Matemática.atan2(a,b)

dobro

Parte do ângulo da transformação

de coordenadas retangulares para

polares

Matemática.ceil(a)

dobro

Menor número inteiro maior ou

igual a

Matemática.cbrt(a)

dobro

Raiz cúbica de um

Matemática.cos(a)

dobro

Cosseno

Matemática.cosh(a)

dobro

Cosseno hiperbólico

Matemática.exp(a)

dobro

Matemática.Eao poder um

Math.floor(a)

dobro

Maior número inteiro menor ou

igual a

Matemática.hipot(a,b)

dobro

Cálculo de precisão do sqrt() de a2

+ b2

Math.log(a)

dobro

Logaritmo natural de um

Math.log10(a)

dobro

Log de base 10 de um

Matemática.max(a, b)

interno, longo,

O valor a ou b mais próximo de

flutuante, duplo

Long.MAX_VALUE

Matemática.min(a, b)

interno, longo,

O valor a ou b mais próximo de

flutuante, duplo

Long.MIN_VALUE

Matemática.pow(a, b)

dobro

aao poder b

Matemática.random()

Nenhum

Gerador de números aleatórios

Matemática.rint(a)

dobro

Converte valor duplo em valor

integral em formato duplo

Tipo(s) de

Método

argumento

Funcionalidade

Rodada matemática (a)

flutuador,

Arredonda para número inteiro

dobro

Matemática.signum(a)

flutuador,

Obtenha o sinal do número em 1,0,

dobro

–1,0 ou 0

Matemática.sin(a)

dobro

Seno

Matemática.sinh(a)

dobro

Seno hiperbólico

Matemática.sqrt(a)

dobro

Raiz quadrada

Matemática.tan(a)

dobro

Tangente

Matemática.tanh(a)

dobro

Tangente hiperbólica

Matemática para graus

dobro

Converter radianos em graus

(a)

Math.toRadians(a)

dobro

Converter graus em radianos

Os métodos log(), pow() e sqrt() podem lançar uma ArithmeticException em tempo de execução. Os métodos abs(), max() e min() são sobrecarregados para todos os valores escalares (int, long, float e double) e retornam o tipo correspondente. Versões de Math.round() aceitam float ou double e retornam int ou long, respectivamente. O resto dos métodos operam e retornam valores duplos:

duplo irracional = Math.sqrt(2.0); //1.414...

int maior = Math.max(3, 4); //4

longo = Math.round(1.125798); //1

Apenas para destacar a conveniência dessa opção de importação estática, experimente estas funções simples no jshell:

jshell> importar estático java.lang.Math.*

jshell> duplo irracional = sqrt(2.0)

irracional ==> 1.4142135623730951

jshell> int maior = max(3,4)

maior ==> 4

jshell> longo = redondo (1,125798)

um ==> 1

Matemáticatambém contém as constantes duplas finais estáticas E e PI. Para encontrar o perímetro de um círculo, por exemplo:

circunferência dupla = diâmetro * Math.PI;

Matemática em ação

Já mencionamos o uso da classe Math e seus métodos estáticos em "Acessando Campos e Métodos". Podemos usá-lo novamente para tornar nosso jogo um pouco mais divertido, aleatoriamente onde as árvores aparecem. O método Math.random() retorna um duplo aleatório maior ou igual a 0 e menor que 1. Adicione um pouco de aritmética e arredondamento ou truncamento e você pode usar esse valor para criar números aleatórios em qualquer intervalo que precisar. Em particular, a conversão deste valor em um intervalo desejado segue esta fórmula:

```
int randomValue = min + (int)(Math.random() * (máx - min));
```

Tente! Tente gerar um número aleatório de quatro dígitos no jshell. Você pode definir o mínimo para 1.000 e o máximo para 10.000, assim:

```
jshell> int min = 1000

min ==> 1000

jshell> int máximo = 10000

máximo ==> 10000

jshell> int fourDigit = min + (int)(Math.random() * (max - min))

quatro dígitos ==> 9603

jshell> fourDigit = min + (int)(Math.random() * (max - min))
```

quatro dígitos ==> 9178

jshell> fourDigit = min + (int)(Math.random() * (max - min))

quatro dígitos ==> 3789

Para posicionar nossas árvores, precisaremos de dois números aleatórios para as coordenadas xey. Podemos definir um intervalo que manterá as árvores na tela pensando em uma margem nas bordas. Para a coordenada x, uma maneira de fazer isso pode ser assim:

privado int bomX() {

// pelo menos metade da largura da árvore mais alguns pixels

int leftMargin=Field.TREE_WIDTH_IN_PIXELS/2+5;

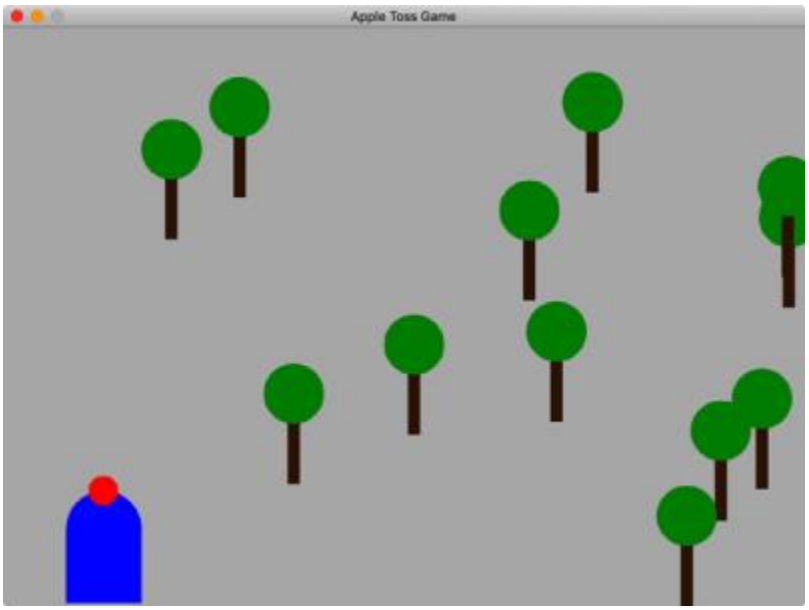
// agora encontre um número aleatório entre as margens esquerda e direita int rightMargin = FIELD_WIDTH - leftMargin;

// E retorna um número aleatório começando na margem esquerda

return leftMargin + (int)(Math.random() * (rightMargin - leftMargin));

}

Configure um método semelhante para encontrar um valor y e você deverá começar a ver algo como a imagem mostrada emFigura 8-1. Você pode até se divertir e usar o método isTouching() que discutimos anteriormentecapítulo 5para evitar colocar



árvores em contato direto com nosso físico. Aqui está nosso loop de configuração de árvore atualizado:

for (int i = field.trees.size(); i < Field.MAX_TREES; i++) {

Árvore t = nova Árvore();

t.setPosition(bomX(), bomY());

// As árvores podem estar próximas umas das outras e sobrepor-se,

// mas eles não deveriam cruzar nosso físico

enquanto(player1.isTouching(t)) {

// Nós cruzamos esta árvore, então vamos tentar novamente

System.err.println("Reposicionando uma árvore de interseção..."); t.setPosition(bomX(), bomY());

}

campo.addTree(t);

}

Figura 8-1. Árvores distribuídas aleatoriamente

Tente sair do jogo e iniciá-lo novamente. Você deverá ver as árvores em lugares diferentes cada vez que executar o aplicativo.

Números grandes/precisos

Se os tipos long e double não forem grandes ou precisos o suficiente para você, o pacote java.math fornece duas classes, BigInteger e BigDecimal, que suportam números de precisão arbitrária. Essas classes completas possuem vários métodos para executar operações de precisão arbitrária.5matemática e controlar com precisão o arredondamento dos restos. No exemplo a seguir, usamos BigDecimal para somar dois números muito grandes e depois criar uma fração com resultado de 100 dígitos: l1 longo = 92233720368547758O7L; // Longo.MAX_VALUE

longo l2 = 92233720368547758O7L;

System.out.println(l1 + l2); // -2 ! Não é bom.

tentar {

BigDecimal bd1 = novo BigDecimal("92233720368547758O7");

BigDecimal bd2 = novo BigDecimal(92233720368547758O7L);

```
System.out.println(bd1.add(bd2) );
//18446744073709551614

Numerador BigDecimal = novo BigDecimal(1);

Denominador BigDecimal = novo BigDecimal(3);

Grande fração decimal =

numerador.divide(denominador, 100,
BigDecimal.ROUND_UP);

// fração de 100 dígitos = 0,333333 ... 3334

}

pegar (NumberFormatException nfe) { }

pegar (ArithmeticException ae) { }
```

Se você implementa algoritmos criptográficos ou científicos por diversão, o BigInteger é crucial. O BigDecimal, por sua vez, pode ser encontrado em aplicações que lidam com dados monetários e financeiros. Fora isso, provavelmente você não precisará dessas aulas.

Datas e horários

Trabalhar com datas e horários sem as ferramentas adequadas pode ser uma tarefa árdua. Java inclui três classes que tratam de casos simples para você. A classe java.util.Date encapsula um momento no tempo. A classe java.util.GregorianCalendar, que estende o java.util.Calendar abstrato, traduz entre um ponto no tempo e campos de calendário como mês, dia e ano. Finalmente, a classe java.text.DateFormat sabe como

gerar e analisar representações de strings de datas e horas em vários idiomas e localidades.

Embora as classes Data e Calendário cobrissem muitos casos de uso, faltava granularidade e outros recursos. Várias bibliotecas de terceiros surgiram, todas com o objetivo de tornar mais fácil para os desenvolvedores trabalharem com datas, horas e durações de tempo. O Java 8 forneceu melhorias muito necessárias nesta área com a adição do pacote java.time. O restante deste capítulo explora esse pacote, mas você ainda encontrará muitos, muitos exemplos de Data e Calendário disponíveis, por isso é útil saber que eles existem. Como sempre, odocumentos on-linesão uma fonte inestimável para revisar partes da API Java que não abordamos aqui.

Datas e horários locais

A classe java.time.LocalDate representa uma data sem informações de hora para sua região local. Pense em um evento anual, como o solstício de inverno em 21 de dezembro. Da mesma forma, java.time.LocalTime representa um horário sem qualquer informação de data. Talvez o seu despertador toque às 7h15 todas as manhãs. O java.time.LocalDateTime armazena valores de data e hora para coisas como consultas com seu oftalmologista (para que você possa continuar lendo livros sobre Java). Todas essas classes oferecem métodos estáticos para criar novas instâncias, usando valores numéricos apropriados com o método of() ou analisando strings com parse(). Vamos entrar no jshell e tentar criar alguns exemplos:

```
jshell> importar java.time.*

jshell>LocalDate.of(2019,5,4)
```

$ 2 ==> 04/05/2019

jshell> LocalDate.parse("04/05/2019")

$ 3 ==> 04/05/2019

jshell> LocalTime.of(7,15)

$4 ==> 07:15

jshell> LocalTime.parse("07:15")

US$ 5 ==> 07:15

jshell>LocalDateTime.of(2019,5,4,7,0)

$ 6 ==> 04-05-2019T07:00

jshell> LocalDateTime.parse("2019-05-04T07:15")

US$ 7 ==> 04/05/2019T07:15

Outro ótimo método estático para criar esses objetos é now(), que fornece a data, hora ou data e hora atuais, conforme esperado:

jshell> LocalTime.now()

$ 8 ==> 15:57:24,052935

jshell>LocalDate.now()

US$ 9 ==> 31/03/2023

jshell>LocalDateTime.now()

US$ 10 ==> 31/03/2023T15:57:37.909038

Ótimo! Após importar o pacote java.time, você pode criar instâncias de cada uma das classes Local… para momentos específicos ou para “agora”. Você deve ter notado que os objetos criados com now() incluem segundos e nanossegundos no tempo. Você pode fornecer esses valores aos métodos of() e parse() se desejar ou precisar deles.

Não é muito emocionante aí, mas depois de ter esses objetos, você pode fazer muito com eles. Leia!

Comparando e manipulando datas e horários

Uma das grandes vantagens de usar classes java.time é o conjunto consistente de métodos disponíveis para comparar e alterar datas e horas. Por exemplo, muitos aplicativos de bate-papo mostram “há quanto tempo” uma mensagem foi enviada. O

subpacote java.time.temporal tem exatamente o que precisamos: a interface ChronoUnit. Ele contém várias unidades de data e hora, como MESES, DIAS, HORAS, MINUTOS, etc. Essas unidades podem ser usadas para calcular diferenças. Por exemplo, poderíamos calcular quanto tempo leva para criar dois exemplos de data e hora em jshell usando o método between():

```
jshell> LocalDateTime primeiro = LocalDateTime.now()

primeiro ==> 31/03/2023T16:03:21.875196

jshell> LocalDateTime segundo = LocalDateTime.now()

segundo ==> 2023-03-31T16:03:33.175675

jshell> importar java.time.temporal.*
```

jshell> ChronoUnit.SECONDS.between(primeiro, segundo)

$ 12 ==> 11

Uma verificação visual mostra que realmente demorou cerca de 11 segundos para digitar a linha que criou nossa segunda variável. Confira adocumentos para

ChronoUnitpara obter uma lista completa, mas você obtém a gama completa, de nanossegundos a milênios.

Essas unidades também podem ajudá-lo a manipular datas e horas com os métodos plus() e minus(). Para definir um lembrete para uma semana a partir de hoje, por exemplo:

jshell> LocalDate hoje = LocalDate.now()

hoje ==> 31/03/2023

jshell> Lembrete LocalDate = hoje.plus(1, ChronoUnit.WEEKS)

lembrete ==> 2023-04-07

Organizado! Mas este exemplo de lembrete traz outra manipulação que você pode precisar realizar de vez em quando. Você pode querer um lembrete em um horário específico no dia 7. Você pode converter entre datas, horas e datas com bastante facilidade com os métodos atDate() ou atTime():

jshell> LocalDateTime melhorReminder = lembrete.atTime(LocalTime.of(9,0)) melhorLembrete ==> 2023-04-07T09:00

Agora você receberá um lembrete às 9h. Exceto, e se você definir esse lembrete em Atlanta e depois voar para São Francisco? Quando o alarme dispararia?

LocalDateTime é, bem, local! Portanto, a parte T09:00 ainda é 9h, onde quer que você esteja quando executa o programa. Mas se você estiver lidando com algo como agendar uma reunião, não poderá ignorar os diferentes fusos horários envolvidos.

Felizmente, o pacote java.time também pensou nisso.

Fusos horários

Os autores do novo pacote java.time incentivam você a usar variações locais das classes de hora e data sempre que possível. Adicionar suporte para fusos horários significa adicionar complexidade ao seu aplicativo – eles querem que você evite essa complexidade, se possível. Mas há muitos cenários em que o suporte a fusos horários é inevitável. Você pode trabalhar com datas e horas “zoneadas” usando as classes ZonedDateTime e OffsetDateTime. A variante zoneada entende fusos horários nomeados e coisas como ajustes de horário de verão. A variante de deslocamento é um deslocamento numérico simples e constante de UTC/Greenwich.

A maioria dos usos de datas e horas voltados para o usuário usará a abordagem de zona nomeada, então vamos dar uma olhada na criação de uma data e hora zoneada.

Para anexar uma zona, usamos a classe ZoneId, que possui o método estático of() comum para criar novas instâncias. Você pode fornecer uma zona de região como uma String para obter o valor zoneado:

jshell> LocalDateTime piLocal =
LocalDateTime.parse("2023-03-14T01:59") piLocal ==>
2023-03-14T01:59

jshell> ZonedDateTime piCentral =
piLocal.atZone(ZoneId.of("América/Chica go"))

piCentral ==> 2023-03-14T01:59-
05:00[América/Chicago]

E agora você pode fazer coisas como garantir que seus amigos em Paris possam se juntar a você no momento correto usando o método withZoneSameInstant() detalhado, mas apropriadamente chamado:

jshell> ZonedDateTime piAlaMode =

piCentral.withZoneSameInstant(ZoneId.of("Europa/Paris")
)

piAlaMode ==> 2023-03-14T07:59+01:00[Europa/Paris]

Se você tem outros amigos que não estão convenientemente localizados em uma grande região metropolitana, mas deseja que eles também participem, você pode usar o método systemDefault() de ZoneId para escolher seu fuso horário

programaticamente:

jshell> ZonedDateTime piOther =

piCentral.withZoneSameInstant(ZoneId.systemDefault())

piOutro ==> 2023-03-14T02:59-
04:00[América/Nova_Iorque]

Estávamos executando o jshell em um laptop no fuso horário do Leste dos Estados Unidos. piOther sai exatamente como esperado. O ID da zona systemDefault() é uma maneira muito útil de adaptar rapidamente datas e horas de outras zonas para corresponder ao que o relógio e o calendário do usuário provavelmente dirão. Em aplicativos comerciais, você pode permitir que o usuário informe sua zona preferida, mas systemDefault() geralmente é um bom palpite.

Análise e formatação de datas e horas

Para criar e mostrar nossos datas e horários locais e zoneados usando strings, contamos com os formatos padrão que seguem os valores ISO. Geralmente funcionam sempre que precisamos aceitar ou exibir datas e horas. Mas como todo programador sabe, “geralmente” não é “sempre”. Felizmente, você pode usar a classe de utilitário java.time.format.DateTimeFormatter para ajudar na análise de entrada e formatação de saída.

O núcleo do DateTimeFormatter centra-se na construção de uma string de formato que rege a análise e a formatação. Você constrói seu formato com as peças listadas emTabela 8-4. Estamos listando apenas uma parte das opções disponíveis aqui, mas

elas devem ajudá-lo na maior parte das datas e horários que você encontrará. Observe que o caso é importante ao usar os caracteres mencionados!

Tabela 8-4. Elementos DateTimeFormatter populares e úteis

Personagem Descrição

Exemplo

a

manhã-tarde-do-dia

PM

d

dia do mês

10

E

dia da semana

Ter; Terça-feira; T

G

era

AC, CE

k

hora do dia (1-24)

24

K

hora da manhã (0-11)

0

eu

mês do ano

julho; Julho; J.

h

relógio-hora-da-manhã (13h-12h) 12

H

hora do dia (0-23)

0

eu

minuto da hora

30

M

mês do ano

7; 07

é

segundo do minuto

55

S

fração de segundo

033954

você

ano (sem época)

2004; 04

sim

ano da época

2004; 04

z

nome do fuso horário

Hora padrão do Pacífico; PST

Z

deslocamento de zona

+0000; -0800; -08:00

Para montar um formato abreviado comum nos EUA, por exemplo, você pode usar os caracteres M, d e y. Você constrói o formatador usando o método estático ofPattern().

Agora você pode usar (e reutilizar) o formatador com o método parse() de qualquer uma das classes de data ou hora:

jshell> importar java.time.format.DateTimeFormatter

jshell> DateTimeFormatter shortUS =

...> DateTimeFormatter.ofPattern("MM/dd/aa")

shortUS ==> Valor(MonthOfYe ...) ... (YearOfEra,2,2,2000-01-01)

jshell> LocalDate dia dos namorados = LocalDate.parse("14/02/23", shortUS

)

dia dos namorados ==> 14/02/2023

jshell> LocalDate piDay = LocalDate.parse("14/03/23", shortUS) piDia ==> 14/03/2023

E como mencionamos anteriormente, o formatador funciona nas duas direções. Basta usar o método format() do seu formatador para produzir uma representação em string da sua data ou hora:

jshell> LocalDate hoje = LocalDate.now()

hoje ==> 2023-12-14

jshell> shortUS.format(hoje)

$ 30 ==> "14/12/23"

jshell> shortUS.format(piDay)

$ 31 ==> "14/03/23"

É claro que os formatadores também funcionam para horários e datas!

jshell> DateTimeFormatter militar =

...> DateTimeFormatter.ofPattern("HHmm")

militar ==> Valor(HourOfDay,2)Value(MinuteOfHour,2)

```
jshell> LocalTime pôr do sol = LocalTime.parse("2020",
militar) pôr do sol ==> 20:20

jshell> DateTimeFormatter básico =

...> DateTimeFormatter.ofPattern("h:mm a")

básico ==> Valor(ClockHourOfAmPm)': ... ,SHORT)

jshell> basic.format(pôr do sol)

$ 42 ==> "20h20"

jshell> compromisso DateTimeFormatter =

DateTimeFormatter.ofPattern("h:mm a MM/dd/aa z")

compromisso ==>

Valor(ClockHourOfAmPm)':' ...

0-01-01)' 'ZonaText(CURTO)
```

Observe na parte ZonedDateTime a seguir que colocamos o identificador de fuso horário (o caractere z) no final – provavelmente não onde você esperava!

```
jshell> ZonedDateTime dentista =

ZonedDateTime.parse("10h30 01/11/23 EST",
compromisso)

dentista ==> 2023-11-01T10:30-
04:00[América/Nova_Iorque]

jshell> ZonedDateTime agoraEST =
ZonedDateTime.now()
```

agoraEST ==> 2023-12-14T09:55:58.493006-05:00[América/Nova_Iorque]

jshell> compromisso.format(nowEST)

$ 47 ==> "9h55 14/12/23 EST"

Queríamos ilustrar o poder desses formatos. Você pode criar um formato para acomodar uma ampla variedade de estilos de entrada ou saída. Dados legados e formulários web mal projetados vêm à mente como exemplos diretos onde o DateTimeFormatter pode ajudar.

Erros de análise

Mesmo com todo esse poder de análise ao seu alcance, às vezes as coisas dão errado.

Lamentavelmente, as exceções que você vê são muitas vezes vagas demais para serem úteis imediatamente. Considere a seguinte tentativa de analisar um horário com horas, minutos e segundos:

jshell> DateTimeFormatter withSeconds =

...> DateTimeFormatter.ofPattern("hh:mm:ss")

comSegundos ==>

Valor(ClockHourOfAmPm,2)':' ...

Valor(SecondOfMinute,2)

jshell> LocalTime.parse("03:14:15", comSegundos)

| Exceção java.time.format.DateTimeParseException:

| O texto '03:14:15' não pôde ser analisado: não foi possível obter

| LocalTime do TemporalAccessor: {MinuteOfHour=14, MilliOfSecond=0,

| SecondOfMinute=15, NanoOfSecond=0, HourOfAmPm=3,

| MicroOfSecond=0},ISO do tipo java.time.format.Parsed

| em DateTimeFormatter.createError (DateTimeFormatter.java:2020)

| em DateTimeFormatter.parse (DateTimeFormatter.java:1955)

| em LocalTime.parse (LocalTime.java:463)

| em (#33:1)

| Causado por: java.time.DateTimeException:

Não é possível obter LocalTime de...

| em LocalTime.from (LocalTime.java:431)

| em Parsed.query (Parsed.java:235)

| em DateTimeFormatter.parse (DateTimeFormatter.java:1951)

| ...

Caramba! Java lançará uma DateTimeParseException sempre que não puder analisar a entrada da string. Java também lançará a exceção em casos como o nosso exemplo acima; os campos foram analisados

corretamente a partir da string, mas não forneceram informações suficientes para criar um objeto LocalTime. Pode não ser óbvio, mas nosso horário, “3:14:15”, pode ser no meio da tarde ou bem cedo pela manhã. Nossa escolha do padrão hh para as horas acaba sendo a culpada. Podemos escolher um padrão horário que use uma escala inequívoca de 24 horas ou podemos adicionar um elemento AM/PM explícito:

```
jshell> DateTimeFormatter válido1 =

...> DateTimeFormatter.ofPattern("hh:mm:ss a")

válido1 ==> Valor(ClockHourOfAmPm,...y,SHORT)

jshell> DateTimeFormatter válido2 =

...> DateTimeFormatter.ofPattern("HH:mm:ss")

válido2 ==> Valor(HourOfDay,2)': ... Minuto,2)

jshell> Hora Local piDay1 =

...> LocalTime.parse("15:14:15", válido1)

piDia1 ==> 15:14:15

jshell> Hora Local piDay2 =

...> LocalTime.parse("03:14:15", válido2)

piDia2 ==> 03:14:15
```

Se você receber uma DateTimeParseException, mas sua entrada parecer uma correspondência correta para o formato, verifique novamente se o seu formato inclui tudo o que é necessário para criar sua data ou hora. Um pensamento final sobre essas exceções: talvez você

precise usar o caractere não mnemônico u para analisar anos se suas datas não incluírem naturalmente um designador de época como CE.

Existem muitos, muitos mais detalhes sobre DateTimeFormatter. Para isso, mais do que para a maioria das classes utilitárias, vale a pena uma visitaleia os documentos

on-line.

Formatando datas e horas

Agora que você sabe como criar, analisar e armazenar datas e horas, precisa exibir esses dados úteis. Felizmente, você pode construir strings agradáveis e legíveis por humanos usando o mesmo formatador que você construiu para analisar datas e horas a partir de strings. Lembra do nosso withSeconds e formatadores militares? Você pode escolher a hora atual e transformá-la rapidamente em qualquer formato, assim: jshell> DateTimeFormatter withSeconds =

...> DateTimeFormatter.ofPattern("hh:mm:ss")

withSeconds ==> Value(ClockHou ... OfMinute,2)

jshell> DateTimeFormatter militar =

...> DateTimeFormatter.ofPattern("HHmm")

militar ==> Valor(HourOfDay,2)Value(MinuteOfHour,2)

jshell> LocalTime t = LocalTime.now()

==> 09:17:34,356758

jshell> withSeconds.format(t)

$ 7 ==> "09:17:34"

jshell> militar.format(t)

$ 8 ==> "0917"

Você pode usar qualquer padrão de data ou hora criado a partir das partes mostradas emTabela 8-4para produzir esta saída formatada. Entre no jshell e tente criar alguns formatos. Você pode usar os métodos LocalTime.now() e LocalDate.now() para criar alguns alvos fáceis para seus testes de formatação.

Carimbos de data e hora

Um outro conceito popular de data e hora que java.time entende são os carimbos de data/hora. Em qualquer situação em que você precise acompanhar o fluxo de informações, será necessário um registro de quando exatamente as informações são produzidas ou modificadas. Você ainda verá a classe java.util.Date usada para armazenar esses momentos no tempo, mas a classe java.time.Instant carrega tudo que você precisa para um carimbo de data/hora e vem com todos os outros benefícios das outras classes no java.time pacote:

jshell> Instant time1 = Instant.now()

hora1 ==> 14/12/2019T15:38:29.033954Z

jshell> Hora instantânea2 = Instant.now()

hora2 ==> 14/12/2019T15:38:46.095633Z

jshell> time1.isAfter(time2)

$ 54 ==> falso

```
jshell> time1.plus(3, ChronoUnit.DAYS)

$ 55 ==> 17/12/2019T15:38:29.033954Z
```

Se datas ou horários aparecerem em seu trabalho, o pacote java.time será um ajudante bem-vindo. Você tem um conjunto maduro e bem projetado de ferramentas para lidar com esses dados – sem necessidade de bibliotecas de terceiros!

Outros utilitários úteis

Vimos alguns dos blocos de construção do Java, incluindo strings e números, bem como uma das combinações mais populares dessas strings e números – datas – nas classes LocalDate e LocalTime. Esperamos que esta gama de utilitários tenha lhe dado uma noção de como o Java funciona com muitos dos elementos que você

provavelmente encontrará.

Certifique-se de ler a documentação dos pacotes java.util, java.text e java.time para obter mais utilitários que podem ser úteis. Por exemplo, você pode usar java.util.Random para gerar as coordenadas aleatórias das árvores emFigura 8-1. Às vezes, o trabalho “utilitário” é realmente complexo e requer atenção cuidadosa aos detalhes. Pesquisar online exemplos de código ou até mesmo bibliotecas completas escritas por outros desenvolvedores pode acelerar seus próprios esforços.

A seguir, começaremos a desenvolver esses conceitos fundamentais. Java continua tão popular porque inclui suporte para técnicas mais avançadas além das básicas. Uma dessas técnicas são os recursos de “thread”, que estão integrados. Threads fornecem melhor acesso a sistemas modernos e poderosos, mantendo o

desempenho de seus aplicativos mesmo durante a execução de muitas tarefas complexas. Mostraremos como aproveitar esse recurso de assinatura emCapítulo 9.

Perguntas de revisão

1. Qual classe contém a constante π? Você precisa importar essa classe para usar π?

2. Qual pacote contém melhores substitutos para a classe java.util.Date original?

3. Qual classe você deve usar para formatar uma data para uma saída amigável?

4. Que símbolo você usaria em uma expressão regular para ajudar a combinar as palavras “sim” e “sim”?

5. Como você converteria a string “42” no número inteiro 42?

6. Como você compararia duas strings (como “yes” e “YES”) para ver se elas correspondem, ignorando qualquer capitalização?

7. Qual operador concatena strings?

Exercícios de código

Vamos revisitar nosso aplicativo gráfico Hello Java e atualizá-lo usando alguns novos utilitários e recursos de string discutidos neste capítulo. Você pode começar com a classe HelloChapter8 na pasta exercícios/ch08. Queremos que o programa suporte alguns argumentos de linha de comando para a mensagem e posição inicial.

Seu programa deve aceitar 0, 1 ou 2 argumentos:

•

Zero argumentos devem colocar o texto “Olá, utilitários!” no centro para começar.

•

Um argumento deve ser tratado como a mensagem a ser exibida; deve estar centralizado para começar:

–

Lembre-se de que as mensagens com várias palavras devem ser

colocadas entre aspas.

–

Se a mensagem for a palavra hoje, seu código deverá gerar uma data

formatada para usar como mensagem.

•

Dois argumentos representam a mensagem e as coordenadas iniciais de onde exibi-la:

–

As coordenadas devem ser uma string entre aspas contendo um par de

números separados por vírgula e espaço em branco opcional. Estas são

todas strings de coordenadas válidas:

-

150.150

•

50, 50

•

100, 220

–

O argumento coordenadas também pode ser a palavra random, o que

significa que seu código deve gerar uma posição inicial aleatória.

aqui estão alguns exemplos:

$ java OláCapítulo8

// "Olá, utilitários!" centralizado na janela

$ java HelloChapter8 "Funciona!"

// "Funciona!" centralizado na janela

$ java HelloChapter8 "Sinto-me encurralado" "20,20"

// "Sinto-me encurralado" no canto superior esquerdo

Se o usuário tentar passar três ou mais argumentos, seu código deverá gerar uma mensagem de erro e sair.

Comece testando o número de argumentos. Se o seu programa obtiver pelo menos um argumento, use o

primeiro argumento para a mensagem. Se forem dois, você precisará dividir as coordenadas e convertê-las em números. Se você obtiver o argumento aleatório, certifique-se de gerar números aleatórios que manterão a mensagem visível.

(Você pode assumir um comprimento padrão razoável para a mensagem; não há problema se parte de uma mensagem mais longa ficar truncada no lado direito.) Teste sua solução com algumas execuções. Experimente coordenadas diferentes.

Experimente a opção aleatória. Experimente a opção aleatória algumas vezes sucessivamente para ter certeza de que a posição inicial realmente muda. O que acontece se você digitar aleatoriamente aleatoriamente no segundo argumento?

Para uma atualização adicional: tente escrever uma expressão regular para aceitar algumas variações aleatórias enquanto ainda ignora o caso:

•

aleatório

•

Rand

•

rndm

•

R

Como sempre, você pode encontrar algumas dicas para esse problema emApêndice B.

Nossas soluções estão na pasta ch08/exercises do código-fonte.

1Na dúvida, meça! Se o seu código de manipulação de String for limpo e fácil de entender, não o reescreva até que alguém prove que ele é muito lento. As chances são de que eles estejam errados. E não se deixe enganar por comparações relativas. Um milissegundo é mil vezes mais lento que um microssegundo, mas ainda assim pode ser insignificante para o desempenho geral do seu aplicativo.

2Na maioria das plataformas a codificação padrão é UTF-8. Você pode obter mais detalhes sobre conjuntos de caracteres, conjuntos padrão e conjuntos padrão suportados por Java nodocumentação oficial do Javadocpara a classe java.nio.charset.Charset.

3Validar endereços de e-mail acaba sendo muito mais complicado do que podemos abordar aqui. Expressões regulares podem abranger a maioria dos endereços válidos, mas se você estiver fazendo validação para um aplicativo comercial ou outro aplicativo profissional, você pode querer investigar bibliotecas de terceiros, como aquelas disponíveis emApache Commons.

4Você é bem vindo parasolicite seu próprio TLD global personalizadose você tiver algumas centenas de milhares de dólares por aí.

5O tipo float é de “precisão simples” e double é, bem, precisão dupla. (Daí seu nome!) Um double pode reter números com aproximadamente o dobro da precisão de um float. A precisão arbitrária significa simplesmente que

você pode ter quantos dígitos precisar antes e depois da vírgula decimal. Para ser justo, a NASA usa um valor para π

com apenas 15 dígitos de precisão, que o dobro pode lidar bem.

Capítulo 9. Tópicos

Assumimos como certo que os sistemas de computador modernos podem gerenciar muitos aplicativos e tarefas do sistema operacional (SO) executados simultaneamente e fazer parecer que todo o software está sendo executado simultaneamente. A maioria dos sistemas atuais possui múltiplos processadores ou múltiplos núcleos, ou ambos, e podem atingir um grau impressionante de simultaneidade. O sistema operacional ainda faz malabarismos com aplicativos de nível superior, mas volta sua atenção de um para o outro tão rapidamente que eles também parecem ser executados ao mesmo tempo.

Observação

Na programação, a operação simultânea denota múltiplas tarefas, normalmente não relacionadas, executadas ao mesmo tempo. Pense em um cozinheiro de fast-food preparando vários pedidos em uma grelha. A operação paralela geralmente envolve dividir uma tarefa grande em subtarefas relacionadas que podem ser executadas lado a lado para produzir o resultado final mais rapidamente. Nosso cozinheiro poderia preparar um cheeseburger duplo com bacon “em paralelo”, jogando dois

hambúrgueres e um pouco de bacon na grelha ao mesmo tempo. Em ambos os casos, os programadores falam de

forma mais geral sobre essas tarefas e subtarefas que ocorrem simultaneamente. Isso não quer dizer que tudo começa e termina exatamente no mesmo instante, mas significa que os tempos de execução dessas tarefas se sobrepõem.

Antigamente, a unidade de simultaneidade de um sistema operacional era o aplicativo ou processo. Para o sistema operacional, um processo era mais ou menos uma caixa preta que decidia o que fazer por conta própria. Se um aplicativo exigisse maior simultaneidade, ele só poderia obtê-la executando vários processos e comunicando-se entre eles, mas essa era uma abordagem pesada e não muito elegante.

Mais tarde, os sistemas operacionais adicionaram o conceito de threads.

Conceitualmente, um thread é um fluxo de controle dentro de um programa. (Você pode ter ouvido falar de um “thread de execução”, por exemplo.) Threads fornecem simultaneidade refinada dentro de um processo sob o controle do próprio aplicativo.

Threads existem há muito tempo, mas historicamente têm sido difíceis de usar. Os utilitários de simultaneidade Java abordam padrões e práticas comuns em aplicativos multithread e os elevam ao nível de métodos e classes tangíveis. Coletivamente, isso significa que Java suporta threading em níveis superiores e inferiores.

Esse amplo suporte torna mais fácil para os programadores escreverem código multithread e para compiladores e tempos de execução otimizarem esse código. Isso também significa que as APIs Java aproveitam ao máximo o threading, por isso é importante

que você ganhe algum grau de familiaridade com esses conceitos no início de sua exploração de Java. Nem todos os desenvolvedores precisarão escrever aplicativos que usem explicitamente threads ou simultaneidade, mas a maioria usará algum recurso que os envolva.

Threads são essenciais para o design de muitas APIs Java, especialmente aquelas envolvidas em aplicativos, gráficos e som do lado do cliente. Por exemplo, quando olhamos para a programação GUI emCapítulo 12, você verá que o método paint() de um componente não é chamado diretamente pelo aplicativo, mas sim por um thread de desenho separado dentro do sistema de tempo de execução Java. A qualquer momento, muitos desses threads em segundo plano podem estar realizando atividades junto com seu aplicativo, mas você ainda recebe atualizações oportunas em sua tela. No lado do servidor, os threads Java também estão presentes, atendendo todas as solicitações e executando seu aplicativo. É importante entender como seu código se encaixa nesse ambiente.

Neste capítulo, falaremos sobre como escrever aplicativos que criam e usam seus próprios threads explicitamente. Falaremos primeiro sobre o suporte de thread de baixo nível integrado à linguagem Java e depois discutiremos o pacote de utilitários de thread java.util.concurrent. Também abordaremos os novos threads virtuais visualizados no Java 19 sob o nome de Project Loom.

Apresentando Tópicos

Um thread é semelhante à noção de um processo ou programa em execução, exceto que diferentes threads dentro do mesmo aplicativo estão muito mais

relacionados e compartilham muito do mesmo estado do que diferentes programas executados na mesma máquina. É como um campo de golfe que muitos golfistas usam ao mesmo tempo. Os threads cooperam para compartilhar uma área de trabalho. Eles se revezam e esperam por outros tópicos. Eles têm acesso aos mesmos objetos, incluindo variáveis estáticas e de instância, em seu aplicativo. No entanto, os threads têm suas próprias cópias de variáveis locais, assim como os jogadores compartilham o campo de golfe ou o carrinho de golfe, mas não compartilham tacos ou bolas.

Vários threads em um aplicativo têm os mesmos problemas que os jogadores de golfe em um campo – em uma palavra, sincronização. Assim como você não pode ter dois conjuntos de jogadores jogando o mesmo green ao mesmo tempo, você não pode ter vários threads tentando acessar as mesmas variáveis sem algum tipo de coordenação.

Caso contrário, alguém poderá se machucar. Um thread pode reservar o direito de usar um objeto até terminar sua tarefa, assim como um grupo de golfe obtém direitos exclusivos sobre o green até que cada um dos jogadores desse grupo termine. E um tópico que é mais importante pode aumentar a sua prioridade, afirmando o seu direito de “prosseguir”.

O diabo está nos detalhes, é claro, e esses detalhes há muito dificultam o uso dos fios.

Felizmente, Java simplifica a criação, o controle e a coordenação de threads integrando alguns desses conceitos diretamente na linguagem.

É comum tropeçar em tópicos quando você trabalha com eles pela primeira vez. A criação de um thread exercitará muitas de suas novas habilidades em Java de uma só vez. Apenas lembre-se de que dois jogadores estão sempre envolvidos na execução de um thread: um objeto Java Thread que representa o próprio thread e um objeto de destino arbitrário que contém o método que o thread executará. Mais tarde, veremos maneiras de combinar essas duas funções, mas essas abordagens apenas mudam a embalagem, não o relacionamento.

A classe Thread e a interface executável

Toda execução em Java está associada a um objeto Thread, começando com um thread

“principal” que a JVM inicia para lançar sua aplicação. Um novo thread nasce quando você cria uma instância da classe java.lang.Thread. O objeto Thread representa um thread real no interpretador Java e serve como um identificador para controlar e coordenar sua execução. Com ele, você pode iniciar o thread, esperar que ele termine, fazer com que ele hiberne por um tempo ou interrompa sua atividade.

O construtor da classe Thread aceita informações sobre onde o thread deve iniciar sua execução. Gostaríamos de dizer qual método executar. Existem várias maneiras de fazer isso. A abordagem clássica usa a interface java.lang.Runnable para marcar um objeto que contém um método “executável”.

Executáveldefine um método run() único e de uso geral: interface pública executável {

abstrato público void run();

```
}
```

Cada thread começa sua vida executando o método run() em um objeto Runnable, que é o “objeto alvo” passado para o construtor do thread. O método run() pode conter qualquer código, mas deve ser público, não receber argumentos, não ter valor de retorno e não gerar exceções verificadas.

Qualquer classe que contenha um método run() apropriado pode declarar que implementa a interface Runnable. Uma instância desta classe se torna um objeto executável que pode servir como alvo de um novo thread. Se você não quiser colocar o método run() diretamente em seu objeto (e muitas vezes não o faz), você sempre pode criar uma classe adaptadora que sirva como Runnable para você. O método run() do adaptador pode então chamar qualquer método desejado após o thread ser iniciado.

Mostraremos exemplos dessas opções posteriormente.

Criando e iniciando threads

Um thread recém-nascido permanece ocioso até que lhe demos um tapa figurativo na parte inferior, chamando seu método start(). O thread então acorda e executa o método run() de seu objeto de destino. start() pode ser chamado apenas uma vez durante a vida de um thread. Depois que um thread é iniciado, ele continua em execução até que o método run() do objeto de destino retorne ou gere uma exceção não verificada.

A classe a seguir, Animator, implementa um método run() para conduzir um loop de desenho. Poderíamos usar algo semelhante em nosso jogo para atualizar o campo de jogo:

class Animator implementa Runnable {

animação booleana = true;

execução nula pública() {

enquanto (animando) {

// move as maçãs ativas um "quadro"

//repintar o campo

// pausa

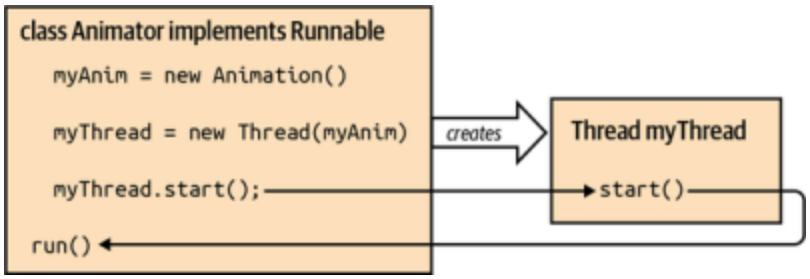
// ...

}

}

}

Para usá-lo, crie um objeto Thread, passe para ele uma instância do Animator como seu objeto alvo e invoque seu método start():



Animador meuAnimator = new Animator();

Tópico meuThread = new Tópico(meuAnimator);

meuThread.start();

Criamos uma instância de nossa classe Animator e a passamos como argumento para o construtor de myThread. Como mostrado em Figura 9-1, quando chamamos o método start(), myThread começa a executar o método run() do Animator.

Figura 9-1. Animator como uma implementação do Runnable

Deixe o show começar!

Um fio nato

A interface Runnable permite transformar um objeto arbitrário no alvo de um thread, como no exemplo anterior. Este é o uso geral mais importante da classe Thread. Na maioria das situações em que você precisa usar threads, você criará uma classe (possivelmente uma classe adaptadora simples) que implementa a interface Runnable.

Outra opção de design para criar um thread torna nossa classe alvo uma subclasse de um tipo que já pode ser executado. Acontece que a própria classe Thread implementa convenientemente a interface Runnable; ele tem seu próprio método run(), que podemos substituir diretamente para cumprir nossa licitação:

```
class Animator estende Thread {

animação booleana = true;

execução nula pública() {

enquanto (animando) {

//desenha quadros
```

```
//faço outras coisas...

}

}

}
```

O esqueleto da nossa classe Animator parece o mesmo de antes, exceto que nossa classe agora é uma subclasse de Thread. Para seguir esse esquema, o construtor padrão da classe Thread torna-se o destino padrão - ou seja, por padrão, o Thread executa seu próprio método run() quando chamamos o método start(), conforme mostrado emFigura 9-2. Agora nossa subclasse pode simplesmente substituir o método run() na classe Thread. (O próprio thread define um método run() vazio.)

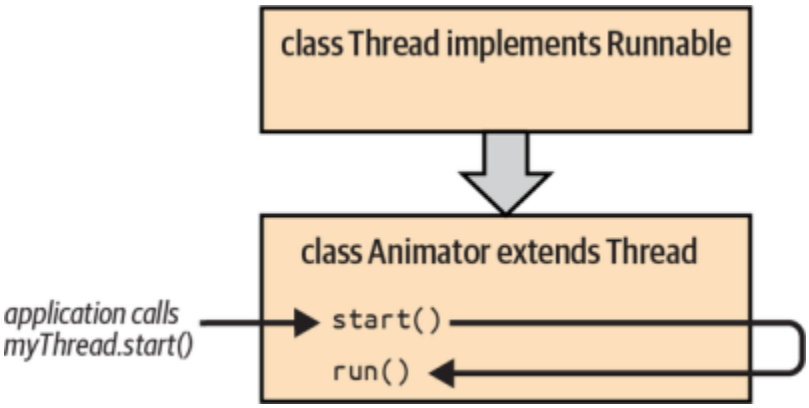


Figura 9-2. Animator como uma subclasse de Thread

A seguir, criamos uma instância do Animator e chamamos seu método start() (que também herdou de Thread):

```
Animador saltitante = new Animator();

saltitante.start();
```

Estender Thread pode parecer uma maneira conveniente de agrupar um thread e seu método run() de destino. No entanto, esta abordagem muitas vezes não é o melhor design. Se você estender Thread para implementar um thread, estará dizendo que precisa de um novo tipo de objeto que seja um tipo de Thread, que exponha todos os métodos públicos da classe Thread. Embora haja algo de satisfatório em pegar um objeto que se preocupa principalmente com a execução de uma tarefa e torná-lo um Thread, as situações reais em que você deseja criar uma subclasse de Thread não devem ser muito comuns. Na maioria dos casos, é mais natural deixar os requisitos do seu programa ditarem a estrutura da classe e usar Runnables para conectar a execução e a lógica do seu programa.

Controlando Threads

Agora que você viu o método start() usado para iniciar a execução de um novo thread, vamos dar uma olhada nos métodos de instância que permitem controlar

explicitamente o comportamento de um thread em tempo de execução:

Thread.sleep()método

Faz com que o thread em execução no momento espere por um período de tempo designado (mais ou menos), sem consumir muito (ou possivelmente nenhum) tempo de CPU.

espere()e métodos join()

Coordene a execução de dois ou mais threads. Discutiremos isso em detalhes quando falarmos sobre sincronização de threads posteriormente neste capítulo.

interromper()método

Ativa um thread que está suspenso em uma operação sleep() ou wait() ou que está bloqueado em uma operação de E/S longa.1

Métodos obsoletos

Devemos também mencionar três métodos de controle de thread obsoletos: stop(), suspend() e resume(). O método stop() complementa start(); isso destrói o fio. start() e o obsoleto método stop() podem ser chamados apenas uma vez no ciclo de vida do

thread. Por outro lado, os métodos obsoletos suspend() e resume() pausam arbitrariamente e depois reiniciam a execução de um thread.

Embora esses métodos obsoletos ainda existam na versão mais recente do Java (e provavelmente estarão lá para sempre), eles não devem ser usados no

desenvolvimento de novos códigos. O problema com stop() e suspend() é que eles assumem o controle da execução de um thread de maneira descoordenada e severa.

Você pode criar e monitorar algumas variáveis como a melhor maneira de afetar a execução de um thread (se essas variáveis forem booleanas, você poderá vê-las chamadas de “sinalizadores”). Os primeiros exemplos de threads neste livro usam essa técnica de uma forma ou de outra. Exemplos posteriores apresentarão alguns dos outros recursos de controle disponíveis através das classes de simultaneidade.

O método sleep()

Os programadores geralmente precisam dizer a um thread para ficar ocioso ou

“hibernar” por algum período de tempo. Pode ser necessário esperar que algum recurso externo fique disponível, por exemplo. Até mesmo nossos threads de animação simples fazem pequenas pausas entre os quadros. Enquanto um thread está adormecido ou bloqueado de algum tipo de entrada, ele não consome tempo de CPU

nem compete com outros threads para processamento. Para tais pausas, podemos chamar o método estático Thread.sleep(), que afeta o thread em execução no momento. A chamada faz com que o thread fique inativo por um número especificado de milissegundos:

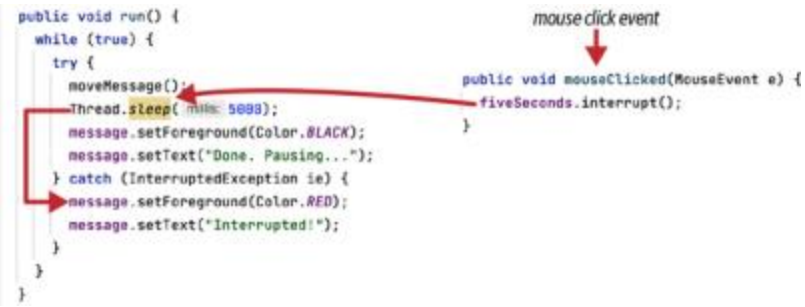
```
tentar {

//O tópico atual

Thread.sleep(1000);

} catch (InterruptedException e) {

// alguém nos acordou prematuramente

}
```

O método sleep() pode lançar uma InterruptedException se for interrompido por outro thread através do método interrupt() (mais abaixo). Como você viu no código anterior, o thread pode capturar essa exceção e aproveitar a oportunidade para executar alguma ação – como verificar uma variável para determinar se o thread deve ou não sair – e então voltar a dormir.

Os métodos join(), wait() e notify()

Se você precisar coordenar as atividades de um thread aguardando que outro thread conclua sua tarefa, você pode usar o método join(). Chamar o método join() de um thread faz com que esse thread seja bloqueado até que o thread de destino seja concluído. Alternativamente, você pode chamar join() com um número de milissegundos de espera como argumento. Nesse formato, o thread de chamada espera até que o thread de destino seja concluído ou o período especificado decorra.

Esta é uma forma muito grosseira de sincronização de threads.



Se precisar coordenar as atividades de um thread com algum outro recurso, como verificar o estado de um arquivo ou conexão de rede, você pode usar os métodos wait() e notify(). Chamar wait() em um thread irá pausá-lo, semelhante a usar join(), mas ele permanecerá pausado até que seja interrompido() por algum outro thread, ou até que você mesmo chame notify() no thread.

Java suporta mecanismos mais gerais e poderosos para coordenar a atividade de thread no pacote java.util.concurrent. Mostraremos mais deste pacote posteriormente neste capítulo.

O método interrupção()

O método interrupt() faz mais ou menos o que diz na lata. Ele interrompe o fluxo normal de execução de um thread. Se esse thread estiver ocioso em uma operação sleep(), wait() ou operação de E/S demorada, ele será ativado. Quando você interrompe um thread, seu sinalizador de status de interrupção é definido. Você pode testar esse sinalizador com o método isInterrupted(). Você também pode usar um formato alternativo, isInterrupted(boolean), para indicar se deseja ou não que o thread limpe seu status de interrupção após recuperar o valor atual.

Embora você provavelmente não use o interrupt() com tanta frequência, ele definitivamente pode ser útil. Se você já ficou impaciente enquanto um aplicativo de desktop tenta — e falha — se conectar a um servidor ou banco de dados, você já passou por um daqueles momentos em que uma interrupção pode ser a coisa certa.

Vamos simular este cenário com uma pequena aplicação gráfica. Mostraremos um rótulo na tela e o moveremos para um local novo e aleatório a cada cinco segundos.

Durante essa pausa de cinco segundos, um clique em qualquer lugar da tela interromperá a pausa. Mudaremos a mensagem e então iniciaremos o ciclo de movimentos aleatórios novamente. Você pode executar o exemplo completo em ch09/examples/Interruption.java, masFigura 9-3destaca o fluxo e o efeito da chamada de interrupção().

Figura 9-3. Interrompendo um tópico

Revisitando a animação com threads

Gerenciar animações é uma tarefa comum em interfaces gráficas. Às vezes as animações são transições sutis; outras vezes, eles são o foco do aplicativo, como acontece com nosso jogo de lançar maçãs. Veremos duas maneiras de lidar com a animação: usando threads simples junto com as funções sleep() e usando um timer.

Combinar uma dessas opções com algum tipo de função de etapa ou “próximo quadro”

é uma abordagem popular e também fácil de entender.

Você pode usar um tópico semelhante a“Criando e iniciando tópicos”para produzir animação real. A ideia básica é pintar ou posicionar todos os seus objetos animados, pausar, movê-los para os próximos pontos e depois repetir. Vamos primeiro dar uma olhada em como desenhamos algumas peças do nosso campo de jogo sem animação.

Incluiremos uma nova lista para quaisquer maçãs ativas, além das listas existentes para árvores e sebes. Você pode obter este código em seu editor na pasta ch09/examples/game:

```
// Da classe Campo...

protegido void paintComponent(Gráficos g) {

g.setColor(fieldColor);

g.fillRect(0,0, getWidth(), getHeight());

físico.draw(g);

for (Árvore t: árvores) {t.draw(g); }
```

```
for (Hedge h: hedges) { h.draw(g); }

físico.draw(g);

para (Apple a: maçãs) {a.draw(g); }

}

// E da aula da Apple...

public void draw(Gráficos g) {

// Certifique-se de que nossa maçã ficará vermelha e
depois pinte-a!

g.setColor(Color.RED);

g.fillOval(x, y, dimensionadoLength,
dimensionadoLength);

}
```

Começamos pintando o campo de fundo, depois as árvores e sebes, depois o nosso físico e, finalmente, quaisquer maçãs. Pintar as maçãs por último garante que elas ficarão “em cima” dos demais elementos.

O que muda na tela enquanto você joga? Na verdade, existem apenas dois itens

“móveis”: a maçã que nosso físico está mirando e quaisquer maçãs que voem ativamente após serem lançadas. O físico aponta em resposta à entrada do usuário (movendo o mouse ou clicando em um botão). Isso não requer animação separada, então adicionaremos essa funcionalidade emCapítulo 12. Por enquanto, podemos nos concentrar no manuseio de maçãs voadoras.

A etapa de animação do nosso jogo deve mover todas as maçãs ativas, de acordo com as regras da gravidade. Primeiro, adicionamos um método toss() à nossa classe Apple, onde podemos definir as condições iniciais para a nossa maçã usando informações do nosso físico. (Como o físico ainda não é interativo, falsificaremos alguns dados.) Em seguida, fazemos um movimento para uma determinada maçã no método step():

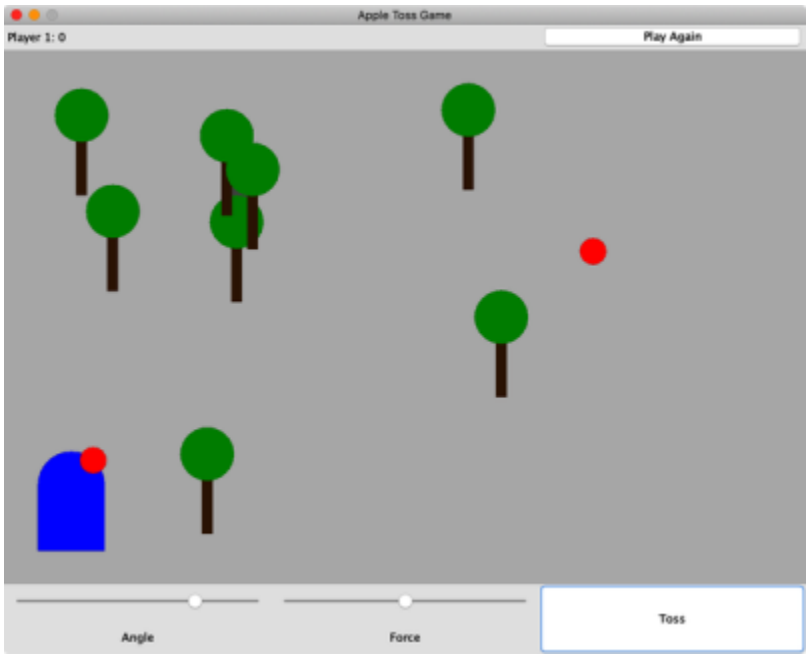
```
// Arquivo: ch09/examples/game/Apple.java

// Parâmetros fornecidos pelo físico

lance de vazio público (ângulo de flutuação, velocidade de flutuação) {

lastStep = System.currentTimeMillis();

radianos duplos = ângulo / 180 * Math.PI;

velocidadeX = (flutuação)(velocidade * Math.cos(radianos) / massa);

// Começa negativo já que "up" significa valores menores de y velocidadeY = (float)(-velocity * Math.sin(radianos) / massa);

}

passo vazio público() {

// Certifique-se de que a maçã ainda esteja se movendo

// usando nosso rastreador lastStep como sentinela

if (última etapa > 0) {

//Aplica a lei da gravidade à posição vertical da maçã
```

```
longo agora = System.currentTimeMillis();

float slice = (agora - lastStep) / 1000.0f;

velocidadeY = velocidadeY + (fatia *
Campo.GRAVIDADE);

int novoX = (int)(centroX + velocidadeX);

int novoY = (int)(centroY + velocidadeY);

setPosition(novoX, novoY);

}

}
```

Começamos calculando a velocidade com que a maçã se moverá (as variáveis speedX e speedY) no método toss(). Em nosso método step(), atualizamos a posição da maçã com base nessas duas velocidades e, em seguida, ajustamos a velocidade vertical com base na força da nossa gravidade. Não é muito sofisticado, mas produzirá um belo arco para as maçãs. Em seguida, colocamos esse código em um loop que fará os cálculos de atualização, redesenhará o campo e as maçãs, pausará e repetirá:

```
// Arquivo: ch09/examples/game/Field.java

//duração de um quadro de animação em milissegundos

público estático final int STEP = 40;

// ...

// Uma classe interna simples com nosso método run()
```

```
class Animator implementa Runnable {

execução nula pública() {

// "animação" é uma variável global que nos permite

// para parar de animar e conservar recursos

// se não houver maçãs ativas para mover

enquanto (animando) {

System.out.println("Passo " + apples.size() +

"maçãs");

para (maçã a: maçãs) {

um passo();

}

// Volte para nossa instância de classe externa para
repintar

Field.this.repaint();

// E livre-se de todas as maçãs que estiverem no chão

cullFallenApples();

tentar {

Thread.sleep(STEP);
```



} catch (InterruptedException, ou seja) {

System.err.println("Animação interrompida");

animação = falso;

}

}

}

}

Usaremos esta implementação de Runnable em um thread simples. Nossa classe Field manterá uma instância do thread e contém o seguinte método de início simples:

// Arquivo: ch09/examples/game/Field.java

Animação de threadThread;

// outros estados e métodos ...

```
void startAnimation() {

animaçãoThread = new Thread(new Animator());

animaçãoThread.start();

}
```

Discutiremos eventos em“Eventos”; você usará esses eventos para lançar uma maçã sob comando. Por enquanto, lançaremos apenas uma maçã automaticamente, conforme mostrado na Figura 9-4.

Figura 9-4. Maçãs jogáveis em ação

Não parece muito uma captura de tela estática, mas é incrível pessoalmente. ;^) **Morte de um fio**

Todas as coisas boas chegam ao fim. Um thread continua a ser executado até que uma das três coisas a seguir aconteça:

- Ele retorna explicitamente de seu método run() de destino.
- Ele encontra uma exceção de tempo de execução não detectada.
- O desagradável e obsoleto método stop() é chamado.

O que acontece se nada disso ocorrer e o método run() de um thread nunca terminar?

O thread pode continuar vivo, mesmo após a conclusão do código que o criou. Você deve estar ciente de como os threads eventualmente terminam, ou seu aplicativo pode acabar deixando threads órfãos em execução que consomem recursos

desnecessariamente ou até mesmo manter o aplicativo ativo quando, de outra forma, seria encerrado.

Em muitos casos, você deseja threads em segundo plano que executem tarefas simples e periódicas em um aplicativo. Você pode criar um desses trabalhadores em segundo plano usando o método setDaemon() para marcar um thread como um thread daemon. Threads daemon podem terminar como outros threads, mas se o aplicativo que os iniciou estiver encerrando, eles deverão ser eliminados e descartados quando nenhum outro thread de aplicativo não-daemon permanecer.2Normalmente, o interpretador Java continua em execução até que todos os threads sejam concluídos.

Mas quando os threads daemon são os únicos ainda ativos, o interpretador será encerrado.

Aqui está um esboço “diabólico” usando threads daemon:

classe Diabo estende Tópico {

Diabo() {

setDaemon(verdadeiro);

começar();

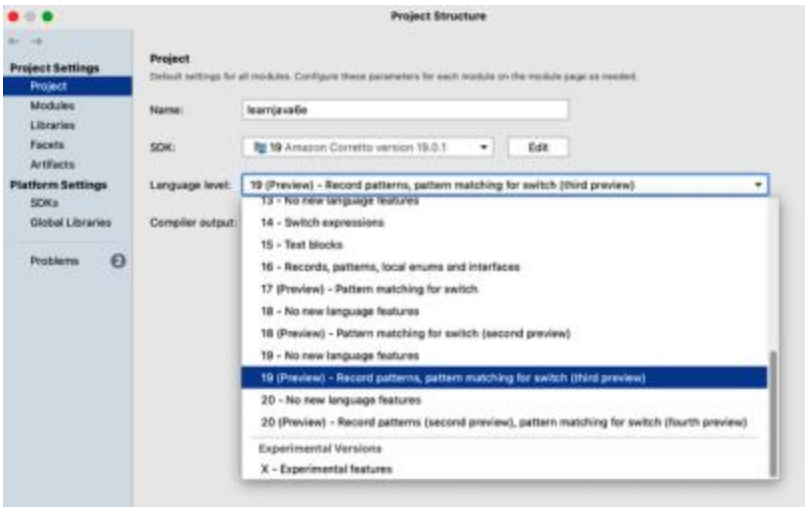
}

```
execução nula pública() {

//realiza algumas tarefas

}

}
```

Neste exemplo, o thread Devil define seu status de daemon quando é criado. Se algum thread do Devil permanecer quando nosso aplicativo for concluído, o sistema de tempo de execução o encerrará para nós. Não precisamos nos preocupar em limpá-los.

Uma nota final sobre como eliminar threads com elegância. Novos desenvolvedores geralmente encontram um problema comum na primeira vez que criam um aplicativo usando um componente gráfico Swing: seu aplicativo nunca sai. O Java VM parece travar indefinidamente depois que tudo é concluído e a janela do aplicativo é fechada.

Java cria um thread de UI para processar eventos de entrada e pintura. Este thread de interface do usuário não é um thread daemon, portanto, ele não sai automaticamente quando outros threads do aplicativo são concluídos. O desenvolvedor deve chamar System.exit() explicitamente. Se você pensar bem, isso faz sentido. Como a maioria dos aplicativos GUI são orientados a eventos e aguardam a entrada do usuário, eles seriam encerrados após a conclusão do código de inicialização.



Tópicos Virtuais

Pré-visualizado em Java 19 e finalizado em Java 21, Project Loom3traz threads virtuais leves para Java. Um dos principais objetivos do Projeto Loom é melhorar o ecossistema de threads em Java para que os desenvolvedores possam investir menos energia para manter aplicativos multithread estáveis e mais energia para resolver problemas de nível superior.

Visualizar tangente do recurso

O que queremos dizer com “visualizado em Java 19”? A partir do Java 12, a Oracle começou a introduzir alguns recursos da linguagem como visualizações. Esses recursos de visualização são bem especificados e totalmente implementados, mas não totalmente preparados. A Oracle ainda poderá fazer modificações substanciais em versões futuras. Eventualmente, esses recursos se tornarão partes permanentes do JDK ou serão removidos. A Oracle produz uma página de atualização de linguagem para cada nova versão do Java que contém um bom histórico de mudanças recentes na linguagem, bem como umvisão geral dos recursos de visualizaçãoem geral.

Como qualquer recurso de visualização pode acabar sendo descartado do Java, a Oracle exige que você inclua sinalizadores especiais ao compilar ou executar um aplicativo que o utiliza. Este requisito é uma pequena proteção para garantir que você não use acidentalmente código que pode não funcionar em uma versão futura do Java.

Configurando IDEs para recursos de visualização

Se você usar um IDE para demonstrações e exercícios, talvez seja necessário configurá-lo para suportar recursos de visualização. O IntelliJ IDEA, por exemplo, não oferece suporte a recursos de visualização por padrão. Você precisa alterar uma configuração na caixa de diálogo Arquivo → Estrutura do Projeto, conforme mostrado

emFigura 9-5.

Figura 9-5. Habilitando os recursos de visualização do Java no IntelliJ IDEA Depois de escolher a versão do Java desejada no menu suspenso do SDK, você pode ativar o suporte ao recurso de visualização escolhendo a opção apropriada no menu suspenso Nível de idioma. (Os recursos que o IDEA lista ao lado dos números de versão não são uma lista exaustiva.) Clique em OK após definir o nível de linguagem e o IDEA deverá estar pronto para compilar e executar qualquer código com recursos de visualização.

Renomeando arquivos de origem de visualização

A classe VirtualDemo (ch09/examples/VirtualDemo.java.preview) usa um thread virtual para fazer uma breve pausa antes de emitir nossa saudação favorita “Hello Java”. Antes de compilá-lo ou executá-lo, você precisará renomeá-lo.

Adicionamos o sufixo .preview a qualquer arquivo que inclua um recurso de visualização no código. O

sufixo impede que IDEs como o IntelliJ IDEA os compilem proativamente até que você tenha a chance de configurar o suporte à visualização, como mencionamos na seção anterior.

Você pode usar o menu de contexto (clique com o botão direito) no IntelliJ IDEA para renomear o arquivo no item de menu Refatorar. Você também pode renomear um arquivo rapidamente de um terminal no Linux ou macOS usando o comando mv:

```
% cd ch09/exemplos

% mv VirtualDemo.java.preview VirtualDemo.java

% cd ../..
```

Em um terminal ou prompt de comando do Windows, você pode usar o comando renomear:

```
C:\> cd ch09\exemplos

C:\> renomear VirtualDemo.java.preview
VirtualDemo.java

C:\>cd..\..
```

Compilando classes com recursos de visualização

A Oracle adicionou um par de opções de linha de comando para compilar código com recursos de visualização. Se você tentar compilar nosso arquivo fonte VirtualDemo com Java 19, por exemplo, provavelmente verá um erro semelhante a este:

% javac --versão

javac 19.0.1

% javac VirtualDemo.java

VirtualDemo.java:4: erro: startVirtualThread (Runnable)

é uma API de visualização e está desabilitada por padrão.

Thread thread = Thread.startVirtualThread(executável);

^

(use --enable-preview para ativar APIs de visualização)

O erro nos dá uma dica de como devemos proceder. Vamos tentar adicionar o sinalizador sugerido e compilar novamente:

% javac --enable-preview VirtualDemo.java

erro: --enable-preview deve ser usado com -source ou --release

Ratos! Outro erro diferente. Pelo menos também inclui algumas dicas. Para compilar, você precisa fornecer dois sinalizadores: --enable-preview e então -source ou --

release.4O compilador usa -source para especificar quais regras de linguagem se aplicam ao código-fonte que está sendo compilado. (O bytecode compilado ainda é

direcionado à mesma versão do Java que seu JDK.) Você pode usar a opção --release para especificar a versão de origem e a versão do bytecode.

Embora existam muitos cenários em que pode ser necessário compilar para sistemas mais antigos, os recursos de visualização devem ser usados com a versão atual do JDK.

Dessa forma, quando usarmos qualquer recurso de visualização do livro, combinaremos --enable-preview com --release e simplesmente forneceremos a mesma versão de lançamento da nossa versão do Java. Voltando ao nosso recurso de visualização de thread virtual, por exemplo, podemos usar Java 19 para testá-lo. Nossa chamada javac final e correta é assim:

% javac --versão

javac 19.0.1

% javac --enable-preview --release 19 VirtualDemo.java

Nota: VirtualDemo.java usa recursos de visualização do Java SE 19.

Nota: Recompile com -Xlint:preview para obter detalhes.

As “notas” que aparecem após a conclusão da compilação são puramente informativas.

Eles lembram que seu código depende de um recurso instável que pode não estar disponível no futuro. A nota não pretende dissuadi-lo de usar esses recursos, mas se estiver planejando compartilhar seu código com outros usuários ou desenvolvedores, você terá algumas restrições extras de compatibilidade a serem lembradas.

Se estiver curioso, você pode usar a opção -Xlint:preview mencionada nas notas para ver exatamente qual código de visualização causou o aviso:

src$ javac --enable-preview --release 19 -Xlint:preview VirtualDemo.java VirtualDemo.java:4: aviso: [visualização] startVirtualThread (Runnable) é uma API de visualização e pode ser removida em uma versão futura.

Thread thread = Thread.startVirtualThread(executável);

^

1 aviso

Não há surpresas nisso, mas, novamente, este é apenas um pequeno programa de demonstração. Com programas maiores ou códigos desenvolvidos em equipes, esse sinalizador -Xlint:preview extra pode ser muito útil.

Executando arquivos de classe de visualização

A execução de classes Java que incluem recursos de visualização também requer o sinalizador --enable-preview. Se você tentar executar o VirtualDemo com Java 19

como faria com qualquer outra classe, receberá um erro como este:

% java VirtualDemonstração

Erro: LinkageError ocorreu ao carregar a classe principal VirtualDemo java.lang.UnsupportedClassVersionError: Os recursos de visualização não s ão

habilitado para VirtualDemo (arquivo de classe versão 63.65535).

Tente executar com '--enable-preview'

Novamente, você pode usar o sinalizador mencionado no erro, --enable-preview, e pronto:

% java --enable-preview Demonstração Virtual

Olá tópico virtual! ID: 20

Se quiser brincar com um recurso de visualização no jshell, você também pode fornecer o mesmo sinalizador --enable-preview:

% jshell --enable-preview

A inclusão desse sinalizador permitiria que uma sessão jshell do Java 19 usasse threads virtuais, assim como nos permitiu executar nosso programa de demonstração acima.

Uma comparação rápida

A equipe do Loom projetou seus threads virtuais para serem fáceis de usar se você já tiver alguma habilidade com threads Java. Vamos retrabalhar o exemplo de thread trivial que usamos para testar o sinalizador --enable-preview para mostrar os dois tipos de thread:

classe pública VirtualDemo2 {

public static void main(String args[]) lança exceção {

Executável executável = new Runnable() {

execução nula pública() {

Tópico t = Thread.currentThread();

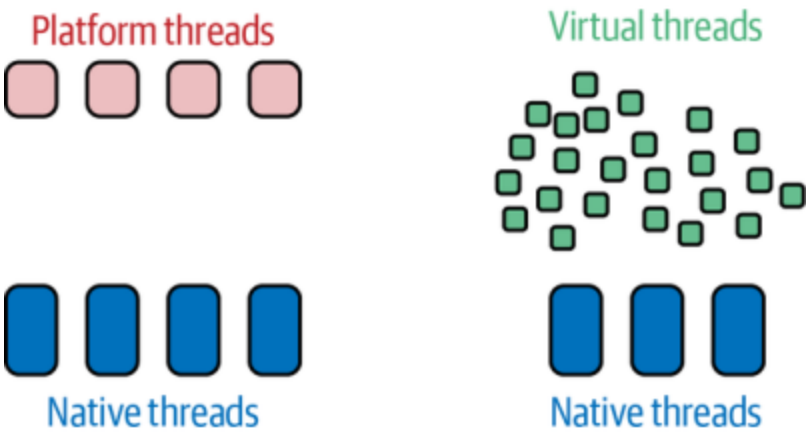
System.out.println("Olá tópico! " +

```
(t.isVirtual() ? "virtual " : "plataforma") +

"ID: " + t.threadId());

}

};

Thread thread1 = novo Thread (executável);

thread1.start();

Thread thread2 = Thread.startVirtualThread(executável);

thread1.join();

thread2.join();

}

}
```

Nessa reformulação, expandimos nossa classe interna anônima Runnable para fazer uma pequena investigação no thread atual. Imprimimos o número de identificação do thread e se é ou não um thread virtual. Mas veja como são parecidas (e simples) as linhas que lançam as duas threads: ambas aceitam nosso objeto executável e

“encaixam” na classe Thread. Para desenvolvedores com código estabelecido, mudar para usar esses threads virtuais deve ser simples. Aqui está o resultado após compilar e executar (com os sinalizadores de visualização apropriados, é claro): $ javac --enable-preview --source 19 VirtualDemo2.java

Nota: VirtualDemo.java usa recursos de visualização do Java SE 19.



Nota: Recompile com -Xlint:preview para obter detalhes.

$ java --enable-preview VirtualDemo2

Olá tópico! ID virtual: 21

Olá tópico! ID da plataforma: 20

Ambos os threads são executados conforme o esperado. De fato, um thread se reporta como um thread virtual. Usamos a palavra plataforma para descrever o outro thread, já que é assim que a documentação do Oracle os chama. Threads de plataforma representam um relacionamento direto e individual com os threads nativos que seu sistema operacional (a plataforma) fornece. Os threads virtuais, por outro lado, têm um relacionamento indireto muitos-para-um com os threads nativos do sistema operacional, conforme mostrado emFigura 9-6.

Figura 9-6. Threads de plataforma e virtuais são mapeados de maneira diferente para threads nativos

Essa separação é um dos principais recursos de design dos threads virtuais. Ele permite que Java tenha muitos (muitos!) threads em execução ao mesmo tempo, sem os custos de desempenho de criação e gerenciamento de

threads nativos correspondentes. Threads virtuais são projetados para serem baratos de criar e de alto desempenho quando estiverem instalados e em execução.

Sincronização

Cada tópico tem uma mente própria. Normalmente, um thread realiza seus negócios sem se importar com o que os outros threads do aplicativo estão fazendo. Os threads podem ser divididos em intervalos de tempo, o que significa que podem ser executados em surtos e rajadas arbitrárias, conforme direcionado pelo sistema operacional. Em um sistema multiprocessador ou multicore, é ainda possível que muitos threads diferentes sejam executados simultaneamente em CPUs diferentes.

Esta seção trata da coordenação das atividades de dois ou mais threads para que possam trabalhar juntos e usar as mesmas variáveis e métodos (sem colidir, como jogadores no campo de golfe).

Java fornece algumas estruturas simples para sincronizar as atividades dos threads.

Todos são baseados no conceito de monitores, um esquema de sincronização amplamente utilizado. Você não precisa saber os detalhes sobre como os monitores funcionam para poder usá-los, mas pode ser útil terFigura 9-7em mente.

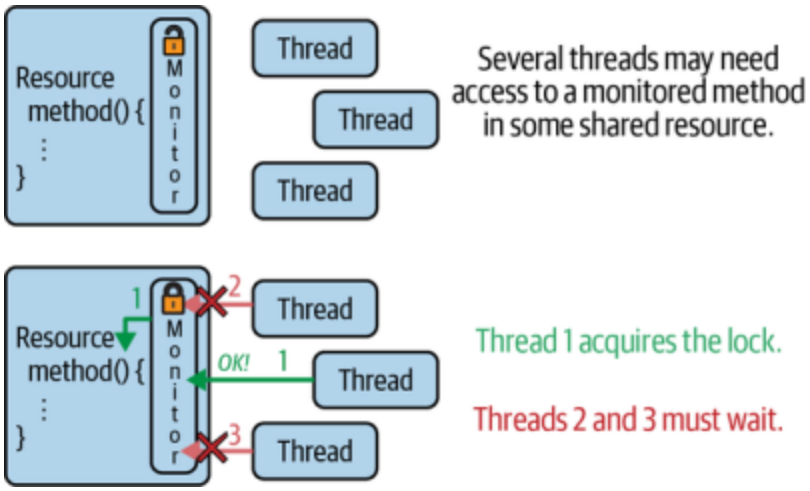


Figura 9-7. Sincronizando o acesso com um monitor

Um monitor é essencialmente uma fechadura. O bloqueio é anexado a um recurso que muitos threads podem precisar acessar, mas que deve ser acessado por apenas um thread por vez. É muito parecido com um banheiro com fechadura na porta; se estiver destrancado, você pode entrar e trancar a porta enquanto a usa. Se o recurso não estiver sendo utilizado, um thread poderá adquirir o bloqueio e acessar o recurso.

Quando o fio termina, ele abre mão da fechadura, assim como você destranca a porta do banheiro e a deixa aberta para a próxima pessoa (ou fio).

Entretanto, se outro thread já tiver o bloqueio para o recurso, todos os outros threads deverão esperar até que o thread atual termine e libere o bloqueio. É como quando o banheiro está ocupado quando você chega: você tem que esperar até que o usuário atual termine e destranque a porta.

Java facilita bastante a sincronização do acesso aos recursos. A linguagem trata da configuração e aquisição

de bloqueios; tudo que você precisa fazer é especificar os recursos para sincronizar.

Serializando acesso a métodos

O motivo mais comum para sincronizar threads em Java é serializar seu acesso a um recurso (como um objeto ou variável) — em outras palavras, para garantir que apenas um thread por vez possa manipular esse objeto.5Em Java, cada classe e cada instância de uma classe possui seu próprio bloqueio. A palavra-chave sincronizada marca os locais onde um thread deve adquirir o bloqueio antes de prosseguir.

Por exemplo, suponha que implementemos uma classe SpeechSynthesizer que contém um método say(). Não queremos vários threads chamando say() ao mesmo tempo porque não seríamos capazes de entender nada do que o sintetizador diz. Portanto, marcamos o método say() como sincronizado, o que significa que um thread deve adquirir o bloqueio no objeto SpeechSynthesizer antes de poder falar: classe Sintetizador de Fala {

void sincronizado dizer(palavras de string) {

// fala as palavras fornecidas

}

}

Quando say() for concluído, o thread de chamada desiste do bloqueio, o que permite que o próximo thread em espera adquira o bloqueio e execute o método. Não importa se o thread pertence ao próprio SpeechSynthesizer ou a algum outro objeto; cada

thread deve adquirir o mesmo bloqueio da instância do SpeechSynthesizer. Se say() fosse um método de classe (estático) em vez de um método de instância, ainda poderíamos marcá-lo como sincronizado. Nesse caso, como nenhum objeto de instância está envolvido, o bloqueio está no próprio objeto da classe SpeechSynthesizer.

Freqüentemente, você deseja sincronizar vários métodos da mesma classe para que apenas um método modifique ou examine os dados da classe por vez. Todos os métodos estáticos sincronizados em uma classe usam o mesmo bloqueio de objeto de classe. Da mesma forma, todos os métodos de instância em uma classe usam o mesmo bloqueio de objeto de instância. Isso garante que apenas um de um conjunto de métodos sincronizados esteja em execução por vez. Por exemplo, uma classe SpreadSheet pode conter diversas variáveis de instância que representam valores de células, bem como alguns métodos que manipulam todas as células em uma linha:
classe Planilha {

int célulaA1, célulaA2, célulaA3;

sincronizado int sumRow() {

retornar célulaA1 + célulaA2 + célulaA3;

}

sincronizado void setRow(int a1, int a2, int a3) {

célulaA1 = a1;

célulaA2 = a2;

célulaA3 = a3;

```
}

// outras coisas da planilha...

}
```

Os métodos setRow() e sumRow() acessam os valores das células. Você pode ver que podem surgir problemas se um thread estiver alterando os valores das variáveis em setRow() ao mesmo tempo em que outro thread estiver lendo os valores em sumRow(). Para evitar isso, marcamos ambos os métodos como sincronizados.

Quando threads encontram recursos sincronizados, apenas um thread é executado por vez. Se um thread estiver no meio da execução de setRow() quando outro thread tentar chamar sumRow(), o segundo thread deverá esperar até que o primeiro termine de executar setRow() antes de executar sumRow(). Essa sincronização nos permite preservar a consistência da Planilha. A melhor parte é que todo esse bloqueio e espera é feito pelo Java; é invisível para o programador.

Além de sincronizar métodos inteiros, a palavra-chave sincronizada pode ser usada em uma construção especial para proteger blocos menores de código dentro de um método. Nesta forma, também é necessário um argumento explícito que especifica qual bloqueio de objeto adquirir:

```
sincronizado (meuObject) {

// Funcionalidade que necessita de acesso exclusivo aos recursos

}
```

Este bloco sincronizado pode aparecer em qualquer método. Quando um thread chega até ele, o thread deve adquirir o bloqueio em myObject antes de continuar. Desta forma, podemos sincronizar métodos (ou partes de métodos) em diferentes classes da mesma forma que métodos na mesma classe.

Isso significa que um método de instância sincronizado é equivalente a um método com suas instruções sincronizadas no objeto atual:

sincronizado void meuMetodo () {

// corpo do método

}

é equivalente a:

void meuMetodo() {

sincronizado (este) {

// corpo do método

}

}

Podemos demonstrar os fundamentos da sincronização com um cenário clássico de

“produtor/consumidor”. Digamos que temos alguns produtores criando novos recursos e consumidores aproveitando e usando esses mesmos recursos: por exemplo, uma série de web crawlers que coletam imagens online. O “produtor” nisso pode ser um thread (ou vários threads) fazendo o trabalho real de carregar e

analisar páginas da web para procurar imagens e seus URLs. Podemos dizer para colocar esses URLs em uma fila comum. O(s) thread(s) de “consumidor” pegariam o próximo URL

na fila e baixariam a imagem para o sistema de arquivos ou banco de dados. Não tentaremos fazer todas as E/S reais aqui (mais sobre URLs e redes emCapítulo 13),

mas vamos configurar alguns threads de produção e consumo para mostrar como funciona a sincronização.

Sincronizando uma fila de URLs

Vejamos primeiro a fila onde os URLs serão armazenados. É apenas uma lista onde podemos anexar URLs (como Strings) ao final e retirá-las da frente. Usaremos um LinkedList semelhante ao ArrayList que vimos emCapítulo 7. Queremos uma estrutura projetada para acesso e manipulação eficientes:

```
pacote ch09.examples;

importar java.util.LinkedList;

classe pública URLQueue {

LinkedList<String> urlQueue = new LinkedList<>();

público sincronizado void addURL(String url) {

urlQueue.add(url);

}

string sincronizada pública getURL() {

if (!urlQueue.isEmpty()) {
```

```
return urlQueue.removeFirst();

}

retornar nulo;

}

public boolean isEmpty() {

return urlQueue.isEmpty();

}

}
```

Observe que nem todos os métodos são sincronizados! Qualquer thread pode perguntar se a fila está vazia sem atrasar outros threads que possam estar adicionando ou removendo itens. Isso significa que isEmpty() pode reportar uma resposta errada – se o tempo de diferentes threads estiver exatamente errado.

Felizmente, nosso sistema é um tanto tolerante a falhas, então a eficiência de não bloquear a fila apenas para verificar seu tamanho supera o conhecimento mais perfeito.6

Agora que sabemos como armazenaremos e recuperaremos as URLs, podemos criar as classes produtor e consumidor. O produtor executará um loop para simular um rastreador da web criando URLs falsos. Ele irá prefixar esses URLs com um ID de produtor e, em seguida, armazená-los em nossa fila. Aqui está o método run() para URLProducer:

```
execução nula pública() {
```

```
for (int i = 1; i <= urlCount; i++) {

String url = "https://some.url/at/path/" + i;

queue.addURL(producerID + " " + url);

System.out.println(producerID + "produzida" + url);

tentar {

Thread.sleep(delay.nextInt(500));

} catch (InterruptedException, ou seja) {

System.err.println("Produtor " + ID do produtor + "
interrompido. Saindo.

");

quebrar;

}

}

}
```

A classe consumidora é semelhante, com a exceção óbvia de retirar URLs da fila. Ele extrairá uma URL, prefixá-la-á com um ID do consumidor e recomeçará até que os produtores terminem de produzir e a fila esteja vazia:

```
execução nula pública() {

while (keepWorking || !queue.isEmpty()) {

String url = queue.getURL();
```

```
if (url! = nulo) {

System.out.println(consumerID + "consumido" + url);

} outro {

System.out.println(consumerID + "fila vazia ignorada");

}

tentar {

Thread.sleep(delay.nextInt(1000));

} catch (InterruptedException, ou seja) {

System.err.println("Consumidor " + ID do consumidor +

"interrompido. Desistindo.");

quebrar;

}

}

}
```

Podemos começar por executar a nossa simulação com números muito pequenos: dois produtores e dois consumidores. Cada produtor criará apenas três URLs:

```
classe pública URLDemonstração {

public static void main(String args[]) {

Fila URLQueue = new URLQueue();

URLProducer p1 = new URLProducer("P1", 3, fila);
```

```
URLProducer p2 = new URLProducer("P2", 3, fila);

URLConsumer c1 = new URLConsumer("C1", fila);

URLConsumer c2 = new URLConsumer("C2", fila);

System.out.println("Iniciando...");

Tópico tp1 = novo Tópico(p1);

tp1.start();

Tópico tp2 = novo Tópico(p2);

tp2.start();

Tópico tc1 = novo Tópico(c1);

tc1.start();

Tópico tc2 = novo Tópico(c2);

tc2.start();

tentar {

//Aguarde os produtores terminarem de criar urls

tp1.join();

tp2.join();

} catch (InterruptedException, ou seja) {

System.err.println("Interrompida aguardando a conclusão
dos produtores");

}
```

```
c1.setKeepWorking(falso);

c2.setKeepWorking(falso);

tentar {

//Agora espere os trabalhadores limparem a fila

tc1.join();

tc2.join();

} catch (InterruptedException, ou seja) {

System.err.println("Interrompido aguardando a conclusão
dos consumidores"

);

}

System.out.println("Concluído");

}

}
```

Mesmo com esses pequenos números envolvidos, você ainda pode ver os efeitos do uso de vários threads para fazer o trabalho:

```
Iniciando...

C1 pulou fila vazia

C2 pulou fila vazia

P2 produzido https://some.url/at/path/1
```

P1 produzido https://some.url/at/path/1

P1 produzido https://some.url/at/path/2

P2 produzido https://some.url/at/path/2

C2 consumiu P2 https://some.url/at/path/1

P2 produzido https://some.url/at/path/3

P1 produzido https://some.url/at/path/3

C1 consumiu P1 https://some.url/at/path/1

C1 consumiu P1 https://some.url/at/path/2

C2 consumiu P2 https://some.url/at/path/2

C1 consumiu P2 https://some.url/at/path/3

C1 consumiu P1 https://some.url/at/path/3

Feito

Os threads não dão voltas perfeitas, mas cada thread recebe pelo menos algum tempo de trabalho. E os consumidores não estão presos a produtores específicos. A ideia é fazer uso eficiente de recursos limitados. Os produtores podem continuar adicionando tarefas sem se preocupar com quanto tempo cada tarefa levará ou a quem atribuí-la.

Os consumidores, por sua vez, podem realizar uma tarefa sem se preocupar com outros consumidores. Se um consumidor receber uma tarefa simples e terminar antes dos outros consumidores, ele poderá voltar e realizar uma nova tarefa imediatamente.

Tente executar este exemplo você mesmo e aumente alguns desses números. O que acontece com centenas de URLs? O que acontece com centenas de produtores ou consumidores? Em escala, esse tipo de multitarefa é quase obrigatório. Você não encontrará programas grandes por aí que não usem threads para gerenciar pelo menos parte de seu trabalho em segundo plano. O próprio pacote gráfico do Java, Swing, precisa de um thread separado para manter a UI responsiva e correta, não importa quão pequeno seja o seu aplicativo.

Sincronizando threads virtuais

E quanto aos threads virtuais? Eles têm as mesmas preocupações de simultaneidade?

Principalmente sim. Embora leves, os threads virtuais ainda representam o conceito padrão de “thread de execução”. Eles ainda podem interromper uns aos outros de maneira confusa e ainda precisam coordenar o acesso a recursos compartilhados.

Mas, felizmente, os objetivos de design do Projeto Loom vêm em socorro. Podemos reutilizar todos os nossos truques de sincronização com threads virtuais. Na verdade, para criar threads virtuais em nossa demonstração de produção e consumo de URL,

tudo o que precisamos fazer é substituir o pedaço de código no método main() que inicia os threads:

```
// arquivo: URLDemo2.java

System.out.println("Iniciando threads virtuais...");

// Converta estas linhas de duas etapas:
```

```
//Tópico tp1 = novo Tópico(p1);

//tp1.start();

// Para estas linhas mais simples de criação e início:

Tópico vp1 = Thread.startVirtualThread(p1);

Thread vp2 = Thread.startVirtualThread(p2);

Tópico vc1 = Thread.startVirtualThread(c1);

Tópico vc2 = Thread.startVirtualThread(c2);
```

Os threads virtuais respeitam a palavra-chave sincronizada em nossos métodos URLQueue e entendem as chamadas join() exatamente como os threads de

plataforma. Se você compilar URLDemo2.java e executá-lo (não esqueça que pode ser necessário ativar os recursos de visualização), você verá a mesma saída de antes, com pequenas variações das pausas aleatórias, é claro.

Dissemos que os threads virtuais têm principalmente as mesmas preocupações de simultaneidade que os threads de plataforma. Adicionamos isso porque criar e executar threads virtuais é muito mais barato do que gerenciar um conjunto de threads de plataforma, para não sobrecarregar o sistema operacional. (Lembre-se de que cada thread de plataforma é mapeado para um thread nativo.) Você não agrupa threads virtuais – você apenas cria mais.

Acessando variáveis de classe e instância de vários threads

No exemplo do SpreadSheet, protegemos o acesso a um conjunto de variáveis de instância com um método sincronizado para evitar que um thread alterasse uma das variáveis enquanto outro thread estava lendo as outras, para mantê-las coordenadas.

Mas e os tipos de variáveis individuais? Eles precisam ser sincronizados?

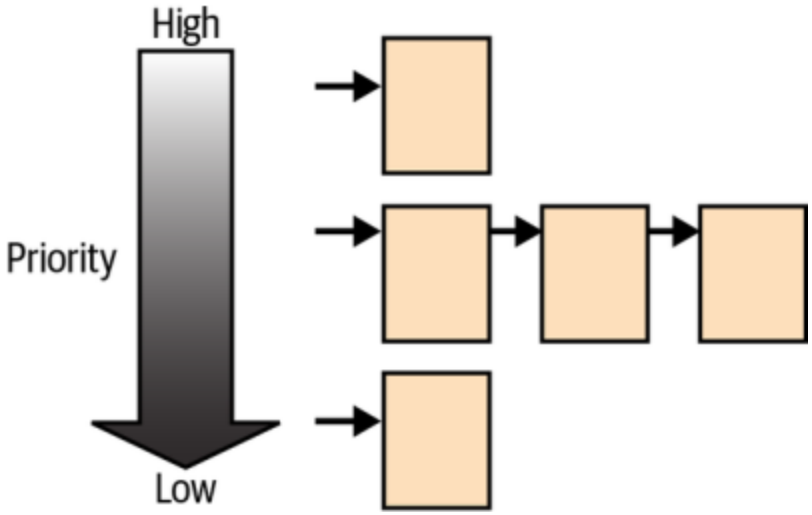
Normalmente, não. Quase todas as operações em primitivos e tipos de referência de objeto em Java acontecem atomicamente: ou seja, a JVM as trata em uma única etapa, sem oportunidade de colisão de dois threads. Isso evita que threads procurem referências enquanto outros threads as acessam.

Aviso

Cuidado: a especificação da JVM não garante que ela lidará com tipos primitivos duplos e longos atomicamente. Ambos os tipos representam valores de 64 bits. O

problema tem a ver com o funcionamento da pilha da JVM. Você deve sincronizar o acesso às suas variáveis de instância duplas e longas por meio de métodos de acesso ou usar uma classe wrapper atômica, que descrever[emos em“Utilitários de

simultaneidade”.]



Agendamento e Prioridade

Java oferece poucas garantias sobre como agenda threads. Quase todo o agendamento de threads é deixado para a implementação Java e, até certo ponto, para a aplicação.

Os projetistas de Java poderiam ter especificado um algoritmo de escalonamento, mas um único algoritmo não é adequado para todas as funções que Java pode desempenhar. Em vez disso, os designers de Java colocam sobre você a

responsabilidade de escrever um código robusto que funcione independentemente do algoritmo de escalonamento e deixar a implementação ajustar o algoritmo para o melhor ajuste.

As regras de prioridade na especificação da linguagem Java são cuidadosamente redigidas para serem uma diretriz geral para o agendamento de threads. Você deve poder confiar nesse comportamento em geral (estatisticamente), mas não é uma boa ideia escrever código que dependa de recursos muito específicos do agendador para funcionar corretamente. Em vez disso,

use as ferramentas de controle e sincronização descritas neste capítulo para coordenar seus threads.7

Cada thread tem uma prioridade. Em geral, sempre que um thread de prioridade mais alta que o thread atual se torna executável (é iniciado, para de dormir ou é notificado), ele interrompe o thread de prioridade mais baixa e começa a ser executado. Em alguns sistemas, threads com a mesma prioridade são agendados round-robin, o que significa que quando um thread começa a ser executado, ele continua até executar um dos seguintes procedimentos:

•

Dorme, chamando Thread.sleep() ou wait()

•

Aguarda um bloqueio para executar um método sincronizado

•

Bloqueia uma E/S, por exemplo, em uma chamada read() ou accept()

•

Gera explicitamente o controle, chamando yield()

•

Termina completando seu método de destino8

Esta situação se parece com algoFigura 9-8.

Figura 9-8. Agendamento round-robin com prioridade prioritária

Você pode definir uma prioridade em um thread de plataforma com o método setPriority() e pode ver a prioridade atual de um thread usando a chamada complementar getPriority(). As prioridades devem estar dentro de um intervalo,

limitado pelas constantes da classe Thread MIN_PRIORITY e MAX_PRIORITY. A prioridade padrão é mantida na constante NORM_PRIORITY.

Observação

Todos os threads virtuais são executados com NORM_PRIORITY. Se você chamar setPriority() em um thread virtual, a nova prioridade passada será simplesmente ignorada.

Estado do tópico

Em qualquer momento do seu ciclo de vida, um thread está em um dos cinco estados gerais. Você pode consultá-los usando o método getState() da classe Thread: **NOVO**

O tópico foi criado, mas ainda não foi iniciado.

EXECUTÁVEL

O thread está em seu estado ativo normal, mesmo se estiver bloqueado em uma operação de E/S, como leitura ou gravação em um arquivo ou em uma conexão de rede.

BLOQUEADO

O thread está bloqueado, aguardando para inserir um método sincronizado ou bloco de código. Isso inclui momentos em que um thread foi despertado por um notify() e está tentando readquirir seu bloqueio após um wait().

ESPERANDO, TIMED_WAITING

O thread está aguardando outro thread por meio de uma chamada para wait() ou join(). No caso de TIMED_WAITING, a chamada possui timeout.

TERMINADO

O encadeamento foi concluído devido a um retorno, uma exceção ou interrupção.

Você pode mostrar o estado de todos os threads de plataforma no grupo de threads atual com o seguinte trecho de código:

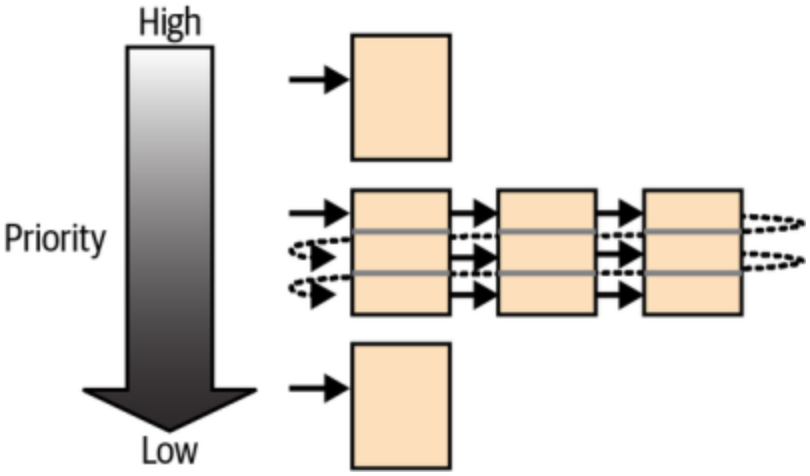
Tópico [] tópicos = novo Tópico [64]; // máximo de threads para mostrar int num = Thread.enumerate(threads);

for(int i = 0; i < num; i++)

System.out.println(threads[i] +":"+ threads[i].getState()); A chamada Thread.enumerate() preencherá nosso array de threads até seu comprimento. Você provavelmente não usará esse método em programação geral, mas é interessante e útil para experimentar e aprender sobre threads Java.

Fatiamento de tempo

Além da priorização, todos os sistemas modernos (com exceção de alguns ambientes embarcados e “micro” Java) implementam divisão de tempo de thread. Em um sistema



dividido em intervalos de tempo, o processamento de threads é dividido de modo que cada thread seja executado por um curto período de tempo antes que o contexto seja alternado para o próximo thread, conforme mostrado em Figura 9-9.

Figura 9-9. Agendamento com divisão de tempo prioritário e preventivo Threads de prioridade mais alta ainda antecipam threads de prioridade mais baixa neste esquema. Adicionar divisão de tempo mistura o processamento entre threads de mesma prioridade; em uma máquina multiprocessadora, os threads podem até ser executados simultaneamente. Isso pode alterar o comportamento de aplicativos que não usam threads e sincronização adequadamente.

A rigor, como Java não garante divisão de tempo, você não deve escrever código que dependa desse tipo de agendamento; qualquer software que você escrever deve funcionar sob agendamento round-robin. Se você está se

perguntando o que seu tipo específico de Java faz, tente o seguinte experimento:

```
classe pública Thready {

public static void main(String args []) {

novo Thread(new ShowThread("Foo")).start();

novo Thread(new ShowThread("Bar")).start();

}

classe estática ShowThread implementa Runnable {

Mensagem de sequência;

ShowThread(String mensagem) {

esta.mensagem = mensagem;

}

execução nula pública() {

enquanto (verdadeiro)

System.out.println(mensagem);

}

}

}
```

Ao executar este exemplo, você verá como sua implementação Java faz seu agendamento. A classe Thready inicia dois objetos ShowThread. ShowThread é

um thread que entra em um loop interminável9(geralmente de má forma, mas útil para esta demonstração) e imprime sua mensagem. Como não especificamos uma prioridade para nenhum dos threads, ambos herdam a prioridade do seu criador, portanto têm a mesma prioridade. Sob um esquema round-robin, apenas Foo deve ser

impresso; A barra nunca deve aparecer. Em uma implementação de divisão de tempo, você deverá ver ocasionalmente as mensagens Foo e Bar alternadas.

A pasta ch09/examples também contém um exemplo VirtualThready se você quiser ver como os threads virtuais se comportam. Eles funcionam com um agendador que

“rouba trabalho”. (Sinta-se à vontade para mergulhar nodocumentos oficiais da

Oraclena estrutura fork/join onde esse algoritmo é apresentado.) Tivemos que adicionar algumas chamadas join() à versão do thread virtual. Ao contrário dos threads de plataforma, os threads virtuais não manterão a JVM “ativa” sem essas solicitações explícitas para aguardar a conclusão dos threads.

Prioridades

As prioridades de thread são uma diretriz geral sobre como a JVM deve alocar tempo entre threads concorrentes. Infelizmente, os threads da plataforma Java são mapeados para threads nativos de maneiras tão complexas que você não pode confiar no significado exato das prioridades. Em vez disso, considere-os uma dica para a JVM.

Vamos brincar com a prioridade dos nossos threads:

```
classe Thready2 {

public static void main(String args []) {

Tópico foo = new ShowThread("Foo");

foo.setPriority(Thread.MIN_PRIORITY);

Barra de tópicos = new ShowThread("Barra");

bar.setPriority(Thread.MAX_PRIORITY);

foo.start();

barra.start();

}

}
```

Você pode esperar que, com essa mudança em nossa classe Thready2, o thread Bar assuma completamente o controle. Se você executar esse código em uma

implementação Solaris antiga do Java 5.0, é exatamente isso que acontece. O mesmo não acontece na maioria das versões modernas do Java. Da mesma forma, se você alterar as prioridades para valores diferentes de mínimo e máximo, poderá não ver nenhuma diferença. As sutilezas de prioridade e desempenho estão relacionadas a como os threads e prioridades Java são mapeados para threads reais no sistema operacional. Por esse motivo, você geralmente deve reservar o ajuste das prioridades dos threads para o desenvolvimento do sistema e da estrutura.

Desempenho de thread

O uso de threads ditou a forma e a funcionalidade de vários pacotes Java.

O custo da sincronização

Adquirir bloqueios para sincronizar threads leva tempo, mesmo quando não há contenção. Em implementações mais antigas de Java, este tempo pode ser significativo. Com JVMs mais recentes, é quase insignificante. No entanto, a sincronização desnecessária de baixo nível ainda pode retardar os aplicativos, bloqueando threads onde o acesso simultâneo poderia ser permitido. Para evitar essa penalidade, duas APIs importantes, a estrutura de coleções Java e a API Swing, foram criadas especificamente para colocar a sincronização sob o controle do desenvolvedor.

A estrutura de coleções java.util substitui tipos agregados Java simples anteriores - ou seja, Vector e Hashtable - por tipos mais completos e, notavelmente, não sincronizados (Lista e Mapa). Em vez disso, a estrutura de coleções recorre ao código do aplicativo para sincronizar o acesso às coleções quando necessário e fornece funcionalidade especial de “falha rápida” para ajudar a detectar o acesso simultâneo e lançar uma exceção. Ele também fornece “invólucros” de sincronização que podem fornecer acesso seguro no estilo antigo. Implementações especiais de fácil acesso simultâneo das coleções Map e Queue estão incluídas como parte do pacote java.util.concurrent. Essas implementações vão ainda mais longe e permitem um alto grau de acesso simultâneo sem qualquer sincronização do usuário.

A API Java Swing adota uma abordagem diferente para fornecer velocidade e segurança. Swing usa um único thread para modificar seus componentes, com uma exceção: o thread de despacho de eventos, também chamado de fila de eventos. O

Swing resolve problemas de desempenho e quaisquer problemas de ordenação de eventos, forçando um único superthread a controlar a GUI. O aplicativo acessa o thread de despacho de eventos indiretamente, enviando comandos para uma fila por meio de uma interface simples. Veremos como fazer exatamente isso emCapítulo 12.

Consumo de recursos de thread

Um padrão fundamental em Java é iniciar muitos threads para lidar com recursos externos assíncronos, como conexões de soquete. Para obter eficiência máxima, um desenvolvedor web pode ficar tentado a criar um thread para cada conexão de cliente em um servidor. Quando cada cliente tem seu próprio thread, as operações de E/S

podem ser bloqueadas e retomadas conforme necessário. Mas por mais eficiente que isso possa ser em termos de rendimento para um determinado cliente, é um uso muito ineficiente dos recursos do servidor.

Threads consomem memória; cada thread tem sua própria “pilha” para variáveis locais, e a alternância entre threads em execução (conhecida como troca de contexto) adiciona sobrecarga à CPU. Os threads são relativamente leves. É possível ter centenas ou milhares de servidores rodando em um servidor grande. Mas depois de um certo ponto, o custo de gerenciar os threads existentes começa a superar os benefícios de

iniciar mais threads. Criar um thread por cliente nem sempre é uma opção escalável.

Uma abordagem alternativa é criar “pools de threads”, onde um número fixo de threads extrai tarefas de uma fila e retorna para mais trabalho quando terminadas.

Essa reciclagem de threads proporciona escalabilidade sólida, mas muitas vezes tem sido difícil de implementar de forma eficiente para servidores em Java. E/S básica (para coisas como soquetes) em Java não oferece suporte total a operações sem bloqueio. O pacote java.nio, New I/O (ou simplesmente NIO), possui canais de I/O

assíncronos. Os canais podem realizar leituras e gravações sem bloqueio. Eles também têm a capacidade de testar a prontidão dos fluxos para movimentação de dados.

Threads podem fechar canais de forma assíncrona, proporcionando interações elegantes. Discutiremos NIO nos próximos capítulos sobre como trabalhar com arquivos e conexões de rede.

Java fornece pools de threads e serviços de “executor” de tarefas como parte do pacote java.util.concurrent. Isso significa que você não precisa escrevê-los sozinho. Iremos resumi-los quando discutirmos os utilitários de simultaneidade em Java.

Desempenho de thread virtual

O Projeto Loom teve como objetivo melhorar o desempenho dos threads –

especialmente quando milhares ou milhões de threads estão envolvidos. O código de um método run() não é mais rápido ou mais lento quando você o executa em um thread de plataforma em vez de um thread virtual. O que é mais rápido, porém, é criar e gerenciar esses threads.

Vamos dar uma olhada em nossa classe URLDemo. Em vez de quatro threads no total, vamos aumentar esse número para vários milhares. Eliminaremos nossos produtores e preencheremos previamente a fila com URLs para que possamos nos concentrar em nossos novos consumidores. Criaremos consumidores cujo único trabalho é consumir um URL – sem atrasos aleatórios e artificiais antes de voltar para outro URL. Esse comportamento imita um caso de uso real para threads virtuais: um único servidor que lida com milhões de pequenas solicitações em um curto período. Também modificaremos nossas instruções de impressão para que apareçam em marcos, e não depois que cada URL for consumido. Nosso novo URLDemo3 terá dois argumentos de linha de comando opcionais: o número de URLs a serem criados (o padrão é 100.000) e se devemos usar plataforma ou threads virtuais (o padrão é plataforma), para que possamos comparar a diferença no desempenho.

Confira o código fonte de URLConsumer3 na pasta ch09/examples para ver os ajustes que fizemos para esta nova variação. Então, vamos examinar mais de perto o loop de processamento no método main() para ver como ele lida com os novos consumidores.

Aqui está a seção relevante:

// Cria e preenche nosso objeto de fila compartilhada

```
Fila URLQueue = new URLQueue();

for (int u = 1; u <= contagem; u++) {

queue.addURL("http://some.url/path/" + você);

}

// Agora começa a diversão! Crie um consumidor para
cada URL

for (int c = 0; c < contagem; c++) {

Consumidor URLConsumer3 = new URLConsumer3("C" +
c, fila);

if (useVirtual) {

Thread.startVirtualThread(consumidor);

} outro {

new Thread(consumidor).start();

}

}
```

Este código não tenta reutilizar consumidores. A propósito, existem razões válidas para não reutilizar threads no mundo real. Você precisa limpar manualmente alguns dados compartilhados entre usos, por exemplo. Esqueça esse pouco de “administrivia”

e você poderá vazar informações confidenciais. (Se você estivesse processando transações bancárias, não iria querer usar acidentalmente o número da conta anterior.) Muitas vezes você pode simplificar seu código assumindo

que um único thread fará todo o trabalho e depois encerrará. Isso é verdade quer você esteja usando threads virtuais ou não.

Testamos esta versão com 1.000.000 de URLs em um sistema desktop Linux mediano.

Os threads da plataforma limparam a fila em pouco menos de um minuto (58.661 s de acordo com a utilidade de tempo aproximado). Nada mal! Os threads virtuais, por outro lado, limparam a fila em pouco menos de 2 segundos (1.867 s). O teste para imprimir um URL de marco é trivial. Não é a tarefa que cada consumidor realiza que retarda as coisas. O verdadeiro gargalo dos threads de plataforma é solicitar milhares de vezes ao sistema operacional um recurso novo e caro. O Projeto Loom elimina muitas dessas despesas. Usar threads virtuais não é garantia de melhor desempenho, mas em casos como esse certamente pode ser um benefício!

Utilitários de simultaneidade

Até agora neste capítulo, demonstramos como criar e sincronizar threads em baixo nível, usando primitivas da linguagem Java. O pacote e subpacotes java.util.concurrent baseiam-se nesta funcionalidade, adicionando importantes utilitários de threading e codificando alguns padrões de design comuns, fornecendo implementações padrão.

Aproximadamente em ordem de generalidade, essas áreas incluem:

Implementações de coleções com reconhecimento de thread

O pacote java.util.concurrent aumenta a API de coleções Java emCapítulo 7com

diversas implementações para modelos de threading específicos. Isso inclui implementações de espera cronometrada e de bloqueio da interface Queue, bem como implementações sem bloqueio e otimizadas para acesso simultâneo das interfaces Queue e Map. O pacote também adiciona implementações de lista e conjunto "copiar na gravação" para casos extremamente eficientes de "quase sempre leitura". Eles podem parecer complexos, mas cobrem muito bem alguns casos comuns.

Executores

Executors executam tarefas, incluindo Runnables, e abstraem o conceito de criação e pool de threads do usuário (o que significa que você não precisa escrever o seu próprio). Os executores pretendem ser um substituto de alto nível para a criação de novos threads para atender uma série de trabalhos. Junto com o Executor, as interfaces Callable e Future permitem gerenciamento, retorno de valor e tratamento de exceções.

Construções de sincronização de baixo nível

O pacote java.util.concurrent.locks contém um conjunto de classes, incluindo Lock e Condition, que são paralelas às primitivas de sincronização em nível de linguagem Java e as promovem ao nível de uma API concreta. Por exemplo, a classe auxiliar LockSupport inclui dois métodos, park() e unpark(), que substituem os métodos obsoletos suspend() e resume() da classe Thread. O pacote locks também adiciona o conceito de bloqueios

de leitor/gravador não exclusivos, permitindo maior simultaneidade no acesso sincronizado aos dados.

Construções de sincronização de alto nível

Isso inclui as classes CyclicBarrier, CountDownLatch, Semaphore e Exchanger. Essas classes implementam padrões de sincronização comuns extraídos de outras linguagens e sistemas e podem servir como base para novas ferramentas de alto nível.

Operações atômicas (parece muito James Bond, não é?)

O pacote java.util.concurrent.atomic fornece wrappers e utilitários para operações atômicas do tipo “tudo ou nada” em tipos primitivos e referências. Isso inclui operações atômicas de combinação simples, como testar um valor antes de defini-lo e obter e incrementar um número em uma operação.

Com a possível exceção das otimizações feitas pelo Java VM para o pacote de operações atômicas, todos esses utilitários são implementados em Java puro, além das construções de sincronização da linguagem Java padrão. Isso significa que eles são, de certa forma, apenas utilitários de conveniência e não adicionam realmente novos recursos à linguagem. Sua principal função é oferecer padrões e expressões padrão em threading Java, tornando-os mais seguros e eficientes de usar. Um bom exemplo disso é o utilitário Executor, que permite ao usuário gerenciar um conjunto de tarefas em um modelo de threading predefinido sem ter que se aprofundar na criação de threads. APIs de nível superior como essa simplificam a codificação e permitem maior otimização dos casos comuns.

Atualizando nossa demonstração de fila

Muitos dos recursos de simultaneidade incorporados ao Java serão mais úteis em projetos maiores e mais complexos. Mas ainda podemos atualizar nossa escassa demonstração de processamento de URL usando a classe ConcurrentLinkedQueue thread-safe do pacote java.util.concurrent. Podemos parametrizar seu tipo e eliminar totalmente nossa classe URLQueue personalizada:

```
// Diretório: ch09/exemplos

// em URLDemo4.java

ConcurrentLinkedQueue<String> fila =

new ConcurrentLinkedQueue<>();

// em URLProducer4.java, apenas "add" em vez de "addURL"

fila.add(producerID + " " + url);

// em URLConsumer4.java, "poll" em vez de "getURL"

String url = queue.poll();

// ...
```

Precisamos ajustar um pouco o código do consumidor e do produtor, mas apenas um pouco, e principalmente apenas para usar os nomes normais das operações de fila de add e poll em vez de nossos nomes de métodos personalizados centrados em URL.

Mas não precisamos nos preocupar com a classe URLQueue. Às vezes você precisará de estruturas de

dados personalizadas porque o mundo real é confuso. Mas se você puder usar uma das opções de armazenamento sincronizado integradas, você saberá que está obtendo armazenamento e acesso robustos que podem ser usados com segurança em seu aplicativo multithread.

Outra atualização a considerar são as classes de conveniência atômicas. Você deve se lembrar que nossa classe de consumidor possui um sinalizador booleano que pode ser definido como falso para encerrar o loop de processamento do consumidor. Como é razoável supor que vários threads possam ter acesso ao nosso consumidor, podemos refazer esse sinalizador como uma instância da classe AtomicBoolean para garantir que os threads em conflito não consigam derrotar nosso sinalizador pobre.

(Poderíamos sincronizar nosso método acessador, é claro, mas queremos destacar algumas das opções existentes já no JDK.) Aqui está uma olhada nas partes interessantes do URLConsumer4:

AtomicBoolean keepWorking;

//...

execução nula pública() {

enquanto (keepWorking.get() || !queue.isEmpty()) {

String url = queue.poll();

//...

}

```
}

public void setKeepWorking(boolean newState) {

keepWorking.set(novoEstado);

}
```

Usar AtomicBoolean requer um pouco mais de digitação - chamar métodos set/get em vez de simples atribuições ou comparações - mas você obtém todo o manuseio seguro que poderia desejar. Quando você tem lógica complexa e multithread em todos os lugares, você pode fazer seu próprio gerenciamento de estado. Porém, em situações

em que você não possui muitos outros códigos que exijam sincronização, essas classes de conveniência podem ser realmente muito convenientes.

Simultaneidade Estruturada

Além das melhorias impressionantes que os threads virtuais trazem para aplicativos altamente simultâneos, o Project Loom também traz simultaneidade estruturada para Java. Você deve ter ouvido falar sobre “programação paralela” no mundo encadeado.

Você tem a opção de buscar uma solução de programação paralela quando puder dividir um problema maior em problemas menores que podem ser resolvidos separadamente e ao mesmo tempo (em paralelo, entendeu?).

Esta noção de uma tarefa grande que pode ser transformada em subtarefas partilha muitas semelhanças com a nossa demonstração que utiliza produtores e

consumidores, mas os dois tipos de problemas não são inteiramente iguais. Uma grande diferença está em como lidar com erros. Se não conseguirmos criar um consumidor em nossas classes URLDemo, por exemplo, poderíamos simplesmente criar outro e continuar. Mas se uma subtarefa falhar em um cálculo paralelo, não será tão óbvio como recuperá-la. Todas as outras subtarefas devem ser canceladas? E se alguns deles já tiverem sido concluídos? E se quisermos cancelar a tarefa "pai" maior?

Java 19 introduziu um recurso de incubadora, a classe StructuredTaskScope, para encapsular melhor o trabalho realizado com subtarefas. (Se você chamasse um recurso de visualização, como threads virtuais, de aprimoramento "beta", os recursos da incubadora seriam um aprimoramento "alfa".) Você pode ler sobre os objetivos de design e detalhes de implementação emPEC 428. Não trabalharemos com simultaneidade estruturada ou executores neste livro, mas é importante saber que Java possui muitas ferramentas disponíveis para desenvolvedores que trabalham com aplicações paralelas e simultâneas. Na verdade, o suporte que ele fornece aos desenvolvedores nesta área é precisamente o motivo pelo qual o Java continua sendo um burro de carga tão popular em back-ends de produção.

Tantos fios para puxar

Embora não iremos nos aprofundar nos pacotes de simultaneidade neste capítulo, queremos que você saiba o que fazer em seguida se a simultaneidade for interessante para você ou se for útil no tipo de problemas que você encontra no trabalho. Como nós (pé) observamos em"Sincronizando uma fila de URLs", *Simultaneidade Java na*

prática de Brian Goetz, é leitura obrigatória para projetos multithread do mundo real.

Também queremos agradecer a Doug Lea, autor de Concurrent Programming in Java (Addison-Wesley), que liderou o grupo que adicionou esses pacotes ao Java e é o grande responsável por criá-los.

Juntamente com os threads, o suporte nativo do Java para entrada e saída (E/S) de arquivos básicos figura com destaque em aplicativos de produção. Veremos as classes principais para E/S típicas no próximo capítulo.

Perguntas de revisão

1. O que é um tópico?

2. Que palavra-chave você pode adicionar a um método se quiser que os threads

"se revezem" ao chamá-lo? (O que significa que dois threads não devem executar o método ao mesmo tempo para evitar corromper os dados

compartilhados.)

3. Quais sinalizadores permitem compilar um programa Java que inclui código de recurso de visualização?

4. Quais sinalizadores permitem executar um programa Java que inclui código de recurso de visualização?

5. Quantos threads de plataforma um thread nativo pode suportar?

6. Quantos threads virtuais um thread nativo pode suportar?

7. A afirmação é x = x + 1; uma ação atômica para a variável x?

8. Qual pacote inclui versões thread-safe de classes de coleção populares como Queue e Map?

Exercícios de código

1. Vamos construir um relógio! Usando uma agulha e linha – ou seja, um JLabel e um Thread – crie um pequeno aplicativo de relógio gráfico. O arquivo

Clock.java na pasta ch09/exercises contém um aplicativo esqueleto que exibe uma pequena janela com um objeto JLabel simples. Aumentamos o tamanho da fonte do rótulo para tornar as coisas mais legíveis. Seu relógio deve mostrar horas, minutos e segundos, no mínimo. Crie um thread que ficará suspenso por um segundo e depois incrementará a exibição do relógio. Sinta-se à vontade para revisitar os exemplos de formatação de data e hora de“Formatação de

datas e horários”.

2. O jogo de arremesso de maçã na pasta ch09/exercises/game atualmente usa um thread de plataforma para sua primeira incursão no mundo da animação após a discussão em“Revisitando Animação com Threads”. Compile e execute-o para ver uma maçã ser iniciada quando o jogo começar. A maçã não atingirá nada, mas se moverá em arco, como se tivesse sido lançada. Tornaremos esta animação mais interessante e interativa emCapítulo 12.

Depois de sentir a animação pretendida, converta o thread da plataforma em um thread virtual. Compile sua nova versão e verifique se ela ainda funciona conforme o

esperado. (Lembre-se de que, dependendo da sua versão do Java, pode ser necessário compilar e executar com sinalizadores de visualização extras.)

1Historicamente, interrupt() não funcionou de forma consistente em todas as implementações Java.

2O termo daemon (frequentemente pronunciado day-mun nos círculos Unix) foi inspirado porO demônio de Maxwelle refere-se ao termo grego para uma divindade menor, não um espírito malévolo.

3Muitos aprimoramentos do Java começam como trabalhos em andamento com nomes sofisticados como “Loom”.

4Infelizmente, os prefixos de traço simples e duplo nessas opções não são erros de digitação. Os argumentos de linha de comando têm uma história bastante longa por si só, e Java e suas ferramentas são antigos o suficiente para terem herdado alguns dos padrões legados, embora ainda precisem acomodar abordagens modernas. A maioria das opções funciona com qualquer um dos prefixos, mas ocasionalmente você precisa obedecer ao que parece ser uma regra não escrita. Em caso de dúvida, ferramentas como javac suportam outra opção: -help (ou --help). Fornecer esse argumento imprimirá uma lista concisa de opções e detalhes relevantes.

5Não confunda o termo serializar neste contexto com serialização de objetos Java, que é um mecanismo para tornar objetos persistentes. O significado subjacente (colocar uma coisa após a outra) é o mesmo, entretanto. No caso da serialização de objetos, os dados do objeto são dispostos, byte por byte, em uma determinada

ordem. Com threads, cada thread obtém acesso ao recurso sincronizado por sua vez.

6Mesmo com a capacidade de tolerar pequenas discrepâncias no estado dos objetos, os sistemas multicore modernos podem causar estragos sem um conhecimento perfeito da aplicação. E a perfeição é difícil! Se você espera trabalhar com threads no mundo real, *Simultaneidade Java na prática* por Brian Goetz et al. (Addison-Wesley) é leitura obrigatória.

7Java Threads de Scott Oaks e Henry Wong inclui uma discussão detalhada sobre sincronização, agendamento e outros problemas relacionados a threads.

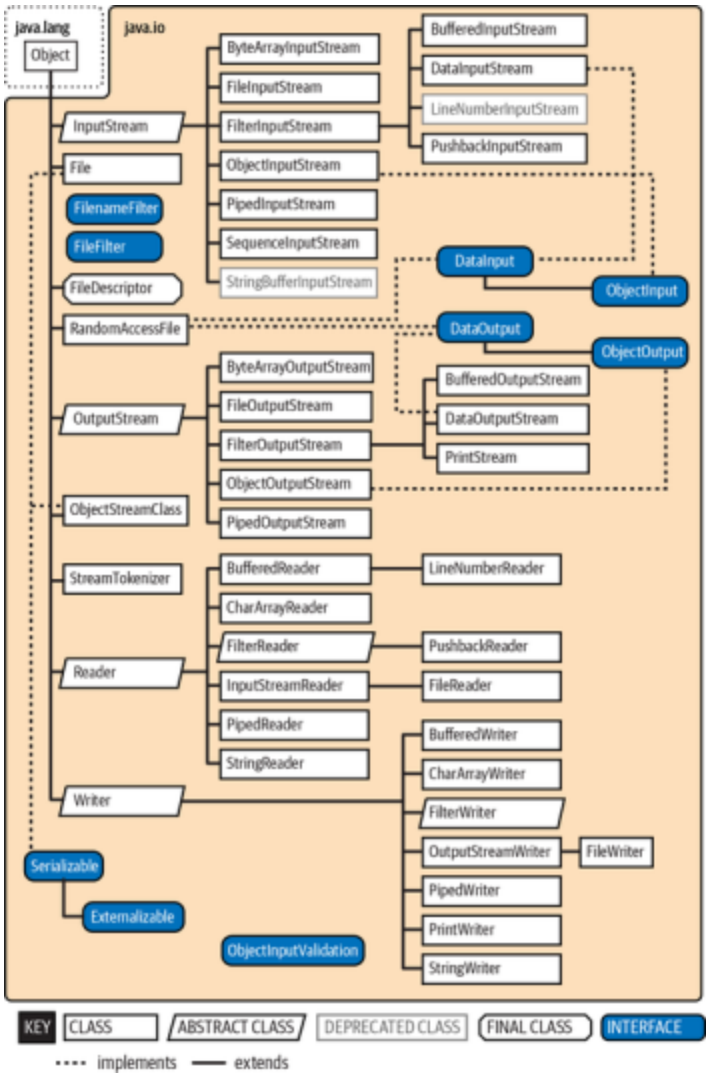
8Tecnicamente, um thread também pode terminar com ochamada stop() obsoleta,

mas como observamos no início do capítulo, isso é ruim por inúmeras razões.

9Você pode digitar Control-C para sair da demonstração quando se cansar de ver Foos voando.

Capítulo 10. Entrada e Saída de Arquivo

A capacidade de armazenar dados em arquivos e recuperá-los posteriormente é crucial para aplicativos de desktop e corporativos. Neste capítulo, veremos algumas das classes mais populares nos pacotes java.io e java.nio. Esses pacotes oferecem um rico conjunto de ferramentas para entrada e saída (E/S) básicas e também fornecem a



estrutura na qual toda a comunicação de arquivos e rede em Java é construída. Figura 10-1 mostra a amplitude do pacote java.io.

Começaremos examinando as classes de stream em java.io, que são subclasses das classes básicas InputStream, OutputStream, Reader e Writer. Em seguida, examinaremos a classe File e discutiremos como você pode ler e gravar arquivos usando classes em java.io. Também daremos uma olhada rápida na compactação e serialização de dados. Ao longo do caminho, apresentamos o pacote java.nio. O “novo”

pacote de E/S (ou NIO) adiciona funcionalidades significativas adaptadas para a construção de serviços de alto desempenho. O NIO se concentra no trabalho com buffers (onde você armazena coisas para fazer uso mais eficiente de outros recursos) e canais (onde você pode colocar com eficiência coisas que podem ser coletadas com a mesma eficiência por outras pessoas). Em alguns casos, o NIO também fornece APIs melhores que podem ser usadas no lugar de alguns recursos do java.io.1

Figura 10-1. A hierarquia de classes java.io

Fluxos

A maior parte da E/S em Java é baseada em fluxos. Conceitualmente, um fluxo representa um fluxo de dados com um gravador em uma extremidade e um leitor na outra. Quando você está trabalhando com o pacote java.io para executar entrada e saída de terminal, lendo ou gravando arquivos ou se comunicando através de soquetes de rede em Java (mais sobre redes emCapítulo 13), você está usando vários tipos de fluxos. Quando olharmos para o pacote NIO, encontraremos um conceito semelhante chamado canal. A principal diferença entre os dois é que os fluxos são orientados em torno de bytes ou caracteres, enquanto os canais são orientados em torno de “buffers”

contendo esses tipos de dados. Um buffer normalmente é um armazenamento rápido e temporário de dados que facilita a otimização do rendimento. Ambos executam aproximadamente o mesmo trabalho. Vamos começar com fluxos. Aqui está uma rápida visão geral das classes de stream mais populares:

Fluxo de entrada, OutputStream

Classes abstratas que definem a funcionalidade básica para leitura ou gravação de uma sequência não estruturada de bytes. Todos os outros fluxos de bytes em Java são construídos sobre o InputStream e OutputStream básicos.

Leitor, Escritor

Classes abstratas que definem a funcionalidade básica para leitura ou escrita de uma sequência de dados de caracteres, com suporte para Unicode. Todos os outros fluxos de caracteres em Java são construídos sobre o Reader e o Writer.

Leitor de entradaStream, OutputStreamWriter

Classes que conectam fluxos de bytes e caracteres convertendo de acordo com um esquema específico de codificação de caracteres, como ASCII ou Unicode. (Lembre-se: em Unicode, um caractere não é necessariamente um byte!)

DataInputStream, DataOutputStream

Filtros de fluxo especializados que adicionam a capacidade de ler e gravar tipos de dados multibyte, como primitivos numéricos e objetos String em um formato padronizado.

ObjectInputStream, ObjectOutputStream

Filtros de fluxo especializados que são capazes de gravar grupos inteiros de objetos Java serializados e reconstruí-los.

BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter Filtros de fluxo

especializados que adicionam buffer para maior eficiência. Para E/S do mundo real, um buffer é quase sempre usado.

ImprimirStream, PrintWriter

Fluxos especializados que simplificam a impressão de texto.

PipedInputStream, PipedOutputStream, PipedReader, PipedWriter

Classes emparelhadas que movem dados dentro de um aplicativo. Os dados gravados em um PipedOutputStream ou PipedWriter são lidos de seu PipedInputStream ou PipedReader correspondente.

FileInputStream, FileOutputStream, FileReader, FileWriter

Implementações de InputStream, OutputStream, Reader e Writer que leem e gravam em arquivos no sistema de arquivos local.

Streams em Java são ruas de mão única. As classes de entrada e saída java.io representam apenas as extremidades de um fluxo simples. Para conversas bidirecionais, você usará um de cada tipo de stream.

Fluxo de entradae OutputStream, conforme mostrado emFigura 10-2, são classes abstratas que definem a interface de nível mais baixo para todos os fluxos de bytes.

Eles contêm métodos para ler ou escrever um fluxo não estruturado de dados em nível de byte. Como essas

classes são abstratas, não é possível criar um fluxo genérico de entrada ou saída.

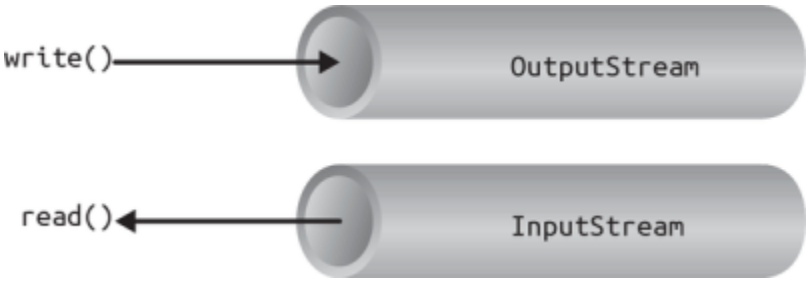


Figura 10-2. Funcionalidade básica de fluxo de entrada e saída

Java implementa subclasses destes para atividades como leitura e gravação em arquivos ou comunicação com conexões de rede. Como todos os fluxos de bytes herdam a estrutura de InputStream ou OutputStream, os vários tipos de fluxos de bytes podem ser usados de forma intercambiável. Um método que especifica um InputStream como argumento pode aceitar qualquer subclasse de InputStream. Tipos especializados de fluxos também podem ser colocados em camadas ou agrupados em fluxos básicos para adicionar recursos como buffer, filtragem, compactação ou manipulação de tipos de dados de nível superior.

Leitore Writer são muito parecidos com InputStream e OutputStream, exceto que lidam com caracteres em vez de bytes. Como verdadeiros fluxos de caracteres, essas classes lidam corretamente com caracteres Unicode, o que nem sempre é o caso dos fluxos de bytes. Freqüentemente, é necessária uma ponte entre esses fluxos de caracteres e os fluxos de bytes de dispositivos físicos, como discos e redes.

InputStreamReader e OutputStreamWriter são classes especiais que usam um esquema de codificação de caracteres como ASCII ou UTF-8 para traduzir entre fluxos de caracteres e bytes.

Esta seção descreve vários tipos de fluxo, com exceção de FileInputStream, FileOutputStream, FileReader e FileWriter. Adiamos a discussão sobre fluxos de arquivos para a próxima seção, onde abordaremos o acesso ao sistema de arquivos em Java.

E/S básica

O exemplo prototípico de um objeto InputStream é a entrada padrão de um aplicativo Java. Como stdin em C ou cin em C++, esta é a fonte de entrada para um programa de linha de comando (não GUI). É um fluxo de entrada do ambiente – geralmente uma janela de terminal ou possivelmente a saída de outro comando. A classe java.lang.System, um repositório geral para recursos relacionados ao sistema, fornece uma referência ao fluxo de entrada padrão na variável estática System.in. Ele também fornece um fluxo de saída padrão e um fluxo de erro padrão nas variáveis out e err, respectivamente.2O exemplo a seguir mostra a correspondência:

```
InputStream stdin = System.in;

OutputStream stdout = System.out;

OutputStream stderr = System.err;
```

Este trecho esconde o fato de que System.out e System.err não são apenas objetos OutputStream, mas objetos PrintStream mais especializados e úteis. Explicaremos isso mais tarde“PrintWriter e PrintStream”,

mas por enquanto podemos referenciar out e err como objetos OutputStream porque eles são derivados de OutputStream.

Você pode ler um único byte por vez da entrada padrão com o método read() do InputStream. Se você olhar atentamente paradocumentação on-line, você verá que o método read() da classe base InputStream é um método abstrato. O que está por trás do System.in é uma implementação específica do InputStream que fornece a implementação real do método read():

```
tentar {

int val = System.in.read();

} catch (IOException e) {

// ...

}
```

Embora tenhamos dito que o método read() lê um valor de byte, o tipo de retorno no exemplo é int, não byte. Isso ocorre porque o método read() de fluxos de entrada básicos em Java usa uma convenção herdada da linguagem C para indicar o fim de um fluxo com um valor especial. Os valores de bytes são retornados no intervalo de 0 a 255, e o valor especial de -1 é usado para indicar que o final do fluxo foi alcançado.

Você testa essa condição ao usar o método read() simples. Você pode então converter o valor em um byte, se necessário. O exemplo a seguir lê cada byte de um fluxo de entrada e imprime seu valor:

```
tentar {
```

valor interno;

enquanto((val=System.in.read()) != -1) {

System.out.println((byte)val);

}

} catch (IOException e) {

// Ops. Lidar com o erro ou imprimir uma mensagem de erro

}

Como mostramos nos exemplos, o método read() também pode lançar uma

IOException se houver um erro ao ler a origem do fluxo subjacente. Várias subclasses de IOException podem indicar que uma fonte (como um arquivo ou conexão de rede) apresentou um erro. Além disso, fluxos de nível superior que leem tipos de dados mais complexos do que um único byte podem lançar EOFException (“fim de arquivo”), o que indica um final inesperado ou prematuro de um fluxo.

Uma forma sobrecarregada de read() preenche um array de bytes com o máximo de dados possível até a capacidade do array e retorna o número de bytes lidos: byte [] buff = novo byte [1024];

int obtido = System.in.read(buff);

Em teoria, também podemos verificar o número de bytes disponíveis para leitura em um determinado momento em um InputStream usando o método available(). Com

essas informações, poderíamos criar um array exatamente do tamanho certo: int esperando = System.in.available();

if (esperando > 0) {

byte [] dados = novo byte [esperando];

System.in.read(dados);

// ...

}

No entanto, a confiabilidade desta técnica depende se a implementação do fluxo subjacente pode detectar a quantidade de dados que pode recuperar. Geralmente funciona para arquivos, mas não deve ser usado para todos os tipos de fluxos.

Esses métodos read() são bloqueados até que pelo menos alguns dados sejam lidos (pelo menos um byte). Você deve, em geral, verificar o valor retornado para determinar quantos dados obteve e se precisa ler mais. (Veremos E/S sem bloqueio posteriormente neste capítulo.) O método skip() de InputStream fornece uma maneira de saltar vários bytes. Dependendo da implementação do fluxo, pular bytes pode ser mais eficiente do que lê-los.

O método close() encerra o fluxo e libera quaisquer recursos do sistema associados. É

importante para o desempenho lembrar de fechar a maioria dos tipos de fluxos quando terminar de usá-los. Em alguns casos, os fluxos podem fechar

automaticamente quando os objetos são coletados como lixo, mas não é uma boa ideia confiar nesse comportamento. O recurso try-with-resources discutido em“tente com

recursos”facilita o fechamento automático de fluxos e outras entidades que podem ser fechadas. Veremos alguns exemplos disso em“Fluxos de arquivos”. A interface java.io.Closeable identifica todos os tipos de fluxo, canal e classes de utilitários relacionados que podem ser fechados.

Fluxos de personagens

Nas primeiras versões do Java, alguns tipos InputStream e OutputStream incluíam métodos para ler e escrever strings, mas a maioria deles operava assumindo ingenuamente que um caractere Unicode de 16 bits era equivalente a um byte de 8

bits no fluxo. Isso funciona para caracteres Latin-1 (ISO 8859-1), mas não para o mundo de outras codificações usadas em idiomas diferentes.

As classes de fluxo de caracteres java.io Reader e Writer foram introduzidas como fluxos que manipulam apenas dados de caracteres. Ao usar essas classes, você pensa apenas em termos de caracteres e dados de string. Você permite que a implementação subjacente lide com a conversão de bytes em uma codificação de caracteres específica.

Como você verá, existem algumas implementações diretas do Reader e do Writer, como aquelas para leitura e gravação de arquivos.

De forma mais geral, duas classes especiais, InputStreamReader e

OutputStreamWriter, preenchem a lacuna entre fluxos de caracteres e fluxos de bytes.

Estes são, respectivamente, um Leitor e um Gravador que podem ser agrupados em qualquer fluxo de bytes subjacente para torná-lo um fluxo de caracteres. Um esquema de codificação converte entre os bytes (que podem vir em grupos que representam caracteres multibyte) e os caracteres de dois bytes do Java. Um esquema de codificação pode ser especificado por nome no construtor de InputStreamReader ou

OutputStreamWriter. Por conveniência, o construtor padrão usa o esquema de codificação padrão do sistema.

Vamos ver como usar leitores e a classe java.text.NumberFormat para recuperar a entrada numérica do usuário na linha de comando. Assumiremos que os bytes provenientes de System.in usam o esquema de codificação padrão do sistema:

// arquivo: ch10/examples/ParseKeyboard.java

tentar {

InputStream in = System.in;

InputStreamReader charsIn = novo
InputStreamReader(in);

BufferedReader bufferedCharsIn = novo
BufferedReader(charsIn);

Linha de string = bufferedCharsIn.readLine();

```
int i =
NumberFormat.getInstance().parse(line).intValue();

// ...

} catch (IOException e) {

// ...

} catch (ParseException pe) {

// ...

}
```

Primeiro, envolvemos um InputStreamReader em torno de System.in. Este leitor converte os bytes recebidos de System.in em caracteres usando o esquema de codificação padrão. Em seguida, envolvemos um BufferedReader em torno do InputStreamReader. BufferedReader adiciona o método readLine(), que podemos usar para capturar uma linha completa de texto (até uma combinação de caracteres de terminação de linha específica da plataforma) em uma String. A string é então analisada em um número inteiro usando as técnicas descritas emCapítulo 8. Tente você mesmo. Quando solicitado, tente fornecer informações diferentes. O que acontece se você inserir um “0”? E se você inserir apenas seu primeiro nome?

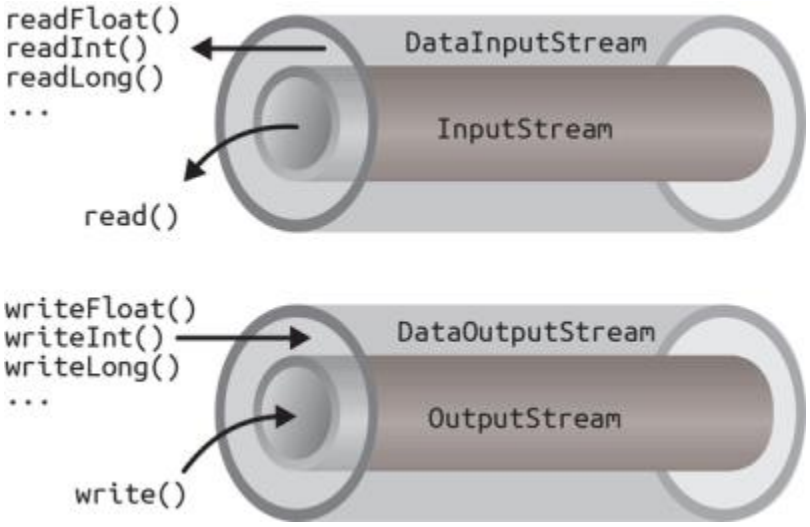
Acabamos de pegar um fluxo de entrada orientado a bytes, System.in, e convertê-lo com segurança em um Reader para leitura de caracteres. Se quiséssemos usar uma codificação diferente da padrão do sistema, poderíamos tê-la especificado no construtor do InputStreamReader, assim:

Leitor InputStreamReader = new InputStreamReader(System.in, "UTF-8"); Para cada caractere lido do leitor, o InputStreamReader lê um ou mais bytes e realiza a conversão necessária para Unicode.

Voltaremos ao tópico de codificações de caracteres em“A nova API de arquivo de

E/S”quando discutimos o pacote java.nio.charset, que permite encontrar e usar codificadores e decodificadores. Tanto InputStreamReader quanto

OutputStreamWriter podem aceitar um objeto de codec Charset, bem como um nome de codificação de caracteres.



Wrappers de fluxo

E se você quiser fazer mais do que ler e escrever uma sequência de bytes ou caracteres? Podemos usar um fluxo de filtro, que é um tipo de InputStream, OutputStream, Reader ou Writer que envolve outro fluxo e adiciona novos recursos.

Um fluxo de filtro utiliza o fluxo de destino como argumento em seu construtor, faz algum processamento adicional próprio e, em seguida, delega chamadas ao destino.

Por exemplo, podemos construir um BufferedInputStream para agrupar a entrada padrão do sistema:

InputStream bufferedIn = novo BufferedInputStream(System.in);

O BufferedInputStream lê antecipadamente e armazena em buffer uma certa quantidade de dados. Ele envolve uma camada adicional de funcionalidade em torno do fluxo subjacente.Figura 10-3mostra esse arranjo para um DataInputStream, que pode ler tipos de dados de nível superior, como primitivos Java e strings.

Figura 10-3. Fluxos em camadas

Como você pode ver no trecho de código anterior, o filtro BufferedInputStream é um tipo de InputStream. Como os fluxos de filtro são subclasses dos tipos de fluxo básicos, eles podem ser usados como argumentos para a construção de outros fluxos de filtro.

Isso permite que os fluxos de filtros sejam colocados uns sobre os outros para fornecer diferentes combinações de recursos. Por exemplo, poderíamos primeiro agrupar nosso System.in com um BufferedInputStream para nos beneficiar do buffer de nossa entrada e, em seguida, agrupar BufferedInputStream com um

DataInputStream para ler tipos de dados especiais com buffer.

Java fornece classes base para criar novos tipos de fluxos de filtro: FilterInputStream, FilterOutputStream, FilterReader e FilterWriter. Essas superclasses fornecem o mecanismo básico para um filtro, delegando todas as suas chamadas de método ao fluxo subjacente. Para criar seu próprio fluxo de filtro, você pode estender essas classes e substituir vários métodos para adicionar o processamento adicional necessário.

Fluxos de dados

DataInputStreame DataOutputStream são fluxos de filtro que permitem ler ou gravar strings (em oposição a caracteres individuais) e tipos de dados primitivos compostos de mais de um único byte. DataInputStream e DataOutputStream implementam as interfaces DataInput e DataOutput, respectivamente. Essas interfaces definem

métodos para ler ou escrever strings e todos os tipos primitivos Java, incluindo números e valores booleanos. DataOutputStream codifica esses valores de uma forma que pode ser lida corretamente em qualquer máquina e depois os grava em seu fluxo de bytes subjacente. DataInputStream coleta os dados codificados de seu fluxo de bytes subjacente e os decodifica em seus tipos e valores originais.

Você pode construir um DataInputStream a partir de um InputStream e então usar um método como readDouble() para ler um tipo de dados primitivo:

```
DataInputStream dis = new DataInputStream(System.in);

duplo d = dis.readDouble();
```

Este trecho envolve o fluxo de entrada padrão em um DataInputStream e o utiliza para ler um valor duplo. O

método readDouble() lê bytes do fluxo e constrói um double a partir deles. Os métodos DataInputStream esperam que os bytes dos tipos de dados numéricos estejam na ordem de bytes da rede, um padrão que especifica que os bytes de ordem superior de quaisquer valores multibyte são enviados primeiro (também conhecido como big-endian; consulte“Ordem de bytes”).

A classe DataOutputStream fornece métodos de gravação que correspondem aos métodos de leitura em DataInputStream. O complemento do nosso snippet de entrada é assim:

```
duplo d = 3,1415926;

DataOutputStream dos = new
DataOutputStream(System.out);

dos.writeDouble(d);
```

Aviso

DataOutputStreame DataInputStream funcionam com dados binários, não com texto legível por humanos. Normalmente, você usaria um DataInputStream para ler o conteúdo produzido por um DataOutputStream. Esses fluxos de filtro são perfeitos para trabalhar diretamente com arquivos de imagem.

Os métodos readUTF() e writeUTF() de DataInputStream e DataOutputStream leem e gravam uma string Java de caracteres Unicode usando a codificação de caracteres UTF-8. Como discutido emCapítulo 8, UTF-8 é uma codificação de caracteres Unicode compatível com ASCII que é amplamente usada. Nem todas as codificações preservam todos os caracteres Unicode, mas o UTF-8 preserva. Você também pode usar UTF-8

com fluxos de leitor e gravador, especificando-o como o nome da codificação.

Fluxos em buffer

As classes BufferedInputStream, BufferedOutputStream, BufferedReader e BufferedWriter adicionam um buffer de dados de um tamanho especificado ao caminho do fluxo. Um buffer pode aumentar a eficiência reduzindo o número de operações físicas de leitura ou gravação que correspondem às chamadas dos métodos read() ou write(), como visto emFigura 10-4.

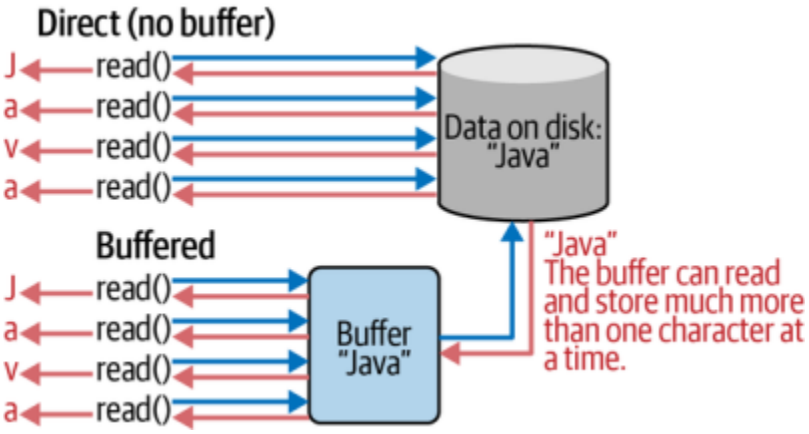


Figura 10-4. Lendo dados com e sem buffer

Você cria um fluxo em buffer com um fluxo de entrada ou saída apropriado e um tamanho de buffer. (Você também pode agrupar outro fluxo em torno de um fluxo em buffer para que ele se beneficie do buffer.) Aqui está um fluxo de entrada em buffer simples:

```
BufferedInputStream bis = new
BufferedInputStream(myInputStream, 32768);

// bis armazenará até 32K de dados de myInputStream
por vez
```

// podemos então ler bis a qualquer momento

byte b = bis.read();

Neste exemplo, especificamos um tamanho de buffer de 32 KB. Se deixarmos de lado o tamanho do buffer no construtor, Java criará um buffer de tamanho razoável para nós.

(Atualmente o padrão é 8 KB.) Em nossa primeira chamada para read(), bis tenta preencher todo o nosso buffer de 32 KB com dados, se estiverem disponíveis. Depois disso, as chamadas para read() recuperam dados do buffer, que é recarregado conforme necessário.

Um BufferedOutputStream funciona de maneira semelhante. Chamadas para write() armazenam os dados em um buffer; os dados são realmente gravados no fluxo subjacente somente quando o buffer fica cheio. Você também pode usar o método flush() para gravar o conteúdo de um BufferedOutputStream a qualquer momento. O

método flush() é na verdade um método da própria classe OutputStream. Ele permite que você tenha certeza de que todos os dados em qualquer fluxo subjacente foram salvos ou enviados.

As classes BufferedReader e BufferedWriter funcionam exatamente como suas contrapartes baseadas em bytes, exceto que operam em caracteres em vez de bytes.

PrintWriter e PrintStream

Outro wrapper útil é java.io.PrintWriter. Esta classe fornece um conjunto de métodos print() sobrecarregados que transformam seus argumentos em strings e os

enviam para fora do fluxo. Um conjunto complementar de métodos de conveniência println() acrescenta uma nova linha ao final das strings. Para saída de texto formatado, printf() e os métodos format() idênticos permitem que você escreva texto formatado no estilo C printf no fluxo.

Gravador de impressãoé um fluxo de caracteres incomum porque pode agrupar um OutputStream ou outro Writer. PrintWriter é o irmão mais velho mais capaz do legado

de fluxo de bytes PrintStream. Os streams System.out e System.err são objetos PrintStream, que você já viu ao longo deste livro:

```
System.out.print("Olá, mundo...\n");

System.out.println("Olá, mundo...");

System.out.printf("A resposta é %d\n", 17);

System.out.println(3.14);
```

Ao criar um objeto PrintWriter, você pode passar um valor booleano adicional para o construtor, especificando se ele deve “liberar automaticamente”. Se esse valor for verdadeiro, o PrintWriter executa automaticamente um flush() no OutputStream ou Writer subjacente cada vez que envia uma nova linha:

```
// O stream é liberado automaticamente após uma nova linha.

PrintWriter pw = new PrintWriter(myOutputStream, true);

pw.println("Olá!");
```

Quando você usa essa técnica com um fluxo de saída em buffer, ela atua como um terminal que gera dados linha por linha.

A outra grande vantagem que PrintStream e PrintWriter têm sobre os fluxos de caracteres regulares é que eles protegem você contra exceções lançadas pelos fluxos subjacentes. Ao contrário dos métodos em outras classes de fluxo, os métodos PrintWriter e PrintStream não lançam IOExceptions. Em vez disso, eles fornecem um método para verificar explicitamente erros, se necessário. Isso facilita muito a operação comum de impressão de texto. Você pode verificar se há erros com o método checkError():

```
System.out.println(reallyLongString);

if (System.out.checkError()) {

// ah ah

}
```

Esse recurso do PrintStream e do PrintWriter significa que geralmente você pode enviar texto para uma variedade de destinos sem agrupar todas as instruções de impressão em um bloco try. Mas ainda dá acesso a quaisquer erros que ocorram se você estiver escrevendo informações importantes e quiser garantir que nada deu errado.

A classe java.io.File

Um destino popular para saída impressa é um arquivo. A classe java.io.File encapsula o acesso a informações sobre um arquivo ou diretório. Você pode usar Arquivo para obter informações de atributos sobre um arquivo,

listar as entradas em um diretório e executar operações básicas do sistema de arquivos, como excluir um arquivo ou criar um novo diretório. Embora o objeto File lide com essas “meta” operações, ele não fornece a API para leitura e gravação de dados de arquivo; você precisará de fluxos de arquivos para isso.

Construtores de arquivos

Você pode criar uma instância de File a partir de um nome de caminho de String: Arquivo fooFile = new Arquivo(" /tmp/foo.txt");

Arquivo barDir = new Arquivo(" /tmp/bar");

Você também pode criar um arquivo com um caminho relativo que comece no diretório de trabalho atual da JVM:

Arquivo f = new Arquivo("foo");

Você pode determinar o diretório de trabalho atual lendo a propriedade user.dir na lista de propriedades do sistema:

System.getProperty("user.dir"); // por exemplo," /Usuários/pat"

Uma versão sobrecarregada do construtor File permite especificar o caminho do diretório e o nome do arquivo como objetos String separados:

Arquivo fooFile = new Arquivo(" /tmp", "foo.txt"); Com ainda outra variação, você pode especificar o diretório com um objeto File e o nome do arquivo com uma String:

Arquivo tmpDir = new Arquivo(" /tmp"); //Arquivo para diretório /tmp Arquivo fooFile = novo arquivo (tmpDir, "foo.txt");

Nenhum desses construtores File realmente cria um arquivo ou diretório, e não é um erro criar um objeto File para um arquivo inexistente. O objeto File é apenas um identificador para um arquivo ou diretório cujas propriedades você deseja ler, escrever ou testar. Por exemplo, você pode usar o método de instância exist() para saber se o arquivo ou diretório existe. Muitos aplicativos realizam esse teste antes de salvar em um arquivo, por exemplo. Se o arquivo escolhido não existir, viva! O

aplicativo pode salvar seus dados com segurança. Se o arquivo já existir, você geralmente receberá um aviso de substituição para garantir que deseja substituir o arquivo antigo.

Localização de caminho

Em Java, espera-se que os nomes de caminhos sigam as convenções do sistema de arquivos local. O sistema de arquivos do Windows usa raízes distintas (diretórios de nível superior) com letras de unidade (por exemplo, “C:”) e uma barra invertida (\) em vez da raiz única e separador de caminho de barra (/) que é usado no Linux e no macOS sistemas.

Java tenta compensar essa diferença. Por exemplo, em plataformas Windows, ele aceita caminhos com barras ou barras invertidas. No macOS e no Linux, entretanto, ele aceita apenas barras.

Sua melhor aposta é seguir as convenções de nome de arquivo do sistema de arquivos host. Se o seu aplicativo

tiver uma GUI que abre e salva arquivos a pedido do usuário, você poderá lidar com essa funcionalidade com a classe Swing JFileChooser. Esta classe encapsula uma caixa de diálogo gráfica de seleção de arquivo. Os métodos do JFileChooser cuidam dos recursos de nome de arquivo dependentes do sistema para você.

Porém, se seu aplicativo precisar lidar com arquivos em seu próprio nome, as coisas ficarão um pouco mais complicadas. A classe File contém algumas variáveis estáticas para facilitar esta tarefa. File.separator define uma String que especifica o separador de arquivos no host local (por exemplo, / em sistemas Unix e macOS, e \ em sistemas Windows); File.separatorChar fornece as mesmas informações que um char.

Você pode usar essas informações dependentes do sistema de diversas maneiras.

Provavelmente, a maneira mais simples de localizar nomes de caminhos é escolher uma convenção que você usa internamente, como a barra (/), e fazer uma substituição de String para substituir o caractere separador localizado:

```
// usaremos barra como padrão

String caminho = "mail/2023/junho";

caminho = caminho.replace('/', Arquivo.separatorChar);

Caixa de correio do arquivo = novo arquivo (caminho);
```

Alternativamente, você poderia trabalhar com os componentes de um nome de caminho e construir o nome de caminho local quando precisar:

```
String [] caminho = { "mail", "2004", "junho", "merle" };
StringBuffer sb = new StringBuffer(caminho[0]);

for (int i=1; i< caminho.comprimento; i++) {

sb.append(Arquivo.separador + caminho[i]);

}

Caixa de correio de arquivo = new File(sb.toString());
```

Observação

Lembre-se de que Java interpreta caracteres de barra invertida (\) no código-fonte como caracteres de escape quando usados em uma String. Para obter uma barra invertida literal, você precisa dobrar: \\.

Para lidar com o problema dos sistemas de arquivos com múltiplas “raízes” (por exemplo, C:\ no Windows), a classe File fornece o método estático listRoots(), que retorna uma matriz de objetos File correspondentes aos diretórios raiz do sistema de arquivos. Você pode tentar isso no jshell:

```
jshell> importar java.io.File;

// Em uma máquina Linux:

jshell> Arquivo.listRoots()

$2 ==> Arquivo[1] { / }

//No Windows:

jshell> Arquivo.listRoots()

$3 ==> Arquivo[2] { C:\, D:\ }
```

Novamente, em um aplicativo GUI, uma caixa de diálogo gráfica de seleção de arquivos geralmente protege você desse problema.

Operações de arquivo

Assim que tivermos um objeto File, podemos usá-lo para realizar uma série de operações padrão no arquivo ou diretório que ele representa. Vários métodos nos permitem fazer perguntas sobre o arquivo. Por exemplo, isFile() retorna verdadeiro se o arquivo representa um arquivo normal, enquanto isDirectory() retorna verdadeiro se for um diretório. isAbsolute() indica se o arquivo encapsula uma especificação de caminho absoluta ou relativa. Um caminho relativo é relativo ao diretório de trabalho do aplicativo. Um caminho absoluto é uma noção dependente do sistema que significa que o caminho não está vinculado ao diretório de trabalho ou à unidade atual. No Unix e no macOS, um caminho absoluto começa com uma barra:

/Users/pat/foo.txt. No Windows, é um caminho completo incluindo a letra da unidade: C:\Users\pat\foo.txt (e, novamente, pode estar em uma letra de unidade diferente do diretório de trabalho se houver várias unidades no sistema).

Os componentes do nome do caminho estão disponíveis através dos métodos getName(), getPath(), getAbsolutePath() e getParent(). O método getName() retorna uma String para o nome do arquivo sem nenhuma informação de diretório. Se o arquivo tiver uma especificação de caminho absoluto, getAbsolutePath() retornará esse caminho. Caso contrário, ele retorna o caminho relativo anexado ao diretório de trabalho atual

(tentando torná-lo um caminho absoluto). O método getParent() retorna o diretório pai do arquivo ou diretório.

A string retornada por getPath() ou getAbsolutePath() pode não seguir as mesmas convenções de caso do sistema de arquivos subjacente. Você pode recuperar a versão do próprio sistema de arquivos (ou “canônica”) do caminho do arquivo usando o método getCanonicalPath(). No Windows, por exemplo, você pode criar um objeto Arquivo cujo getAbsolutePath() seja C:\Autoexec.bat, mas cujo getCanonicalPath() seja C:\AUTOEXEC.BAT; ambos apontam para o mesmo arquivo. Isto é útil para comparar nomes de arquivos ou mostrá-los ao usuário.

Você pode obter ou definir o horário de modificação de um arquivo ou diretório com os métodos lastModified() e setLastModified(). O valor é um long que é o número de milissegundos desde a época (o nome da “primeira” data no Unix: 1º de janeiro de 1970, 00:00:00 GMT). Também podemos obter o tamanho do arquivo, em bytes, com length().

Aqui está um fragmento de código que imprime algumas informações sobre um arquivo:

Arquivo fooFile = new Arquivo(" /tmp/foo.txt"); Tipo de string = fooFile.isFile()? "Arquivo": "Diretório"; String nome = fooFile.getName();

comprimento longo = fooFile.length();

System.out.println(tipo + nome + ", " + len + " bytes "); Se o objeto File corresponder a um diretório, podemos listar os arquivos no diretório com o método list() ou o método listFiles():

Arquivo tmpDir = new Arquivo(" /tmp");

String [] nomes de arquivos = tmpDir.list();

Arquivo [] arquivos = tmpDir.listFiles();

lista()retorna uma matriz de objetos String que contém nomes de arquivos.

listFiles() retorna um array de objetos File. Observe que em nenhum dos casos é garantido que os arquivos estejam em qualquer tipo de ordem (alfabética, por exemplo). Você pode usar a API Collections para classificar strings em ordem alfabética, assim:

Lista lista = Arrays.asList(fileNames);

Coleções.sort(lista);

Se o arquivo se referir a um diretório inexistente, podemos criar o diretório com mkdir() ou mkdirs(). O método mkdir() cria no máximo um único nível de diretório, portanto, quaisquer diretórios intermediários no caminho já devem existir. mkdirs() cria todos os níveis de diretório necessários para criar o caminho completo da especificação do arquivo. Em ambos os casos, se o diretório não puder ser criado, o método retornará falso. Use renameTo() para renomear um arquivo ou diretório e delete() para excluir um arquivo ou diretório.

Embora você possa criar um diretório usando o objeto Arquivo, normalmente não usa Arquivo para criar um arquivo; isso normalmente é feito implicitamente quando você grava dados nele com um FileOutputStream ou FileWriter, como discutiremos em breve. A exceção é o método createNewFile(), que você pode usar para criar

um novo arquivo de comprimento zero no local do Arquivo.

A operação createNewFile() tem garantia de ser atômica3com relação a todas as outras criações de arquivos no sistema de arquivos. Java retorna um valor booleano de createNewFile() que informa se o arquivo foi criado ou não. Criar um novo arquivo dessa maneira é útil quando você também usa deleteOnExit(), que sinaliza o arquivo para ser removido automaticamente quando o Java VM for encerrado. Essa combinação permite proteger recursos ou criar um aplicativo que só pode ser executado em uma única instância por vez.

Outro método de criação de arquivo relacionado à própria classe File é o método estático createTempFile(), que cria um arquivo em um local especificado usando um nome exclusivo gerado automaticamente. Você normalmente usa createTempFile() em conjunto com deleteOnExit(). Os aplicativos de rede usam essa combinação com

frequência para criar arquivos de curta duração para armazenar solicitações ou criar respostas.

O método toURL() converte um caminho de arquivo em um arquivo: objeto URL. URLs são um tipo de abstração que permite apontar para qualquer tipo de objeto em qualquer lugar da rede. Converter uma referência de arquivo em uma URL pode ser útil para consistência com utilitários mais gerais que lidam com URLs. O NIO do Java, por exemplo, usa URLs para fazer referência a novos tipos de sistemas de arquivos que são implementados diretamente no código Java.

Tabela 10-1resume os métodos fornecidos pela classe File.

Tabela 10-1. Métodos de arquivo

Método

Tipo de retorno

Descrição

canExecute()

boleano

O arquivo é executável?

pode ler()

boleano

O arquivo (ou diretório) é

legível?

pode escrever()

boleano

O arquivo (ou diretório) é

gravável?

criarNovoArquivo()

boleano

Cria um novo arquivo.

createTempFile

Arquivo

Método estático para criar um

(Stringpfx, Stringsfx)

novo arquivo, com o prefixo e

o sufixo especificados, no

diretório de arquivo

temporário padrão.

excluir()

boleano

Exclui o arquivo (ou

diretório).

deleteOnExit()

Vazio

Ao sair, o sistema de tempo de

execução Java exclui o

arquivo.

existe()

boleano

O arquivo (ou diretório)

existe?

getAbsolutePath()

Corda

Retorna o caminho absoluto

do arquivo (ou diretório).

getCanonicalPath()

Corda

Retorna o caminho absoluto,

com correção de maiúsculas e

minúsculas e resolvido por

elemento relativo do arquivo

(ou diretório).

Método

Tipo de retorno

Descrição

getFreeSpace()

longo

Obtém o número de bytes de

espaço não alocado na

partição que contém esse

caminho ou 0 se o caminho

for inválido.

getNome()

Corda

Retorna o nome do arquivo

(ou diretório).

getParent()

Corda

Retorna o nome do diretório

pai do arquivo (ou diretório).

getPath()

Corda

Retorna o caminho do arquivo

(ou diretório). (Não deve ser

confundido com toPath().)

getTotalSpace()

longo

Obtém o tamanho da partição

que contém o caminho do

arquivo, em bytes, ou 0 se o

caminho for inválido.

getUseableSpace()

longo

Obtém o número de bytes de espaço não alocado acessível ao usuário na partição que contém esse caminho ou 0 se o caminho for inválido. Este método tenta levar em consideração as permissões de gravação do usuário.

éAbsoluto()

boleano

O nome do arquivo (ou nome do diretório) é absoluto?

éDiretório()

boleano

O item é um diretório?

isArquivo()

boleano

O item é um arquivo?

está escondido()

boleano

O item está oculto?

(Dependente do sistema.)

última modificação()

longo

Retorna a hora da última

modificação do arquivo (ou

diretório).

comprimento()

longo

Retorna o comprimento do

arquivo.

Método

Tipo de retorno

Descrição

lista()

Corda []

Retorna uma lista de arquivos

no diretório.

listaArquivos()

Arquivo[]

Retorna o conteúdo do diretório como uma matriz de objetos File.

listaRoots()

Arquivo[]

Retorna um array de sistemas de arquivos raiz, se houver (por exemplo, C:/, D:/).

mkdir()

boleano

Cria o diretório.

mkdirs()

boleano

Cria todos os diretórios no caminho.

renameTo(Arquivodestino)

boleano

Renomeia o arquivo (ou diretório).

setExecutável()

boleano

Define permissões de

execução para o arquivo.

setLastModified()

boleano

Define a hora da última

modificação do arquivo (ou

diretório).

setReadable()

boleano

Define permissões de leitura

para o arquivo.

setReadOnly()

boleano

Define o arquivo para status

somente leitura.

setWriteable()

boleano

Define as permissões de

gravação do arquivo.

toPath()

java.nio.file.Path Converta o arquivo em um

caminho de arquivo NIO. (Não

deve ser confundido com

getPath().)

paraURL()

java.net.URL

Gera um objeto URL para o

arquivo (ou diretório).

Fluxos de arquivos

Você provavelmente já está cansado de ouvir falar de arquivos — e ainda nem escrevemos um byte! Bem, agora começa a diversão. Java fornece dois fluxos fundamentais para leitura e gravação em arquivos: FileInputStream e

FileOutputStream. Esses fluxos fornecem a funcionalidade básica InputStream e OutputStream orientada a bytes que é aplicada à leitura e gravação de arquivos. Eles podem ser combinados com os fluxos de filtro descritos anteriormente para trabalhar com arquivos da mesma maneira que outras comunicações de fluxo.

Você pode criar um FileInputStream a partir de um nome de caminho String ou de um objeto File:

FileInputStream in = new FileInputStream(" /etc/motd");

Ao criar um FileInputStream, o sistema de tempo de execução Java tenta abrir o arquivo especificado. Assim, os construtores FileInputStream podem lançar uma FileNotFoundException se o arquivo especificado não existir ou uma IOException se ocorrer algum outro erro de E/S. Você deve capturar essas exceções em seu código.

Sempre que possível, é uma boa ideia adquirir o hábito de usar a construção try-with-resources para fechar arquivos automaticamente quando você terminar de usá-los: tente (FileInputStream fin = new FileInputStream(" /etc/motd")) {

// ....

// fin será fechado automaticamente se necessário

// ao sair da cláusula try.

}

Quando você cria o fluxo pela primeira vez, seu método available() e o método length() do objeto File devem retornar o mesmo valor.

Para ler caracteres de um arquivo como um Leitor, você pode agrupar um InputStreamReader em torno de um FileInputStream. Você também pode usar a classe FileReader, que é fornecida por conveniência. FileReader é apenas um

FileInputStream envolvido em um InputStreamReader com alguns padrões.

A classe a seguir, ListIt, é um pequeno utilitário que imprime o conteúdo de um arquivo ou diretório na saída padrão:

```
//arquivo: ch10/examples/ListIt.java

importar java.io.*;

classe ListaIt {

public static void main (String args[]) lança exceção {

Arquivo arquivo = new Arquivo(args[0]);

if (!file.exists() || !file.canRead()) {

System.out.println("Não é possível ler" + arquivo);

retornar;

}

if (arquivo.isDirectory()) {

String [] arquivos = arquivo.list();

para (arquivo de string: arquivos)

System.out.println(arquivo);

} outro {

tentar {

Leitor ir = new InputStreamReader(
```

```
novo FileInputStream(arquivo));

BufferedReader in = novo BufferedReader(ir);

Linha de corda;

while ((linha = in.readLine())! = nulo)

System.out.println(linha);

}

catch (FileNotFoundException e) {

System.out.println("Arquivo desaparecido");

}

}

}

}
```

Listarconstrói um objeto File a partir de seu primeiro argumento de linha de comando e testa o File para ver se ele existe e é legível. Se o arquivo for um diretório, ListIt exibirá os nomes dos arquivos no diretório. Caso contrário, ListIt lê e gera o arquivo, linha por linha. De uma chance! Você pode usar ListIt em ListIt.java?

Para gravar arquivos, você pode criar um FileOutputStream a partir de um nome de caminho String ou de um objeto File. Ao contrário de FileInputStream, entretanto, os construtores FileOutputStream não lançam uma FileNotFoundException. Se o arquivo especificado não existir, o FileOutputStream criará o arquivo. Os construtores FileOutputStream podem lançar uma

IOException se ocorrer algum outro erro de E/S, portanto, você ainda precisará lidar com essa exceção.

Se o arquivo especificado existir, FileOutputStream o abrirá para gravação. Quando você chama posteriormente o método write(), os novos dados substituem o conteúdo atual do arquivo. Se precisar anexar dados a um arquivo existente, você pode usar uma forma do construtor que aceita um sinalizador de acréscimo booleano: FileOutputStream fooOut =

novo FileOutputStream(fooFile); // sobrescreve fooFile

FileOutputStream pwdOut =

novo FileOutputStream(" /etc/passwd", verdadeiro); // acrescenta Outra maneira de anexar dados a arquivos é com RandomAccessFile, que

discutiremos em breve.

Assim como na leitura, para escrever caracteres (em vez de bytes) em um arquivo, você pode agrupar um OutputStreamWriter em torno de um FileOutputStream. Se quiser usar o esquema de codificação de caracteres padrão, você poderá usar a classe FileWriter, que é fornecida como uma conveniência.

O trecho a seguir lê uma linha de dados da entrada padrão e a grava no arquivo

/tmp/foo.txt:

String s = novo BufferedReader(

novo InputStreamReader(System.in)).readLine();

Arquivo de saída = new Arquivo(" /tmp/foo.txt");

FileWriter fw = novo FileWriter (fora);

PrintWriter pw = novo PrintWriter(fw);

pw.println(s);

close();

Observe como agrupamos o FileWriter em um PrintWriter para facilitar a gravação dos dados. Além disso, para ser um bom cidadão do sistema de arquivos, chame o método close() quando terminar. Aqui, fechar o PrintWriter fecha o Writer subjacente para nós.

Arquivo de acesso aleatório

A classe java.io.RandomAccessFile fornece a capacidade de ler e gravar dados em qualquer local de um arquivo. RandomAccessFile implementa as interfaces DataInput e DataOutput, para que você possa usá-lo para ler e escrever strings e tipos primitivos em qualquer lugar do arquivo, como se fosse um DataInputStream e

DataOutputStream. No entanto, como a classe fornece acesso aleatório, em vez de sequencial, aos dados do arquivo, ela não é uma subclasse de InputStream ou OutputStream.

Você pode criar um RandomAccessFile a partir de um nome de caminho String ou de um objeto File. O construtor também usa um segundo argumento String que especifica o modo do arquivo. Use a string "r" para um arquivo somente leitura ou "rw" para um arquivo de leitura/gravação:

tentar {

Usuários RandomAccessFile = new RandomAccessFile("Usuários", "rw")

} catch (IOException e) {... }

Ao criar um RandomAccessFile no modo somente leitura, Java tenta abrir o arquivo especificado. Se o arquivo não existir, RandomAccessFile lança uma IOException. Se, no entanto, você estiver criando um RandomAccessFile no modo leitura/gravação, o objeto criará o arquivo se ele não existir. O construtor ainda pode lançar uma IOException se ocorrer algum outro erro de E/S, portanto, você ainda precisará tratar essa exceção.

Depois de criar um RandomAccessFile, você pode chamar qualquer um dos métodos normais de leitura e gravação, assim como faria com DataInputStream ou DataOutputStream. Se você tentar gravar em um arquivo somente leitura, o método write lançará uma IOException.

O que torna um RandomAccessFile especial é o método seek(). Este método recebe um valor longo e o utiliza para definir o local de leitura e gravação no arquivo. Você pode usar o método getFilePointer() para obter a localização atual. Se você precisar anexar dados ao final do arquivo, use length() para determinar esse local e, em seguida, seek() para ele. Você pode escrever ou procurar além do final de um arquivo, mas não

pode ler além do final de um arquivo. O método read() lança uma EOFException se você tentar fazer isso.

Aqui está um exemplo de gravação de dados para um banco de dados simples: usuários.seek (userNum *

RECORDSIZE);

usuários.writeUTF(nomedeusuário);

usuários.writeInt(userID);

Neste trecho, presumimos que o comprimento da string para userName, junto com quaisquer dados que venham depois dele, se ajuste ao tamanho do registro especificado.

A nova API de arquivo de E/S

Agora vamos voltar nossa atenção da API de arquivo Java “clássica” original para a API de arquivo NIO. Como mencionamos anteriormente, a API de arquivo NIO pode ser considerada um substituto ou um complemento da API clássica. A nova API move o Java em direção a um desempenho mais alto e um estilo de E/S mais flexível, suportando canais selecionáveis e interrompíveis de forma assíncrona. (Mais sobre como selecionar e usar canais em breve.) Ao trabalhar com arquivos, o ponto forte da nova API é fornecer uma abstração mais completa do sistema de arquivos em Java.

Além de melhor suporte para tipos de sistemas de arquivos existentes e reais –

incluindo a nova e bem-vinda capacidade de copiar e mover arquivos, gerenciar links e obter atributos detalhados de arquivos, como proprietários e permissões – o NIO

permite implementar tipos inteiramente novos de sistemas de arquivos diretamente.

em Java. O melhor exemplo disso é o provedor de sistema de arquivos ZIP. Você pode

“montar” um arquivo ZIP como um sistema de arquivos. Você pode trabalhar com os arquivos do arquivo diretamente usando as APIs padrão, como qualquer outro sistema de arquivos.

O pacote NIO File também fornece alguns utilitários que teriam poupado aos desenvolvedores Java muitos códigos repetidos ao longo dos anos, incluindo monitoramento de alterações na árvore de diretórios, travessia do sistema de arquivos, “globbing” de nomes de arquivos (o jargão para usar curingas em nomes de arquivos) e métodos convenientes para leia arquivos inteiros diretamente na memória.

Abordaremos a API básica do arquivo NIO nesta seção e retornaremos ao tópico de buffers e canais no final do capítulo. Em particular, falaremos sobre ByteChannels e FileChannel, que você pode considerar como fluxos alternativos orientados a buffer para leitura e gravação de arquivos e outros tipos de dados.

Sistema de arquivos e caminho

Existem três participantes principais no pacote java.nio.file:

Sistema de arquivo

Um mecanismo de armazenamento subjacente e serve como uma fábrica4para objetos Path.

Sistemas de arquivos

Uma fábrica para objetos FileSystem.

Caminho

A localização de um arquivo ou diretório dentro do sistema de arquivos.

arquivos

Uma classe de utilitário que contém um rico conjunto de métodos estáticos para manipular objetos Path para executar todas as operações básicas de arquivo análogas à API clássica.

A classe FileSystems (plural) é nosso ponto de partida. Vamos criar alguns sistemas de arquivos:

// O sistema de arquivos padrão do computador host

FileSystem fs = FileSystems.getDefault();

// Um sistema de arquivos personalizado para arquivos ZIP, sem propriedad es especiais

Map<String,String> props = new HashMap<>();

URI zipURI = URI.create("jar:file:/Users/pat/tmp/MyArchive.zip"); FileSystem zipfs = FileSystems.newFileSystem(zipURI, adereços);

Conforme mostrado neste trecho, solicitamos que o sistema de arquivos padrão manipule arquivos no ambiente do computador host. Também usamos a classe FileSystems para construir outro FileSystem usando um identificador de recurso uniforme (ou URI, um identificador especial semelhante a uma URL) que faz referência a um tipo de sistema de arquivos personalizado. Usamos jar:file como nosso protocolo URI

para indicar que estamos trabalhando com um arquivo JAR ou ZIP.

Sistema de arquivoimplementa Closeable e, quando um FileSystem é fechado, todos os canais de arquivos abertos e outros objetos de streaming associados a ele também são fechados. A tentativa de ler ou gravar nesses canais gerará uma exceção nesse ponto. Observe que o sistema de arquivos padrão (associado ao computador host) não pode ser fechado.

Depois de ter um FileSystem, você pode usá-lo como uma fábrica para objetos Path que representam arquivos ou diretórios. Você pode obter um Path usando uma representação de string, assim como a classe File clássica. Posteriormente, você pode usar esse objeto Path com métodos do utilitário Arquivos para criar, ler, gravar ou excluir o item:

Caminho fooPath = fs.getPath(" /tmp/foo.txt");

OutputStream out = Files.newOutputStream(fooPath);

Este exemplo abre um OutputStream para gravar no arquivo foo.txt. Por padrão, se o arquivo não existir, ele será criado e, se existir, será truncado (definido como comprimento zero) antes que novos dados sejam gravados — mas você pode alterar esses resultados usando opções. Falaremos mais sobre os métodos Files na próxima seção.

A classe Path implementa a interface java.lang.Iterable, que pode ser usada para iterar por meio de seus componentes de caminho literal, como tmp e foo.txt separados por barra no trecho anterior. (Se você quiser percorrer o caminho para encontrar outros arquivos ou diretórios, você pode estar mais interessado em

DirectoryStream e FileVisitor que discutiremos mais tarde.) Path também implementa a interface java.nio.file.Watchable, que permite que ele ser monitorado quanto a mudanças.

Caminhopossui métodos convenientes para resolver caminhos relativos a um arquivo ou diretório:

Caminho patPath = fs.getPath(" /Usuário/pat/");

Caminho patTmp = patPath.resolve("tmp"); // " /Usuário/pat/tmp"

// Igual ao acima, usando um Path

Caminho tmpPath = fs.getPath("tmp");

Caminho patTmp = patPath.resolve(tmpPath); // " /Usuário/pat/tmp"

// Resolver um determinado caminho absoluto em relação a qualquer caminho apenas produz um determinado caminho

Caminho absPath = patPath.resolve(" /tmp"); // " /tmp"

// Resolve o irmão para Pat (mesmo pai)

Caminho danPath = patPath.resolveSibling("dan"); // " /Usuários/dan"

Neste trecho, mostramos os métodos Path resolve() e resolveSibling() usados para localizar arquivos ou diretórios relativos a um determinado objeto Path. O método resolve() geralmente é usado para anexar um caminho relativo a um caminho existente que representa um diretório. Se o argumento fornecido para o método

resolve() for um caminho absoluto, ele apenas produzirá o caminho absoluto (ele age como o comando cd do Unix ou DOS). O método resolveSibling() funciona da mesma maneira, mas é relativo ao pai do Path alvo; este método é útil para descrever o alvo de uma operação move().

Caminho para arquivos clássicos e vice-versa

Para unir as APIs clássica e nova, os métodos toPath() e toFile() correspondentes foram fornecidos em java.io.File e java.nio.file.Path, respectivamente, para converter para o outro formato. Obviamente, os únicos tipos de caminhos que podem ser produzidos a partir de Arquivo são caminhos que representam arquivos e diretórios no sistema de arquivos host padrão:

Caminho tmpPath = fs.getPath(" /tmp");

Arquivo arquivo = tmpPath.toFile();

Arquivo tmpFile = new Arquivo(" /tmp");

Caminho caminho = tmpFile.toPath();

Operações de arquivo NIO

Assim que tivermos um caminho, podemos operar nele com métodos estáticos do utilitário Arquivos para criar o caminho como um arquivo ou diretório, ler e gravar nele e interrogar e definir suas propriedades. Listaremos a maior parte deles e discutiremos alguns dos mais importantes à medida que prosseguirmos.

Tabela 10-2resume esses métodos da classe java.nio.file.Files. Como seria de esperar, como a classe Files lida com todos os tipos de operações de arquivo, ela contém um grande número de métodos. Para tornar a

tabela mais legível, eliminamos formas sobrecarregadas do mesmo método (aquelas que recebem diferentes tipos de argumentos) e agrupamos tipos de métodos correspondentes e relacionados.

Tabela 10-2. Métodos de arquivos NIO

Método

Tipo de retorno

Descrição

cópia de()

longo ou caminho

Copie um fluxo para um

caminho de arquivo,

caminho de arquivo para

fluxo ou caminho para

caminho. Retorna o número

de bytes copiados ou o

caminho de destino. Um

arquivo de destino pode

opcionalmente ser

substituído se existir (o

padrão é falhar se o destino

existir). Copiar um diretório resulta em um diretório vazio no destino (o conteúdo não é copiado). Copiar um link simbólico copia os dados do arquivo vinculado (produzindo uma cópia normal do arquivo).

Criar diretório(),

Caminho

Crie um único diretório ou

criarDiretórios()

todos os diretórios em um caminho especificado. createDirectory() lança uma exceção se o diretório já existir. createDirectories() irá ignorar os diretórios

Método

Tipo de retorno

Descrição

existentes e criar apenas

conforme necessário.

criarArquivo()

Caminho

Crie um arquivo vazio. A

operação é atômica e só terá

sucesso se o arquivo não

existir. (Esta propriedade

pode ser usada para criar

arquivos de sinalização para

proteger recursos, etc.)

criarLink(),

booleano ou caminho

Crie um link físico ou

createSymbolicLink(),

simbólico, teste para ver se

isSymbolicLink(),

um arquivo é um link

readSymbolicLink(),

simbólico ou leia o arquivo

createLink()

de destino apontado pelo

link simbólico. Links

simbólicos são arquivos que

fazem referência a outros

arquivos. Links regulares

(“hard”) são espelhos de

baixo nível de um arquivo

onde dois nomes de

arquivos apontam para os

mesmos dados subjacentes.

Se você não sabe qual usar,

use um link simbólico.

createTempDirectory(),

Caminho

Crie um diretório ou arquivo

criarTempFile()

temporário, garantido e com

nome exclusivo com o

prefixo especificado.

Opcionalmente, coloque-o

no diretório temporário

padrão do sistema.

excluir(), excluirIfExists() vazio

Exclua um arquivo ou um

diretório vazio.

deleteIfExists() não lançará

uma exceção se o arquivo

não existir.

existe(), não existe()

boleano

Determine se o arquivo

existe (notExists()

simplesmente retorna o

oposto). Opcionalmente,

Método

Tipo de retorno

Descrição

especifique se os links

devem ser seguidos (por padrão, eles são).

getAtributo(), set

Objeto, Mapa ou

Obtenha ou defina atributos

Attribute(), getFile

FileAttributeView

de arquivo específicos do

AttributeView(),

sistema de arquivos, como

readAttributes()

tempos de acesso e atualização, permissões detalhadas e informações do proprietário usando nomes específicos da implementação.

getFileStore()

Armazenamento de

Obtenha um objeto FileStore

arquivos

que representa o

dispositivo, volume ou outro

tipo de partição do sistema

de arquivos no qual reside o

caminho.

getLastModifiedTime(),

ArquivoTimeou

Obtenha ou defina a hora da

setLastModifiedTime()

caminho

última modificação de um

arquivo ou diretório.

getProprietário(),

UsuárioPrincipal

Obtenha ou defina um

setProprietário()

objeto UserPrincipal que

representa o proprietário do

arquivo. Use toString() ou

getName() para obter uma representação em string do nome de usuário.

getPosixFile

Definirou caminho

Obtenha ou defina as Permissõessões(), permissões completas de setPosixFilePermissions() leitura e gravação do estilo POSIX user-group-other para o caminho como um conjunto de valores de enum PosixFilePermission.

éDiretório(),

boleano

Teste recursos do arquivo, isExecutable(), isHidden(), como se o caminho é um isReadable(),

diretório e outros atributos básicos.

Método

Tipo de retorno

Descrição

isRegularFile(), isWriteable()

isSameFile()

boleano

Teste para ver se os dois caminhos fazem referência ao mesmo arquivo (o que pode ser verdade mesmo que os caminhos não sejam idênticos).

mover()

Caminho

Mova um arquivo ou diretório renomeando-o ou copiando-o, especificando

opcionalmente se deseja substituir qualquer destino existente. Renomear será usado a menos que uma cópia seja necessária para mover um arquivo entre armazenamentos de arquivos ou sistemas de arquivos. Os diretórios podem ser movidos usando este método somente se a renomeação simples for possível ou se o diretório estiver vazio. Se uma movimentação de diretório exigir a cópia de arquivos entre armazenamentos de arquivos ou sistemas de arquivos, o método lançará uma IOException. (Neste

caso, você mesmo deve
copiar os arquivos. Consulte
walkFileTree().)
novoBufferedReader(),
Leitor Bufferedou
Abra um arquivo para
novoBufferedWriter()
BufferedWriter
leitura por meio de um
BufferedReader ou crie e
abra um arquivo para
gravação por meio de um
BufferedWriter. Em ambos
os casos, uma codificação de
caracteres é especificada.
Método
Tipo de retorno
Descrição
novoByteChannel()
SeekableByteChannel Crie um novo arquivo ou

abra um arquivo existente como um canal de bytes pesquisável. (Veja a discussão completa sobre NIO posteriormente neste capítulo.) Considere usar FileChannelopen() como alternativa.

novoDirectoryStream() DirectoryStream

Retorne um DirectoryStream para iterar em uma hierarquia de diretórios. Opcionalmente, forneça um padrão glob ou objeto de filtro para corresponder aos arquivos.

novoInputStream(), Fluxo de entradaou

Abra um arquivo para

novoOutputStream()

OutputStream

leitura por meio de um InputStream ou crie e abra um arquivo para gravação por meio de um OutputStream. Opcionalmente, especifique o truncamento de arquivo para o fluxo de saída; se for sobrescrever, o padrão é truncar o arquivo existente.

probeContentType()

Corda

Retorne o tipo MIME do arquivo se ele puder ser determinado pelos serviços FileTypeDetector instalados ou nulo se for desconhecido.

lerTodosBytes(),

byte[] ou

Leia todos os dados do

leiaTodasLinhas()

Lista<String>

arquivo como um byte [] ou

todos os caracteres como

uma lista de strings usando

uma codificação de

caracteres especificada.

Método

Tipo de retorno

Descrição

tamanho()

longo

Obtenha o tamanho, em

bytes, do arquivo no

caminho especificado.

walkFileTree()

Caminho

Aplique um FileVisitor à

árvore de diretórios especificada, especificando opcionalmente se os links devem ser seguidos e uma profundidade máxima de travessia.

escrever()

Caminho

Grave uma matriz de bytes ou uma coleção de strings (com uma codificação de caracteres especificada) no arquivo no caminho especificado e feche o arquivo, especificando opcionalmente o comportamento de acréscimo e truncamento. O padrão é truncar e gravar os dados.

Com esses métodos, podemos buscar fluxos de entrada ou saída, ou leitores e gravadores armazenados em buffer, para um determinado arquivo. Também podemos criar caminhos como arquivos e diretórios e iterar por meio de hierarquias de arquivos. Discutiremos as operações de diretório na próxima seção.

Como lembrete, os métodos resolve() e resolveSibling() de Path são úteis para construir alvos para as operações copy() e move():

// Move o arquivo /tmp/foo.txt para /tmp/bar.txt

Caminho foo = fs.getPath(" /tmp/foo.txt");

Arquivos.move(foo, foo.resolveSibling("bar.txt"));

Para ler e escrever rapidamente o conteúdo de arquivos sem streaming, podemos usar os vários métodos readAll... e escrever que movem matrizes de bytes ou strings para dentro e para fora dos arquivos em uma única operação:

// Ler e escrever uma coleção de String (por exemplo, linhas de texto) Conjunto de caracteres asciiCharset = Charset.forName("US-ASCII"); List<String> csvData = Files.readAllLines(csvPath, asciiCharset); Arquivos.write(newCSVPath, csvData, asciiCharset);

//Lê e escreve bytes

byte [] dados = Files.readAllBytes(dataPath);

Arquivos.write(newDataPath, dados);

Eles são convenientes para arquivos que cabem facilmente na memória.

O Pacote NIO

Vamos retornar ao pacote java.nio e encerrar nossa discussão sobre o núcleo de E/S

Java. Um aspecto do NIO é simplesmente atualizar e aprimorar os recursos do pacote java.io clássico. Grande parte da funcionalidade geral do NIO realmente se sobrepõe às APIs existentes. No entanto, o NIO foi introduzido pela primeira vez para resolver problemas específicos de escalabilidade para grandes sistemas, especialmente em aplicações em rede. As seções a seguir descrevem os elementos básicos do NIO.

E/S assíncrona

A maior parte da necessidade do pacote NIO foi motivada pelo desejo de adicionar E/S

selecionável e sem bloqueio ao Java. Antes do NIO, a maioria das operações de leitura e gravação em Java eram vinculadas a threads e eram forçadas a bloquear por períodos de tempo imprevisíveis. Embora certas APIs, como Sockets (que veremos

em"Tomadas"), forneceu meios específicos para limitar quanto tempo uma chamada de E/S poderia levar, esta foi uma solução alternativa para compensar a falta de um mecanismo mais geral. Em muitas linguagens, mesmo aquelas sem threading, a E/S

ainda pode ser feita de forma eficiente, configurando os fluxos de E/S para um modo sem bloqueio e testando-os quanto à sua prontidão para enviar ou receber dados. Em um modo sem bloqueio, uma leitura ou gravação realiza apenas o trabalho que pode ser feito imediatamente –

preenchendo ou esvaziando um buffer e depois retornando.

Combinado com a capacidade de testar a prontidão, isso permite que um aplicativo de thread único atenda continuamente muitos canais com eficiência. O thread principal

"seleciona" um canal que está pronto, trabalha com ele até bloquear e depois passa para outro. Em um sistema de processador único, isso é fundamentalmente equivalente ao uso de vários threads.

Além de E/S não bloqueante e selecionável, o pacote NIO permite fechar e interromper operações de E/S de forma assíncrona. Como discutido emCapítulo 9,

antes do NIO não havia uma maneira confiável de parar ou ativar um thread bloqueado em uma operação de E/S. Com o NIO, threads bloqueados em operações de E/S sempre são ativados quando interrompidos ou quando outro thread fecha o canal.

Além disso, se você interromper um thread enquanto ele estiver bloqueado em uma operação NIO, seu canal será fechado automaticamente. (Fechar o canal porque o thread foi interrompido pode parecer muito forte, mas geralmente é a coisa certa a fazer. Deixá-lo aberto pode resultar em comportamento inesperado ou sujeitar o canal a manipulação indesejada.)

Desempenho

A E/S de canal é projetada em torno do conceito de buffers, que são uma forma sofisticada de matriz, adaptada para tarefas de comunicação. O pacote NIO suporta o conceito de buffers diretos – buffers que

mantêm sua memória fora do Java VM no sistema operacional host. Como todas as operações reais de E/S precisam, em última análise, funcionar com o sistema operacional host, mantendo o espaço de buffer nele, o uso de buffers diretos pode tornar muitas operações mais eficientes. Os dados movimentados entre dois terminais externos podem ser transferidos sem primeiro copiá-los para Java e devolvê-los.

Arquivos mapeados e bloqueados

NIO fornece dois recursos relacionados a arquivos de uso geral não encontrados em java.io: arquivos mapeados em memória e bloqueio de arquivos. Um arquivo mapeado na memória se comporta como se todo o seu conteúdo estivesse em uma matriz na memória, e não em um disco. Arquivos mapeados na memória estão além do escopo deste capítulo, mas se você trabalha com muitos dados e ocasionalmente precisa de acesso de leitura/gravação muito rápido, procure o arquivoDocumentação do

MappedByteBufferon-line.

O bloqueio de arquivos suporta bloqueios compartilhados e exclusivos em regiões de arquivos – útil para acesso simultâneo por vários aplicativos. Veremos o bloqueio de arquivos“Bloqueio de arquivo”.

Canais

Enquanto java.io lida com streams, java.nio trabalha com canais. Um canal é um ponto final para comunicação. Embora na prática os canais sejam semelhantes aos fluxos, a noção subjacente de canal é simultaneamente mais abstrata e mais primitiva.

Enquanto os fluxos em java.io são definidos em termos de entrada ou saída com métodos para ler e escrever bytes, a interface básica do canal não diz nada sobre como as comunicações acontecem. Simplesmente tem a noção de ser aberto ou fechado, suportado pelos métodos isOpen() e close(). Implementações de canais para arquivos, soquetes de rede ou dispositivos arbitrários adicionam seus próprios métodos para operações, como leitura, gravação ou transferência de dados. NIO fornece os seguintes canais:

•

Canal de arquivo

•

Pipe.SinkChannel, Pipe.SourceChannel

•

SocketChannel, ServerSocketChannel, DatagramChannel

Abordaremos FileChannel e seu primo assíncrono, AsynchronousFileChannel,

em“FileCanal”. (A versão assíncrona essencialmente armazena em buffer todas as suas operações por meio de um pool de threads e relata os resultados por meio de uma API assíncrona.) Os canais Pipe são simplesmente os equivalentes de canal dos recursos

Java.io Pipe. Os canais de soquete e datagrama atuam no mundo das redes Java, que veremos emCapítulo 13. Os canais relacionados à rede também possuem versões assíncronas: AsynchronousSocketChannel,

AsynchronousServerSocketChannel e AsynchronousDatagramChannel.

Todos esses canais básicos implementam a interface ByteChannel, projetada para canais que possuem métodos de leitura e gravação como fluxos de E/S. ByteChannels, entretanto, lê e grava ByteBuffers, em oposição a matrizes de bytes simples.

Além dessas implementações de canal, você pode conectar canais com fluxos de E/S

java.io e leitores e gravadores para interoperabilidade. No entanto, se você combinar esses recursos, poderá não obter todos os benefícios e desempenho que o NIO oferece.

Buffers

A maioria dos utilitários dos pacotes java.io e java.net operam em matrizes de bytes.

As ferramentas correspondentes do pacote NIO são construídas em torno de ByteBuffers (com um buffer baseado em caracteres, CharBuffer, para texto). Matrizes de bytes são simples, então por que os buffers são necessários? Eles servem a vários propósitos:

•

Eles formalizam os padrões de uso para dados em buffer, fornecem coisas como buffers somente leitura e controlam posições e limites de

leitura/gravação dentro de um grande espaço de buffer. Eles também fornecem um recurso de

marcação/redefinição como o de java.io.BufferedInputStream.

•

Eles fornecem APIs adicionais para trabalhar com dados brutos que

representam tipos primitivos. Você pode criar buffers que “visualizam” seus dados de bytes como uma série de primitivos maiores, como shorts, ints ou floats. O tipo mais geral de buffer de dados, ByteBuffer, inclui métodos que permitem ler e escrever todos os tipos primitivos, assim como

DataInputStream e DataOutputStream fazem para streams.

•

Eles abstraem o armazenamento subjacente dos dados, permitindo que o Java otimize o rendimento. Especificamente, os buffers podem ser alocados como buffers diretos que usam buffers nativos do sistema operacional host em vez de arrays na memória do Java. As instalações do Canal NIO que trabalham com buffers podem reconhecer buffers diretos automaticamente e tentar otimizar suas interações com eles. Por exemplo, uma leitura de um canal de arquivo em uma matriz de bytes Java normalmente requer que Java copie os dados para leitura do sistema operacional host para a memória Java. Com um buffer direto, os dados podem permanecer no sistema operacional host, fora do espaço de memória normal do Java, até e a menos que sejam necessários.

Operações de buffer

A classe base java.nio.Buffer é algo como um array com estado. Ele não especifica que tipo de elementos ele contém (ou seja, cabe aos subtipos decidirem), mas define a

funcionalidade que é comum a todos os buffers de dados. Um Buffer tem um tamanho fixo, denominado capacidade. Embora todos os Buffers padrão forneçam “acesso aleatório” ao seu conteúdo, um Buffer geralmente espera ser lido e escrito sequencialmente, então os Buffers mantêm a noção de uma posição onde o próximo elemento é lido ou escrito. Além da posição, um Buffer pode manter duas outras informações de estado: um limite, que normalmente denota os dados disponíveis no modo de leitura e a capacidade do arquivo no modo de gravação, e uma marca, que pode ser usada para lembrar um valor anterior. posição para recall futuro.

Implementações do Buffer adicionam métodos get e put específicos e digitados que leem e gravam o conteúdo do buffer. Por exemplo, ByteBuffer é um buffer de bytes e possui métodos get() e put() que leem e gravam bytes e matrizes de bytes (junto com muitos outros métodos úteis que discutiremos mais tarde). Sair e colocar no Buffer altera o marcador de posição, de modo que o Buffer monitora seu conteúdo como um fluxo. A tentativa de ler ou gravar além do marcador de limite gera um BufferUnderflowException ou BufferOverflowException, respectivamente.

Os valores de marca, posição, limite e capacidade obedecem sempre à seguinte fórmula:

marca <= posição <= limite <= capacidade

A posição de leitura e escrita do Buffer é sempre entre a marca, que serve como limite inferior, e o limite, que serve como limite superior. A capacidade representa a extensão física do espaço do buffer.

Você pode definir os marcadores de posição e limite explicitamente com os métodos position() e limit(). Vários métodos de conveniência são fornecidos para padrões de uso comuns. O método reset() define a posição de volta à marca. Se nenhuma marca tiver sido definida, uma InvalidMarkException será lançada. O método clear() redefine a posição para 0 e limita a capacidade, preparando o buffer para novos dados (a marca é descartada). Observe que o método clear() na verdade não faz nada com os dados no buffer; simplesmente altera os marcadores de posição.

O método flip() é usado para o padrão comum de gravar dados no buffer e depois lê-los novamente. flip torna a posição atual o limite e, em seguida, redefine a posição atual para 0 (qualquer marca é descartada), o que evita a necessidade de controlar quantos dados foram lidos. Outro método, rewind(), simplesmente redefine a posição para 0, deixando o limite de lado. Você pode usá-lo para gravar dados do mesmo tamanho novamente. Aqui está um trecho de código que usa esses métodos para ler dados de um canal e gravá-los em dois canais:

```
Buff de ByteBuffer = ...

while (inChannel.read(buff) > 0) { // posição = ?

buff.flip(); // limite = posição; posição = 0;

outChannel.write(buff);

buff.rewind(); // posição = 0
```

```
outChannel2.write(buff);

buff.claro(); // posição = 0; limite = capacidade

}
```

Isso pode ser confuso na primeira vez que você olha para isso, porque aqui, a leitura do Canal é na verdade uma gravação no Buffer e vice-versa. Como este exemplo grava todos os dados disponíveis até o limite, flip() ou rewind() têm o mesmo efeito neste caso.

Tipos de buffer

As diversas implementações de buffer adicionam métodos get e put para leitura e gravação de tipos de dados específicos. Cada um dos tipos primitivos Java possui um tipo de buffer associado: ByteBuffer, CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer e DoubleBuffer. Cada um fornece métodos get e put para ler e escrever seu tipo e matrizes de seu tipo. Destes, ByteBuffer é o mais flexível. Por ter a “melhor granulação” de todos os buffers, recebeu um complemento completo de métodos get e put para ler e gravar todos os outros tipos de dados, bem como bytes. Aqui estão alguns métodos ByteBuffer:

```
byte obtido()

char getChar()

getShort curto()

int getInt()

longo getLong()
```

float getFloat()

duplo getDouble()

vazio colocado (byte b)

void put(ByteBuffer src)

void put(byte[] src, deslocamento interno, comprimento interno)

void put(byte[] src)

void putChar(valor do caractere)

void putShort(valor curto)

void putInt(valor int)

void putLong (valor longo)

void putFloat(valor flutuante)

void putDouble(valor duplo)

Todos os buffers padrão também suportam acesso aleatório. Para cada um dos métodos de ByteBuffer mencionados acima, um formulário adicional leva um índice, por exemplo:

getLong (índice interno)

putLong(índice int, valor longo)

Mas isso não é tudo! ByteBuffer também pode fornecer “visualizações” de si mesmo como qualquer um dos tipos de granulação grossa. Por exemplo, você pode buscar uma visualização ShortBuffer de um ByteBuffer com o

método asShortBuffer(). A visualização ShortBuffer é apoiada pelo ByteBuffer, o que significa que eles funcionam nos mesmos dados e as alterações em um deles afetam o outro. A extensão do buffer

de visualização começa na posição atual do ByteBuffer e sua capacidade é função do número restante de bytes, dividido pelo tamanho do novo tipo. (Por exemplo, shorts consomem dois bytes cada, floats quatro e longs e doubles levam oito.) Os buffers de visualização são convenientes para ler e gravar grandes blocos de um tipo contíguo dentro de um ByteBuffer.

CharBuffers também são interessantes, principalmente por causa de sua integração com Strings. Tanto CharBuffers quanto Strings implementam a interface java.lang.CharSequence. Esta é a interface que fornece os métodos padrão charAt() e length(). Muitas outras partes do Java (como o pacote java.util.regex) permitem que você use um CharBuffer ou uma String de forma intercambiável. Nesse caso, o CharBuffer atua como uma String modificável com posições iniciais e finais lógicas configuráveis pelo usuário.

Ordem de bytes

Como estamos falando de leitura e escrita de tipos maiores que um byte, surge a pergunta: em que ordem os bytes de valores multibyte (como shorts e ints) são escritos? Existem dois campos neste mundo: big-endian e little-endian.5Big-endian significa que os bytes mais significativos vêm primeiro; little-endian é o inverso. Se você estiver gravando dados binários para consumo por algum aplicativo nativo, isso é importante. Computadores compatíveis com Intel usam little endian, e muitas estações de trabalho que executam Unix usam big

endian. A classe ByteOrder encapsula a escolha. Você pode especificar a ordem dos bytes a ser usada com o método ByteBuffer order(), usando os identificadores ByteOrder.BIG_ENDIAN e ByteOrder.LITTLE_ENDIAN, assim:

byteArray.order(ByteOrder.BIG_ENDIAN);

Você pode recuperar a ordem nativa para sua plataforma usando o método estático ByteOrder.nativeOrder(). Sabemos que você está curioso:

jshell> importar java.nio.ByteOrder;

jshell> ByteOrder.nativeOrder()

$4 ==> LITTLE_ENDIAN

Executamos isso em um desktop Linux com chip Intel. Experimente em seu próprio sistema!

Alocando buffers

Você pode criar um buffer alocando-o explicitamente usando allocate() ou agrupando um tipo de array Java simples existente. Cada tipo de buffer possui um método estático allocate() que utiliza uma capacidade (tamanho) e um método wrap() que utiliza um array existente:

CharBuffer cbuf = CharBuffer.allocate(64*1024);

ByteBuffer bbuf = ByteBuffer.wrap(someExistingArray);

Um buffer direto é alocado da mesma maneira, com o método allocateDirect(): ByteBuffer bbuf2 = ByteBuffer.allocateDirect(64*1024);

Conforme descrevemos anteriormente, os buffers diretos podem usar estruturas de memória do sistema operacional otimizadas para uso com alguns tipos de operações de E/S. A desvantagem é que a alocação de um buffer direto é uma operação um pouco mais lenta e pesada do que um buffer simples, portanto, você deve tentar usá-los para buffers de longo prazo.

Codificadores e decodificadores de caracteres

Codificadores e decodificadores de caracteres transformam caracteres em bytes brutos e vice-versa, mapeando do padrão Unicode para esquemas de codificação específicos. Codificadores e decodificadores existem há muito tempo em Java para uso pelos fluxos Reader e Writer e nos métodos da classe String que funcionam com matrizes de bytes. No entanto, no início não havia API para trabalhar explicitamente com codificação; você simplesmente se referiu aos codificadores e decodificadores sempre que necessário pelo nome como uma String. O pacote java.nio.charset formalizou a ideia de uma codificação de conjunto de caracteres Unicode com a classe Charset.

A classe Charset é uma fábrica para instâncias Charset, que sabem como codificar buffers de caracteres em buffers de bytes e decodificar buffers de bytes em buffers de caracteres. Você pode procurar um conjunto de caracteres por nome com o método estático Charset.forName() e usá-lo em conversões:

```
Charset charset = Charset.forName("US-ASCII");

CharBuffer charBuff = charset.decode(byteBuff); // para ascii
```

ByteBuffer byteBuff = charset.encode(charBuff); // e volta

Você também pode testar para ver se uma codificação está disponível com o método estático Charset.isSupported().

Os seguintes conjuntos de caracteres têm garantia de fornecimento:

•

EUA-ASCII

•

ISO-8859-1

•

UTF-8

•

UTF-16BE (big endian)

•

UTF-16LE (little endian)

•

UTF-16

Você pode listar todos os codificadores disponíveis em sua plataforma usando o método estático availableCharsets():

Mapa mapa = Charset.availableCharsets();

Iterador it = map.keySet().iterator();

enquanto (it.hasNext())

System.out.println(it.next());

O resultado de availableCharsets() é um mapa, porque os conjuntos de caracteres podem ter “aliases” e aparecer sob mais de um nome.

Além das classes orientadas a buffer do pacote java.nio, as classes ponte InputStreamReader e OutputStreamWriter do pacote java.io também funcionam com Charset. Você pode especificar a codificação como um objeto Charset ou por nome.

CharsetEncoder e CharsetDecoder

Você pode obter mais controle sobre o processo de codificação e decodificação criando uma instância de CharsetEncoder ou CharsetDecoder (um codec) com os métodos Charset newEncoder() e newDecoder(). No trecho anterior, presumimos que todos os dados estavam disponíveis em um único buffer. Mais frequentemente, porém, talvez tenhamos que processar os dados à medida que chegam em pedaços. A API do codificador/decodificador permite isso, fornecendo métodos encode() e decode() mais gerais que recebem um sinalizador especificando se mais dados são esperados. O

codec precisa saber disso porque pode ter ficado parado no meio de uma conversão de caracteres multibyte quando os dados acabaram. Se souber que mais dados

estão chegando, não ocorrerá um erro nesta conversão incompleta.

No trecho a seguir, usamos um decodificador para ler um bbuff ByteBuffer e acumular dados de caracteres em um cbuff CharBuffer:

decodificador CharsetDecoder = Charset.forName("US-ASCII").newDecoder(); booleano concluído = falso;

enquanto (!concluído) {

bbuff.clear();

feito = (in.read(bbuff) == -1);

bbuff.flip();

decoder.decode(bbuff, cbuff, pronto);

}

cbuff.flip();

//usa cbuff. . .

Aqui, procuramos a condição de fim de entrada no canal in para definir o sinalizador como concluído. Observe que aproveitamos o método flip() em ByteBuffer para definir o limite para a quantidade de dados lidos e redefinir a posição, preparando-nos para a operação de decodificação em uma única etapa. Em caso de problemas, tanto encode() quanto decode() retornam um objeto de resultado, CoderResult, que pode determinar o progresso da codificação. Os métodos isError(), isUnderflow() e isOverflow() no CoderResult especificam por que a codificação foi interrompida: por um erro, falta

de bytes no buffer de entrada ou buffer de saída cheio, respectivamente.

Canal de arquivo

Agora que cobrimos o básico sobre canais e buffers, é hora de examinar um tipo de canal real. O FileChannel é o equivalente NIO do java.io.RandomAccessFile, mas fornece vários recursos aprimorados, além de algumas otimizações de desempenho.

Em particular, você pode usar um FileChannel no lugar de um fluxo de arquivo java.io simples se desejar usar bloqueio de arquivo, acesso a arquivos mapeados na memória ou transferência de dados altamente otimizada entre arquivos ou entre arquivos e canais de rede. Todos esses são casos de uso bastante avançados, mas se você realizar trabalho de back-end ou lidar com grandes quantidades de dados, eles certamente serão úteis.

Você pode criar um FileChannel para um caminho usando o método estático FileChannel.open():

```
FileSystem fs = FileSystems.getDefault();

Caminho p = fs.getPath(" /tmp/foo.txt");

//Abre o padrão para leitura

tente (canal FileChannel = FileChannel.open(p)) {

//leia do canal...

}

//Abre com opções para escrita

importar java.nio.file.StandardOpenOption.* estático;
```

tente (canal FileChannel =

FileChannel.open(p, WRITE, APPEND, ...)) {

// acrescenta a foo.txt se já existir,

// caso contrário, crie-o e comece a escrever...

}

Por padrão, open() cria um canal somente leitura para o arquivo. Podemos abrir um canal para escrever ou anexar e controlar outros recursos mais avançados, como criação atômica e sincronização de dados, passando opções adicionais, conforme mostrado na segunda parte do exemplo anterior.Tabela 10-3resume essas opções.

Tabela 10-3. java.nio.file.StandardOpenOption

Opção

Descrição

ACRESCENTAR

Abra o arquivo para escrita; todas as gravações são

posicionadas no final do arquivo.

CRIAR

Use com WRITE para abrir o arquivo e criá-lo, se necessário.

CRIE UM NOVO

Use com WRITE para criar um arquivo atomicamente;

falhando se o arquivo já existir.

Opção

Descrição

DELETE_ON_CLOSE

Tente excluir o arquivo quando ele for fechado ou, se estiver

aberto, quando a VM for encerrada.

LER, ESCREVER

Abra o arquivo somente leitura ou somente gravação (o

padrão é somente leitura). Use ambos para leitura e gravação.

ESCASSO

Use ao criar um novo arquivo; solicita que o arquivo seja

esparso. Em sistemas de arquivos onde isso é suportado, um

arquivo esparso lida com arquivos muito grandes, em sua

maioria vazios, sem alocar tanto armazenamento real para

porções vazias.

SINCRONIZAR,

Sempre que possível, garanta que as operações de gravação

DSYNC

sejam bloqueadas até que todos os dados sejam gravados no

armazenamento. O SYNC faz isso para todas as alterações de

arquivo, incluindo dados e metadados (atributos), enquanto o

DSYNC adiciona esse requisito apenas para o conteúdo de

dados do arquivo.

TRUNCATE_EXISTING Use WRITE em um arquivo existente; defina o comprimento do arquivo como zero ao abri-lo.

Um FileChannel também pode ser construído a partir de um FileInputStream, FileOutputStream ou RandomAccessFile clássico:

FileChannel readOnlyFc = novo FileInputStream("arquivo.txt")

.getChannel();

FileChannel readWriteFc = new RandomAccessFile("arquivo.txt", "rw")

.getChannel();

Canal de arquivos criados a partir desses fluxos de entrada e saída de arquivos são somente leitura ou somente gravação, respectivamente. Para obter um FileChannel de leitura/gravação, você deve construir um RandomAccessFile com opções de leitura/gravação, como no exemplo anterior.

Usar um FileChannel é como um RandomAccessFile, mas funciona com um ByteBuffer em vez de matrizes de bytes:

ByteBuffer bbuf = ByteBuffer.allocate(...);

bbuf.clear();

readOnlyFc.posição(índice);

readOnlyFc.read(bbuf);

bbuf.flip();

readWriteFc.write(bbuf);

Você pode controlar quantos dados são lidos e gravados definindo a posição do buffer e os marcadores de limite ou usando outra forma de leitura/gravação que assume

uma posição inicial e comprimento do buffer. Você também pode ler e escrever em uma posição aleatória fornecendo índices com os métodos de leitura e gravação: readWriteFc.read(bbuf,índice)

readWriteFc.write(bbuf, index2);

Em cada caso, o número real de bytes lidos ou gravados depende de vários fatores. A operação tenta ler ou gravar até o limite do buffer e, na grande maioria das

vezes, é isso que acontece com o acesso a arquivos locais. É garantido que a operação será bloqueada apenas até que pelo menos um byte tenha sido processado. Aconteça o que acontecer, o número de bytes processados é retornado e a posição do buffer é atualizada de acordo, preparando você para repetir a operação até que ela seja concluída, se necessário. Esta é uma das conveniências de trabalhar com buffers; eles podem gerenciar a contagem para você. Como os streams padrão, o método read() do canal retorna -1 ao atingir o final da entrada.

O tamanho do arquivo está sempre disponível com o método size(). Isso pode mudar se você escrever além do final do arquivo. Por outro lado, você pode truncar o arquivo para um comprimento especificado com o método truncate().

Acesso simultâneo

Canal de arquivos são seguros para uso por vários threads e garantem uma visualização consistente desses dados entre canais na mesma VM. Entretanto, a menos que você especifique as opções SYNC ou DSYNC, os canais não garantem a rapidez com que as gravações serão propagadas para o mecanismo de armazenamento. Se você precisar ter certeza de que os dados estão seguros apenas de forma intermitente antes de prosseguir, poderá usar o método force() para liberar as alterações no disco. Este método usa um argumento booleano que indica se os metadados do arquivo, incluindo carimbo de data/hora e permissões, devem ser incluídos. Alguns sistemas rastreiam leituras e gravações de arquivos, então você pode salvar muitas atualizações se definir o sinalizador como falso, o que indica que você

não se importa em sincronizar esses metadados imediatamente.

Tal como acontece com todos os canais, qualquer thread pode fechar um FileChannel.

Uma vez fechado, todos os métodos de leitura/gravação e relacionados à posição do canal lançam uma ClosedChannelException.

Bloqueio de arquivo

Canal de arquivos suportam bloqueios exclusivos e compartilhados em regiões de arquivos através do método lock():

FileLock bigLock = fileChannel.lock();

int start = 0, len = fileChannel2.size();

FileLock readLock = fileChannel2.lock (iniciar, len, verdadeiro);

Os bloqueios podem ser compartilhados ou exclusivos. Um bloqueio exclusivo evita que outros adquiram um bloqueio de qualquer tipo no arquivo ou região de arquivo especificado. Um bloqueio compartilhado permite que outros adquiram bloqueios compartilhados sobrepostos, mas não bloqueios exclusivos. Eles são úteis como bloqueios de gravação e leitura, respectivamente. Quando você está escrevendo, você não quer que os outros possam escrever até que você termine, mas ao ler, você só precisa impedir que os outros escrevam e não leiam.

O método lock() sem argumentos no exemplo anterior tenta adquirir um bloqueio exclusivo para todo o arquivo.

A segunda forma aceita parâmetros de início e comprimento, bem como um sinalizador indicando se o bloqueio deve ser compartilhado (verdadeiro) ou exclusivo (falso). O objeto FileLock retornado pelo método lock() pode ser usado para liberar o bloqueio:

bigLock.release();

Aviso

Os bloqueios de arquivos só têm garantia de cooperação. Eles funcionam quando todos os threads os respeitam; eles não impedem necessariamente que um thread não cooperante leia ou grave um arquivo bloqueado. Em geral, a única maneira de garantir que os bloqueios sejam obedecidos é ambas as partes tentarem adquirir o bloqueio e prosseguirem somente se a tentativa for bem-sucedida.

Além disso, os bloqueios compartilhados não são implementados em alguns sistemas; nesse caso, todos os bloqueios solicitados são exclusivos. Você pode testar se um bloqueio é compartilhado com o método isShared().

Canal de arquivoos bloqueios são mantidos até que o canal seja fechado ou interrompido, portanto, executar bloqueios em uma instrução try-with-resources ajudará a garantir que os bloqueios sejam liberados de forma mais robusta: tente (canal FileChannel = FileChannel.open (p, WRITE)) {

canal.lock();

// ...

}

Exemplo de FileChannel

Vamos ver alguns usos concretos de nossos canais e buffers. Criaremos um pequeno arquivo de texto que inclui uma contagem de quantas vezes ele foi acessado pelo nosso programa. Nosso programa, então, abrirá o arquivo, lerá a contagem atual, incrementará essa contagem e então gravará (bem, substituirá) a contagem de volta no arquivo. Você pode experimentar uma versão completa dos trechos abaixo no arquivo AccessNIO.java na pasta ch10/examples.

Observação

Você poderia absolutamente resolver este projeto usando as classes de E/S padrão em java.io. O conjunto NIO não se destina a substituir as classes antigas por atacado, mas

a adicionar funcionalidades que estão faltando nas classes padrão sem quebrar nenhum código que dependa dessas classes. Se você achar o NIO um pouco complexo ou denso, sinta-se à vontade para ignorá-lo até precisar de alguns dos recursos ausentes, como bloqueio de arquivos ou manipulação de metadados.

Nossa primeira tarefa é verificar se nosso arquivo de contagem de acesso existe (access.txt neste exemplo, mas o nome é arbitrário). Caso contrário, precisamos criá-lo (e definir o contador de acesso interno para 1). Podemos usar um objeto Path com os métodos auxiliares estáticos de Arquivos para começar:

classe pública AccessNIO {

String accessFileName = "access.txt";

```
Caminho accessFilePath = Path.of(accessFileName);

int contagem de acesso = 0;

Canal de acesso ao FileChannel;

acesso públicoNIO() {

// ...

booleano inicial = !Files.exists(accessFilePath);

accessChannel = FileChannel.open(accessFilePath,
CREATE, READ, WRITE);

// ...

}

}
```

Se o arquivo ainda não existir, podemos escrever uma mensagem inicial (“Este arquivo foi acessado 0 vezes.”) e então retroceder até o início do novo arquivo. Isso nos dá a mesma linha de base para trabalhar, como se o arquivo existisse o tempo todo:

```
se (inicial) {

String mensagem = buildMessage(); // ajudante para
consistência

accessChannel.write(ByteBuffer.wrap(msg.getBytes()));

accessChannel.position(0);

}
```

Se o arquivo existir, precisamos ter certeza de que podemos ler e gravar nele.

Podemos reunir essas informações com o objeto accessChannel que criamos no construtor. Certamente poderíamos adicionar outros testes e mensagens de erro mais detalhadas, mas essas verificações mínimas são úteis:

```
public boolean isReady() {

return (accessChannel != null &&
accessChannel.isOpen());

}
```

Agora chegamos ao nosso caso de uso principal. O arquivo existe e tem algum conteúdo. Temos as permissões apropriadas para tudo o que queremos fazer.

Começaremos abrindo o arquivo em modo leitura/gravação e lendo seu conteúdo em uma string:

```
int fsize = (int)accessChannel.size();

// Dê-nos espaço extra caso conte

// ultrapassa um limite de dígito (9 -> 10, 99 -> 100, etc.)

ByteBuffer in = ByteBuffer.allocate(fsize + 2);

accessChannel.read(in);

String atual = new String(in.array());
```

Queremos que o arquivo seja legível por humanos por si só, por isso não aproveitaremos a capacidade do FileChannel de ler e gravar dados binários. Podemos usar

nosso conhecimento de como uma única linha de texto está estruturada para analisar nossa contagem de acessos:

```
int contagemInício = 28;

// Sabemos onde começa o número da contagem, então obtenha

// tudo desde aquela posição até o próximo espaço

String rawCount = current.substring(countStart,

current.indexOf(" ", contagemIniciar));

accessCount = Integer.parseInt(rawCount) + 1;
```

Finalmente, podemos redefinir nossa posição e substituir a linha anterior por nossa linha nova e atualizada. Observe que também truncamos nosso arquivo até o final da mensagem salva. Nos demos espaço extra para acomodar um número maior, mas não queremos espaço em excesso no arquivo real:

```
String mensagem = buildMessage();

accessChannel.position(0);

accessChannel.write(ByteBuffer.wrap(msg.getBytes()));

accessChannel.truncate(accessChannel.position());

accessChannel.close();
```

Tente compilar e executar este exemplo algumas vezes. A contagem aumenta conforme o esperado? O que acontece se você abrir o arquivo em outro programa, como um editor de texto? Infelizmente, Java NIO parece

apenas mágica. Acessar o arquivo usando qualquer outro programa não alterará necessariamente seu conteúdo de acordo com as regras do nosso pequeno exemplo.

embrulhar

Quase qualquer aplicativo destinado à distribuição precisará lidar com E/S de arquivos. Java possui suporte robusto para trabalhar eficientemente com arquivos locais, incluindo acesso a metadados para arquivos e diretórios. O compromisso do Java com uma ampla compatibilidade aparece na variedade de codificações de caracteres disponíveis ao trabalhar com arquivos de texto. Java certamente também é conhecido por trabalhar com arquivos não locais. Abordaremos E/S de rede e recursos da Web emCapítulo 13.

Perguntas de revisão

1. Como você poderia verificar se um determinado arquivo já existe?

2. Se você tiver que trabalhar com um arquivo de texto legado usando um esquema de codificação antigo, como ISO 8859, como configurar um leitor para converter adequadamente esse conteúdo para algo como UTF-8?

3. Qual pacote tem as melhores classes para E/S de arquivos sem bloqueio?

4. Que tipo de fluxo de entrada você pode usar para analisar um arquivo binário, como uma imagem compactada em JPEG?

5. Quais são os três fluxos de texto padrão integrados à classe System?

6. Os caminhos absolutos começam em uma raiz (/ ou C:\, por exemplo). Onde começam os caminhos relativos? Mais especificamente, para onde estão os caminhos relativos?

7. Como você recupera um canal NIO de um FileInputStream existente?

Exercícios de código

Para esses exercícios, um arquivo esqueleto Count.java está na pasta ch10/exercises, mas fique à vontade para começar com sua própria aula. Iteramos em um único projeto, para que você possa usar a solução do primeiro exercício como ponto de partida para o segundo e assim por diante. Como testar o programa requer o fornecimento de arquivos diferentes na linha de comando, pode ser mais fácil executar este programa a partir de um terminal ou janela de comando. Você certamente também pode usar a guia do terminal no seu IDE:

1. Usando as classes do pacote java.io, crie um pequeno programa que imprimirá o tamanho de um arquivo especificado na linha de comando. Por exemplo: C:\> java Contagem ../examples/ListIt.java

Analisando ListIt.java

Tamanho: 1011 bytes

Se nenhum argumento de arquivo for fornecido, imprima uma mensagem de erro em System.err.

2. Expanda o exercício anterior para abrir o arquivo fornecido e contar o número de linhas. (Para estes exercícios simples, não há problema em assumir que o

arquivo que está sendo analisado é um arquivo de texto.) Se você quiser praticar um pouco com algumas das ferramentas doCapítulo 8, divida cada linha com base no espaço em branco e inclua uma contagem de palavras na saída. (Você pode usar expressões regulares para dividir palavras em padrões mais sofisticados, como pontuação, mas isso não é obrigatório.)

C:\> java Contagem ../examples/ListIt.java

Analisando ListIt.java

Tamanho: 1011 bytes

Linhas: 36

Palavras: 178

Como antes, se nenhum argumento de arquivo for fornecido, imprima uma mensagem de erro em System.err.

3. Converta sua solução anterior para usar classes NIO como Path e Files em vez de leitores. Você pode usar qualquer parte dos pacotes java.nio e java.nio.file que desejar. É quase certo que você ainda precisará da classe

java.io.IOException da E/S “antiga”, é claro.

Exercícios Avançados

1. Aceite uma segunda linha de comando contendo o nome de um arquivo de log de estatísticas. Em vez de imprimir as várias contagens no terminal, anexe uma linha contendo o carimbo de data/hora atual, o nome do arquivo e suas três contagens. O formato exato da linha não é realmente importante, mas deve ser algo assim:

02/02/2023 08:14:25 Contagem1.java 36 147 1002

Você pode usar seu NIO ou sua antiga solução de E/S (OIO?) como ponto de partida. Se você optar pela versão NIO, tente usar um ByteBuffer e um FileChannel para fazer a escrita.

Se apenas um argumento de linha de comando for fornecido, volte a imprimir as estatísticas na tela como antes. Se nenhum argumento for fornecido ou se o segundo argumento não for gravável, imprima um erro em System.err.

Execute esta versão algumas vezes em alguns arquivos. Verifique seu log para ter certeza de que cada novo resultado está anexado corretamente ao final do arquivo de log e não o está sobrescrevendo.

1Embora o NIO tenha sido introduzido com o Java 1.4 – portanto, não é mais tão novo

– ele era mais recente que o pacote básico original e o nome pegou.

2Erro padrão (stderr) é um fluxo geralmente reservado para mensagens de texto relacionadas a erros que devem ser mostradas ao usuário de um aplicativo de linha de comando. É diferente da saída padrão (stdout), que geralmente é redirecionada para um arquivo de log ou outro aplicativo e não é vista pelo usuário.

3O termo é emprestado do mundo dos threads e significa a mesma coisa: a criação de um arquivo atômico não pode ser interrompida por outro thread.

4Na programação OO, o termo fábrica normalmente se refere a um auxiliar estático que pode construir e

personalizar algum objeto. Uma fábrica (ou método de fábrica) é semelhante a um construtor, mas essa adaptação adicional pode adicionar detalhes ao novo objeto que podem ser difíceis (ou impossíveis) de especificar em um construtor.

5Os termos big-endian e little-endian vêm do romance As Viagens de Gulliver, de Jonathan Swift, onde denotam dois campos de liliputianos: aqueles que comem ovos na ponta grande e aqueles que os comem na ponta menor.

Capítulo 11. Abordagens Funcionais em Java

Java é – e continua sendo – uma linguagem orientada a objetos. Todos os padrões de design e tipos de classe que vimos emcapítulo 5ainda são fundamentais para a forma como a maioria dos desenvolvedores escreve código Java. Java também é flexível, com colaboradores individuais e corporativos propondo e fazendo melhorias. À medida que a programação funcional (FP) volta ao centro das atenções, o Java está acompanhando. FP representa uma forma alternativa de abordar a programação: funções, em vez de objetos, são o foco.

Desde o Java 8, o Java oferece suporte a um conjunto razoável de recursos funcionais com o pacote java.util.function. Este pacote inclui diversas classes e interfaces que permitem aos desenvolvedores usar abordagens funcionais populares para resolução de problemas. Exploraremos algumas dessas abordagens neste capítulo, mas queremos enfatizar o verbo permitir. Se você não gosta de programação funcional, pode ignorar este capítulo com segurança. Esperamos que você experimente alguns dos exemplos. Existem alguns recursos interessantes que podem tornar seu código mais compacto, mantendo sua legibilidade.

Funções 101

As raízes da programação funcional remontam à década de 1930, com o matemático americano Alonzo Church e seu cálculo lambda. Church não estava executando seu cálculo em nenhum hardware, mas o cálculo lambda formalizou uma forma de resolução de problemas que levaria às primeiras linguagens de programação escritas para máquinas operacionais reais.1A linguagem Lisp foi desenvolvida na década de 1950 no MIT e funcionou nas primeiras iterações de computadores modernos, como a série IBM 700. Se você consegue imaginar uma foto antiga em preto e branco com paredes do tamanho de uma estante de luzes piscando, você tem a ideia certa de até que ponto as ideias e padrões de FP remontam na história da computação.

Mas FP não é a única forma de programar um computador. Outros paradigmas, como programação processual e programação orientada a objetos (OOP), disputam regularmente popularidade. Felizmente, você pode atingir os mesmos objetivos em qualquer um desses paradigmas. O paradigma que você escolhe geralmente se resume ao domínio do problema e mais do que uma preferência pessoal.

Considere as tarefas simples de somar dois números e atribuir esse resultado a uma variável. Podemos fazer isso em linguagens orientadas a objetos como Java, em linguagens funcionais como Clojure,2ou linguagens procedurais como C:

```
//objetos Java

BigInteger cinco = new BigInteger(5);

BigInteger doze = novo BigInteger(12);
```

```
Soma BigInteger = cinco.add(doze);

// Clojure

(definitivo cinco 5)

(def doze 12)

(def soma (+ cinco doze))

// C

int cinco = 5;

int doze = 12;

int soma = cinco + doze;
```

Java atingiu o cenário digital quando a OOP estava (novamente) em ascensão e reflete essas raízes. Mesmo assim, a FP sempre teve os seus evangelistas. Java 8 ofereceu algumas adições substanciais à linguagem e abriu a porta para os fãs de programação funcional trabalharem com Java. Vamos dar uma olhada em algumas dessas adições e ver como elas se integram ao mundo mais amplo do Java.

Lambdas

Inspiradas no cálculo lambda, as expressões lambda (ou mais simplesmente, lambdas) formam a unidade central da programação funcional em Java. Lambdas são um meio de encapsular um pouco de lógica. Em uma linguagem funcional, as funções são

“cidadãs de primeira classe” e podem ser criadas, armazenadas, referenciadas, usadas e transmitidas como objetos em Java. Para imitar essa funcionalidade, o Java 8

introduziu uma nova sintaxe junto com várias interfaces especiais. Essas adições permitem definir rapidamente uma função que pode substituir uma classe interna inteira. O resultado dessa definição ainda é um objeto oculto, é claro, mas cuja

“objetividade” está em grande parte oculta.

Abordaremos as expressões lambda e essas interfaces especiais com mais detalhes no restante desta seção. Em seguida, veremos um exemplo popular e concreto do uso dessas expressões para realizar um trabalho real.

Expressões Lambda

Expressões lambda são pequenos pedaços de código que podem aceitar parâmetros e retornar valores, assim como os métodos. Ao contrário dos métodos, porém, você pode facilmente passar um lambda como argumento para algum outro método ou armazená-lo em uma variável, como faria com uma referência de objeto. Os proponentes do FP valorizam essa capacidade de mover código como se fossem dados.

Ele permite que você escreva código interessante e dinâmico sem a confusão de criar classes internas ou anônimas.

Observação

Lambdas não foram feitos para fornecer aumento de desempenho. Embora o uso criterioso de lambdas geralmente produza um código-fonte mais compacto e conciso, essa compactação não remove nenhuma complexidade. Lambdas podem exigir menos digitação, mas não realizam menos trabalho.

Lembre-se do método run() usado por threads, que vimos com tanta frequência

emCapítulo 9. Criamos mais do que algumas pequenas classes que implementaram a interface Runnable para fornecer um “corpo” aos nossos threads. Classes pequenas que não incluem nenhum estado como variáveis de instância são os principais candidatos para usar lambdas: você tem uma tarefa bem definida que pode ser usada em uma situação bem definida.

Vamos revisitar uma de nossas demonstrações de thread e então dar uma olhada em como poderíamos usar uma expressão lambda como alternativa ao uso explícito de Runnable. Simplificaremos a classe VirtualDemo de“Morte de um fio”e concentre-se na classe interna anônima:

```
classe pública VirtualDemo2 {

public static void main(String args[]) lança exceção {

Executável executável = new Runnable() {

execução nula pública() {

System.out.println("Olá tópico! ID: " +

Thread.currentThread().threadId());

}

};

Thread t = Thread.startVirtualThread(executável);

juntar();
```

```
}

}
```

Criamos uma nova instância de Runnable com um método run() simples que imprime uma saudação e o número de ID do thread:

```
% java --enable-preview VirtualDemo2

Olá tópico! ID: 20
```

Ótimo. Tudo funciona conforme o esperado. Agora vamos substituir essa variável executável por uma expressão lambda:

```
classe pública VirtualDemo3 {

public static void main(String args[]) lança exceção {

Tópico t = Thread.startVirtualThread(() ->

System.out.println("Olá tópico! ID: " +

Thread.currentThread().threadId())

);

juntar();

}

}
```

Ainda iniciamos um novo thread virtual e ainda armazenamos esse thread em uma variável (t, em ambos os exemplos), mas não há evidência da interface Runnable.

Passamos um argumento um tanto estranho para o método startVirtualThread() em vez de uma referência a algum objeto. Esse “argumento estranho” é a nossa expressão lambda, anotada emFigura 11-1.

Figura 11-1. Estrutura básica de uma expressão lambda

Este lambda específico é muito simples. Não passamos nenhum argumento e ele não retorna um valor. Muitas vezes isso é tudo que você precisa. Mas lambdas são capazes de muito mais. As expressões lambda também suportam argumentos, podem retornar valores e podem ter corpos mais interessantes.

Passando argumentos

Se pensarmos nos lambdas como pedaços de código, é razoável compará-los com métodos regulares. Os métodos regulares encapsulam a lógica, assim como os lambdas. Mas muitos dos métodos que vimos nos capítulos anteriores também aceitam argumentos. Podemos fornecer argumentos para um lambda?

Considere um iterador que percorre os elementos de uma coleção Java. Vimos vários exemplos emCapítulo 7.

Nesses exemplos, usamos um iterador dentro de um loop, com o corpo do loop fazendo algo com determinado elemento da coleção em cada passagem. Lembre-se do nosso ciclo de pintura de árvores de"Aplicação: Árvores no

Campo":

//Arquivo: Field.java

protegido void paintComponent(Gráficos g) {

g.setColor(fieldColor);

g.fillRect(0,0, getWidth(), getHeight());

for (Árvore t: árvores) {

t.draw(g);

}

// ...

}

O loop for alternativo usa o iterador de árvores para obter cada árvore individual e, em seguida, diz a essa árvore para se desenhar em nosso campo. Poderíamos substituir esse loop por um lambda e o método forEach() da interface Iterable:

//Arquivo: Field.java

protegido void paintComponent(Gráficos g) {

g.setColor(fieldColor);

```
g.fillRect(0,0, getWidth(), getHeight());

árvores.forEach(t -> t.draw(g));

// ...

}
```

Você pode ver o mesmo operador de seta, mas em vez de um par vazio de parênteses, temos uma variável t no lado esquerdo. Essa variável recebe uma árvore por vez da coleção de árvores, assim como o loop for alternativo do primeiro trecho. E assim como o corpo desse loop for, você pode usar a árvore atual no lado direito da expressão. Com esse arranjo, obtemos uma versão um pouco mais concisa do nosso loop, mas mantém sua legibilidade. Você pode usar esse truque útil com qualquer coleção que implemente a interface Iterable.

Observação

Termos como conciso e legível são julgamentos subjetivos. Os defensores do FP

definitivamente acham a sintaxe mais compacta dos lambdas mais fácil de ler, mas essas pessoas já estão confortáveis com a notação. Esperamos que você experimente os exemplos e exercícios deste capítulo para ganhar um pouco dessa familiaridade.

Gostamos de lambdas e as usamos em diversas situações, mas elas nunca são obrigatórias. Se você não achar os lambdas úteis ou legíveis depois de experimentá-los, não será necessário usá-los em seu próprio código.

Você deve ter notado que nossa primeira expressão lambda para um corpo de thread simples não tinha argumentos, então usamos um conjunto vazio de parênteses no lado esquerdo. Mas neste exemplo mais recente, tivemos um argumento e nenhum parênteses. A forma de argumento único é tão comum que o compilador permite essa abreviação sem parênteses. Se você não tiver argumentos ou tiver mais de um argumento, os parênteses serão obrigatórios.

Corpos de expressão

As expressões lambda brilham em situações em que você usaria uma classe interna anônima. Você não pode substituir lambdas em todas as situações que exigem classes internas anônimas, mas há um número surpreendente de pontos em Java que funcionam com lambdas. Com esta variedade de aplicações surge a necessidade de uma computação mais complexa, além das declarações impressas. Se precisar executar algumas instruções ou trabalhar com uma variável temporária, por exemplo, você pode colocar o corpo da expressão entre chaves, assim como um método.

Imagine que as árvores do nosso jogo são sazonais. Você pode especificar a cor das folhas antes de desenhá-las. Você ainda pode usar um lambda:

```
árvores.forEach(t -> {

t.setLeafColor(getSeasonalColor());

t.draw(g);

});
```

Presumivelmente, nosso método fictício getSeasonalColor() faz alguns bons cálculos baseados em datas e retorna uma cor apropriada. Observe que você pode usar métodos (e a maioria das variáveis) do restante da classe dentro de nossa expressão

lambda. Lambdas são bastante poderosos. Mas parte de seu poder vem do uso criterioso – um corpo de expressão de 20 linhas provavelmente prejudicaria a legibilidade do seu código. Mas se você tiver algumas lambdas com algumas linhas, estará em boa forma.

Além de manter suas lambdas legíveis, queremos apontar algumas peculiaridades que podem surgir para você. Se você quiser usar uma variável local do escopo envolvente, ela deverá ser “efetivamente final” de acordo com a documentação. Lembre-se de que as variáveis finais não podem ser modificadas. Efetivamente, variáveis finais são aquelas que não são modificadas, mesmo que não tenham a palavra-chave oficial final em sua declaração. Se você tentar usar uma variável local não final, o compilador irá reclamar. Felizmente, esta restrição só se aplica a variáveis locais. Você é livre para usar (e até mesmo modificar) variáveis declaradas como membros da classe anexa.

A outra peculiaridade gira em torno da palavra-chave this. Se você se lembra de“A

referência “isto””, isso fornece uma referência ao objeto atual. É útil quando métodos ou construtores têm nomes de argumentos que se sobrepõem a variáveis de membro: Posição da classe pública {

interno x, y;

Posição pública(int x, int y) {

```
isto.x = x;

isto.y = y;

}

// ...

}
```

Embora você possa razoavelmente pensar que this dentro de um corpo lambda se referiria ao próprio lambda, na verdade ainda se refere à classe envolvente. Essa peculiaridade significa que você pode usar isso dentro de um lambda da mesma forma que faria com o construtor do exemplo anterior. Isso garante que seus lambdas tenham acesso ao material da sua classe, mesmo que uma variável local possa ocultar algo.

Retornando valores

Quer o seu lambda seja uma linha única ou inclua uma dúzia de linhas que atrapalham a legibilidade, você também pode retornar um valor. Um exemplo (enganosamente) simples é uma função incremental que recebe um argumento inteiro e retorna um número inteiro que é um a mais que a entrada. A expressão em si seria mais ou menos assim:

```
x -> x + 1
```

Nesta forma, Java calculará a resposta para x + 1 e a retornará. Se tivermos um corpo multilinha que deve retornar um valor, podemos usar a palavra-chave return:

```
x -> {

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

retornar x + 1;

}

Retornos explícitos podem ser úteis quando você possui instruções if no corpo. Mas a forma mais simples é preferível se a sua expressão se adequar.

Interfaces Funcionais

Você pode estar se perguntando como o Java categoriza nossa expressão lambda simples. É um int como a entrada ou o resultado? É um objeto parecido com Java? É

algo que ainda não vimos? Vamos ver se o jshell pode esclarecer alguma coisa: jshell> x -> x + 1

| Erro:

| tipos incompatíveis: java.lang.Object não é uma interface funcional

| x -> x + 1

| ^--------^

Hmm, não é exatamente o que esperávamos, mas a frase interface funcional é uma pista. Vamos tentar aquela palavra-chave var que vimos em“Inferir tipos”e veja se nossa expressão lambda pode ser inferida:

jshell> var inc = x -> x + 1

| Erro:

| não é possível inferir o tipo para a variável local inc

| (a expressão lambda precisa de um tipo de destino explícito)

| var inc = x -> x + 1;

| ^------------------^

Atirar. jshell reconheceu nossa expressão lambda, mas esse reconhecimento não é suficiente para estabelecer um tipo.

Então, qual é o tipo de expressão lambda? No caso de nosso lambda de incremento simples, acaba sendo uma IntFunction, uma função que aceita um int como argumento e retorna um int. A interface IntFunction reside no pacote java.util.function junto com várias outras interfaces funcionais. Cada interface neste pacote representa uma

“forma” diferente que uma expressão lambda pode assumir. Vamos tentar: jshell> importar java.util.function.*

jshell> IntFunction inc = x -> x + 1

inc ==> $Lambda$24/0x0000000840087840@23ab930d

Viva! Não recebemos nenhum erro! (Embora o valor resultante pareça bastante assustador.) Felizmente, não precisamos nos preocupar com os detalhes internos do nosso lambda, desde que possamos aplicá-lo a alguns dados. Mas como poderíamos aplicá-lo?

Dê uma olhada nodocumentação on-linepara a interface e você verá que ela tem um método, apply(), de forma bastante apropriada:

```
jshell> inc.apply(7)

$ 16 ==> 8
```

Outro viva! Nosso incrementador incrementa! As outras interfaces possuem um método semelhante definido.

As formas que mencionamos cobrem os diferentes arranjos dos argumentos e resultados das expressões lambda. Se quiséssemos trabalhar com valores duplos em vez de inteiros, por exemplo, poderíamos usar a interface DoubleFunction. Se quisermos fornecer um objeto como argumento, mas não precisarmos retornar um valor, poderíamos usar a interface Consumer<T>. (Como Consumer trabalha com tipos de referência, ele é parametrizado. Se realmente quiséssemos armazenar um lambda que aceitasse uma string, usaríamos o tipo Consumer<String>.) Ou talvez tenhamos um lambda que não aceita argumentos, mas gera um valor longo: a interface LongSupplier resolverá o problema. Não reproduziremos aqui a lista completa de interfaces funcionais, mas vale a pena dar uma olhada noresumo do pacoteon-line.

À medida que você encontrar mais situações em que pode usar lambdas, verá como todas essas formas diferentes são usadas. Mas é importante ressaltar que o termo interface funcional pode ser aplicado a qualquer interface que possua um único método abstrato (geralmente abreviado como SAM na documentação). EmCapítulo 12,

por exemplo, usaremos lambdas para lidar com eventos da interface do usuário, como clicar em um botão. Os eventos de botão são relatados para ActionListeners. A interface ActionListener possui um método abstrato,

actionPerformed(), portanto ela se qualifica como uma interface funcional, mesmo que fizesse parte do Java muito antes de esses recursos funcionais serem adicionados.

Referências de métodos

Um outro recurso está associado à abordagem funcional do Java: uma referência de método. Às vezes, suas expressões lambda são apenas wrappers para outros métodos.

Consideremos a tarefa muito popular de imprimir o conteúdo de uma coleção.

Poderíamos usar o método forEach() que acabamos de aprender e imprimir os elementos de uma lista usando um lambda:

```
jshell> Lista<String> nomes = new ArrayList<>()

nomes ==> []

jshell> nomes.add("Kermit");

$3 ==> verdadeiro

jshell> nomes.add("Fozzie");

$4 ==> verdadeiro

jshell> nomes.add("Gonzo");

$ 5 ==> verdadeiro

jshell> nomes.add("Piggy");

$ 6 ==> verdadeiro
```

```
jshell> nomes.forEach(n -> System.out.println(n))

Sapo

Fozzie

Gonzo

Porquinho
```

Você pode ver que nossa expressão lambda simplesmente transfere cada string para o método System.out.println(). Este é o candidato certo para uma referência de método.

Especificamos tal referência com um operador de dois pontos separando o método de seu objeto (ou sua classe, no caso de um método estático):

```
jshell> nomes.forEach(System.out::println)

Sapo

Fozzie

Gonzo

Porquinho
```

Muito compacto e ainda legível. As referências de métodos funcionam apenas em um conjunto restrito de circunstâncias, mas são opções populares sempre que permitidas.

Tal como acontece com as expressões lambda em geral, não há benefício real de desempenho em comparação ao uso de um lambda. Na verdade, o compilador Java cria uma expressão lambda a partir da nossa referência de

método nos bastidores: jshell> Consumidor<String> impressora = System.out::println

impressora ==> $Lambda$27/0x0000000840087440@63c12fb0

Sinta-se à vontade para usar referências de método onde elas couberem, mas também é bom usar uma expressão lambda explícita se achar mais fácil de ler.

Lambdas Práticas: Classificação

Nossa, isso foi muita teoria! É hora de colocar essas expressões lambda em uso em alguns códigos que você costuma encontrar em aplicativos reais: classificação de dados. A classificação é uma tarefa comum; conversamos sobre isso em "Uma análise

mais detalhada: o método sort()" enquanto discutia coleções. Onde os lambdas se encaixam?

Para colocar qualquer lista em ordem, você precisa comparar dois elementos da lista para saber qual deve vir antes do outro. Algumas listas – digamos, uma lista de salários de funcionários ou uma lista de nomes de arquivos e subpastas em um determinado diretório – têm uma ordem bastante natural que é suficiente na maioria dos casos. Mas às vezes você precisa de uma ordem mais complexa, como classificar as subpastas no topo, antes dos arquivos. Você poderia implementar a interface Comparable como fez antes ou criar uma classe personalizada que implemente a interface Comparator intimamente relacionada, mas também poderia fornecer um lambda.

Para que uma expressão lambda ajude na classificação, ela precisa se comportar como o método compare() do

Comparator. Precisamos de uma expressão que receba dois argumentos, digamos a e b, e retorne um dos três valores int:

•

Algo menor que zero se a < b

•

Algo maior que zero se a > b

•

Zero se a == b

A magia das lambdas nos permite decidir como organizamos uma lista de forma dinâmica. A classe auxiliar java.util.Collections contém um método sort() que aceita uma coleção para classificar, junto com um comparador para fornecer a ordem.

Podemos usar um lambda para fazer essa comparação. Por exemplo, poderíamos criar um lambda simples para classificar nossa lista de nomes em ordem alfabética:
jshell> Collections.sort(nomes, (a, b) -> a.compareTo(b))

jshell> nomes

nomes ==> [Fozzie, Gonzo, Kermit, Piggy]

Organizado conforme o esperado, embora pudéssemos ter usado qualquer um dos truques de classificação do Java para obter essa ordem padrão. Vamos inverter a ordem:

jshell> Collections.sort(nomes, (a, b) -> b.compareTo(a))

```
jshell> nomes

nomes ==> [Piggy, Caco, Gonzo, Fozzie]
```

Organizado! Tudo o que tivemos que fazer foi trocar a ordem dos argumentos usando o método compareTo().

Lambdas podem fazer mais, é claro, especialmente quando você precisa solicitar algo um pouco mais complexo do que uma lista de nomes. Imagine pegar as árvores em nosso jogo de arremesso de maçãs e classificá-las pela distância da origem, (0,0), usando um lambda um pouco mais interessante:

```
Coleções.sort(árvores, (t1, t2) -> {

var t1x = t1.getPositionX();

var t1y = t1.getPositionY();

var t2x = t2.getPositionX();

var t2y = t2.getPositionY();

var dist1 = Math.sqrt(t1x * t1x + t1y * t1Y);

var dist2 = Math.sqrt(t2x * t2x + t2y * t2y);

retornar dist1 - dist2;

});
```

Tornamos o corpo desta expressão mais detalhado do que o necessário para enfatizar que lambdas podem ter muitas linhas de código. Este lambda provavelmente prejudica a legibilidade, mas também destaca um efeito colateral útil de tais expressões: você consegue ver o código sendo usado para classificar exatamente onde a

classificação é feita. Este recurso de autodocumentação é outra razão pela qual o FP

tem tantos proponentes.

Fluxos

Como observamos antes, as expressões lambda não fazem nada que você não pudesse realizar usando outros recursos do Java, mas fornecem uma maneira diferente de pensar sobre os problemas. Na mesma linha, a API Java Streams (não deve ser confundida com todas as várias classes “Stream”, como PrintStream no pacote java.io) fornece uma maneira diferente de pensar sobre os dados.

Você pode obter um stream de uma das classes do pacote java.util.stream ou usando o método stream() de uma coleção. Um fluxo fornece um fluxo constante de objetos e você executa operações em cada objeto à medida que o encontra. As operações podem filtrar objetos indesejados, contá-los ou até mesmo alterá-los antes de repassá-los. Em situações em que há grandes quantidades de dados, os fluxos oferecem uma maneira concisa de processar todos esses dados. Como programador, você pode se concentrar em como lidar com um único objeto e deixar o fluxo fazer o trabalho de preparar esses objetos únicos para você.

Fontes e Operações

Para testar streams, precisaremos de uma fonte de dados. Um começo fácil é usar o método stream() em qualquer classe que implemente a interface Collection ou um de seus descendentes. (Arrays não têm uma opção de stream integrada, mas você pode criar uma facilmente com o método estático Stream.of().)

Assim que tivermos um fluxo em andamento, podemos operá-lo. Veremos muitas outras operações a seguir, mas um ponto de partida simples e popular é a operação count(). Não é de surpreender que esta operação conte cada elemento do fluxo à medida que ele passa e produz um único resultado. Por exemplo, podemos usar nossa lista de nomes em jshell e descobrir quantos amigos estão em nossa lista: jshell> nomes

nomes ==> [Fozzie, Gonzo, Kermit, Piggy]

jshell> nomes.stream().count()

$ 24 ==> 4

É certo que este exemplo não faz nada de surpreendente, mas iremos desenvolver operações mais complexas. O importante a observar é a maneira como atribuímos uma operação ao nosso fluxo. O stream() retorna um objeto stream e usamos o operador ponto (.) imediatamente para obter nossa contagem.

Poderíamos usar outra operação para imprimir nossos nomes:

jshell> nomes.stream().forEach(System.out::println)

Fozzie

Gonzo

Sapo

Porquinho

Fornecemos uma referência de método para a operação forEach(), mas você também pode fornecer um lambda

que recebe um argumento (o nome atual do fluxo) e não retorna um valor.

Reutilização de fluxo

Você deve ter notado que não armazenamos nosso stream em uma variável para reutilização entre nosso exemplo count() e o exemplo forEach() semelhante. Os fluxos são unidirecionais e de uso único. Na verdade, você pode armazenar um fluxo em uma variável, mas se tentar reutilizá-lo depois de processá-lo, receberá um erro: jshell> Stream<String> nameStream = nomes.stream()

nameStream ==> java.util.stream.ReferencePipeline$Head@621be5d1

jshell> nomeStream.count()

$ 27 ==> 4

jshell> nameStream.forEach(System.out::println)

| Exceção java.lang.IllegalStateException: stream tem

| já foi operado ou fechado

| em AbstractPipeline.sourceStageSpliterator

| (AbstractPipeline.java:279)

| em ReferencePipeline$Head.forEach

| (ReferencePipeline.java:658)

| em (#28:1)

Criamos um objeto Stream parametrizado para nosso fluxo de objetos String.

Iniciamos o stream com sucesso e usamos a operação count(), mas não conseguimos usar o mesmo stream para nossa operação forEach(). O processamento de um fluxo não altera a fonte original, portanto, você pode iniciar um novo fluxo com segurança sempre que necessário. Mas depois que um fluxo termina, ele não pode ser reiniciado.

Geradores de fluxo

Outra fonte de dados de fluxo é um gerador. Os geradores criam dados de acordo com alguma regra. Alguns geradores produzem um valor fixo repetidamente, enquanto outros produzem conteúdo aleatório. Você pode gerar coisas simples como números ou coisas complexas como objetos. Se obter dados reais for uma operação cara, você poderá usar geradores para testar mais facilmente sua lógica de fluxo. Da mesma forma, você pode usar um gerador para criar dados bons (ou peculiares, ou cheios de erros) para testar outras partes do seu aplicativo.

O método Stream.generate() utiliza uma instância da interface Supplier. Um fornecedor tem uma função: fornecer um fluxo infinito de elementos. Possui um método: get(), que retorna um elemento do tipo apropriado. E o tipo do elemento é realmente a única restrição que Java impõe ao seu gerador. Vamos tentar gerar algo simples: um fluxo constante do número 42:

```
jshell> Stream.generate(() ->
42).limit(3).forEach(System.out::println) 42

42
```

Nosso Fornecedor neste caso é o lambda muito simples, () → 42. Não há argumentos, e toda vez que a expressão lambda é usada ou avaliada, o resultado é 42. Observe que seguimos nosso método generate() com um novo método , limit(), que fica entre o gerador e nossa etapa println(). Por si só, os geradores geram para sempre.

Discutiremos limit() e outros métodos relacionados na próxima seção, mas precisamos de algo no curto prazo para controlar nossos geradores. Se você não acredita em nós, tente remover essa peça. Basta estar pronto para pressionar Ctrl-C

rapidamente (e repetidamente) para impedir o ataque de 42s infinitos!

Podemos implementar a interface Supplier (ou qualquer uma de suas primas do tipo base, como IntSupplier) em uma classe quando precisarmos de um conjunto mais interessante de dados gerados. Considere um fluxo de nomes de dias aleatórios.

Precisamos de um gerador de números aleatórios e uma lista de dias válidos. Esses requisitos provavelmente resultariam em um lambda embutido confuso, mas são triviais em uma classe pequena:

```
importar java.util.Random;

importar java.util.stream.Stream;

importar java.util.function.Supplier;

classe pública WeekDayGenerator implementa
Supplier<String> {
```

```
String estática privada[] dias =

{ "Dom", "Seg", "Ter", "Qua", "Qui", "Sex", "Sábado" };
private Random randSrc = new Random();

string pública get() {

retornar dias[randSrc.nextInt(dias.length)];

}

public static void main(String args[]) {

Stream.generate(novo WeekDayGenerator())

.limite(5)

.forEach(System.out::println);

}

}
```

O método main() aqui não é necessário, mas facilita o teste. Basta compilar e executar a classe da pasta ch11/examples. Você deverá ver cinco dias aleatórios da semana:

```
% cd ch11/exemplos

% javac WeekDayGenerator.java

% java WeekDayGenerator

Sol

qui
```

sex

Sol

seg

Tente executá-lo algumas vezes apenas para confirmar se o recurso aleatório está funcionando. Sua classe geradora pode ser tão rica quanto necessário. Você só precisa ter certeza de que get() retorna um objeto ou valor apropriado: observe que implementamos uma versão parametrizada de nossa interface: Suppler<String>, e nosso método get() retorna String. Agora você está pronto para ir!

Iteradores de fluxo

Além dos geradores, os fluxos podem ser construídos a partir de iteradores. Esses iteradores não são exatamente iguais aos iteradores que você usa para percorrer uma coleção, mas a ideia é semelhante. Os iteradores de fluxo têm a mesma noção de

“próximo” valor que os iteradores de coleção, mas para fluxos, esse próximo valor vem da realização de um cálculo no valor anterior. Se você precisar de um intervalo de números sequenciais, por exemplo, um iterador é ideal:

jshell> IntStream.iterate(1, i -> i + 1).limit(5).forEach(System.out::pri ntln)

1

2

3

4

5

O método fonte iterate() aceita dois argumentos: um valor inicial e uma expressão lambda. O lambda pega um argumento e o usa para criar o próximo elemento. Esse segundo elemento será colocado de volta na mesma expressão lambda para criar o terceiro e assim por diante. Certamente poderíamos ter feito isso com um fornecedor personalizado, mas para muitas sequências, os iteradores oferecem um ponto de entrada mais simples. E você não está restrito a iterar números — você pode iterar em qualquer tipo de objeto que atenda às suas necessidades. Contanto que você tenha uma maneira de calcular o próximo objeto do fluxo, poderá usar um iterador como fonte. Vamos tentar criar uma sequência de objetos LocalDate como exemplo: jshell> importar java.time.LocalDate

jshell> importar java.time.temporal.ChronoUnit

jshell> Stream.iterate(LocalDate.now(),

...> d -> d.plus(1, ChronoUnit.DAYS)).limit(5).forEach(System.out::printl n)

10/02/2023

11/02/2023

12/02/2023

13/02/2023

14/02/2023

Usamos o método estático LocalDate.now() para obter a data atual para nosso valor inicial. A expressão iterativa recebe um objeto LocalDate como entrada, usa o método plus() para adicionar um dia e retorna o novo LocalDate. (E terminamos em um encontro tão lindo.)

Filtrando fluxos

As operações count() e forEach() nos trechos anteriores são exemplos de operações de terminal. As operações de terminal “encerram” um fluxo. Você pode ter apenas uma operação de encerramento final ao processar um fluxo. A operação limit(), por outro lado, é um exemplo de operação intermediária. Uma operação intermediária pode alterar ou remover alguns dados do fluxo, mas o fluxo continua. A filtragem é um tipo popular de operação intermediária, e limitar o número de elementos que continuam no fluxo é uma forma de filtragem. Mas você pode filtrar por vários motivos. Você pode filtrar para selecionar dados desejáveis ou descartar dados indesejáveis. Você pode filtrar duplicatas. Você pode inserir um fluxo de objetos em seu filtro e fazer com que ele produza um fluxo essencialmente novo para uso na próxima operação.

Por sorte, os filtros genéricos são apenas lambdas que retornam valores booleanos.

Esta é a forma Predicado da grande lista de interfaces funcionais no pacote java.util.function. Você envia um argumento e sai verdadeiro ou falso. Por exemplo, poderíamos usar um filtro para contar os nomes que contêm a letra “o” assim: jshell> nomes.stream().filter(n -> n.indexOf("o") > -1).count() $ 30 ==> 2

Nosso lambda de filtragem leva um nome e usa a operação indexOf() para ver se o nome contém um “o”. Como indexOf() retorna um valor int, nós o comparamos a um índice impossível, -1, para criar o resultado booleano necessário. Se o predicado retornar verdadeiro, esse nome será repassado. Se o predicado retornar falso, o nome será simplesmente eliminado do fluxo.

O detalhe importante novamente é a natureza “intermediária” de um filtro. Podemos continuar fazendo coisas com nosso stream. É comum empilhar vários filtros, por exemplo. Cada filtro seleciona diferentes elementos desejados (ou remove elementos indesejados, dependendo de como você olha para ele). Outro filtro integrado popular é a operação distinta() que elimina duplicatas. Vamos adicionar alguns nomes repetidos à nossa lista e tentar usar duas operações intermediárias:

```
jshell> nomes.add("Gonzo")

$ 32 ==> verdadeiro

jshell> nomes

nomes ==> [Fozzie, Gonzo, Caco, Piggy, Gonzo]

jshell> nomes.stream().

...> filtro(n -> n.indexOf("o") > -1).count()

$ 34 ==> 3

jshell> nomes.stream().

...> filtro(n -> n.indexOf("o") > -1).

...> distinto().count()
```

$ 35 ==> 2

Você pode empilhar quantos filtros precisar, embora ainda seja importante manter seu código legível. (Se você tiver um desafio de 20 filtros, você pode querer reconsiderar como você processa a fonte do stream.) Mas você pode fazer mais do que simplesmente filtrar os elementos do seu stream: você pode transformá-los em outra coisa!

Mapeando Fluxos

Em fluxos, mapeamento é o processo de alterar um elemento em um fluxo antes de transmiti-lo. Assim como na filtragem, você usa uma expressão lambda para realizar a alteração. Você pode mapear mudanças simples, como adicionar imposto sobre vendas a um fluxo de preços, ou pode criar mapas complexos que convertem um tipo de objeto em um tipo totalmente diferente. Ou você pode fazer as duas coisas! O

mapeamento também é uma operação intermediária, portanto você pode empilhar operações de mapa da mesma forma que fez com os filtros. Na verdade, você verá muitos exemplos online de programadores misturando mapas e filtros para alcançar o resultado final.

Vamos começar tentando a tarefa de adicionar imposto sobre vendas. Começaremos com uma pequena lista de valores duplos e uma taxa de 5%. Podemos mapear() o imposto sobre os preços da seguinte forma:

jshell> double[] preços = { 5,99, 9,99, 20,0, 8,5};

preços ==> duplo[4] { 5,99, 9,99, 20,0, 8,5 }

```
jshell>
DoubleStream.of(preços).forEach(System.out::println)

5,99

9,99

20,0

8,5

jshell> DoubleStream.of(preços).map(p -> p*1,05).

...> forEach(System.out::println)

6.2895
```

10.489500000000001

21,0

8.925

A formatação dos nossos preços não é muito polida, mas o imposto foi aplicado corretamente. Embora os tenhamos em mãos, podemos tentar outra operação de terminal útil, sum(), para somar todos os preços:

jshell> DoubleStream.of(preços).map(p -> p*1,05).sum()

$ 7 ==> 46,70400000000001

Novamente, a saída não está bem formatada, mas resumimos um array inteiro de números em uma linha!

Mapeando atributos de objeto

Você também pode usar mapas para examinar objetos internos. Vamos criar uma variação simplificada da nossa classe Employee deCapítulo 7com um atributo salarial adicional. Chamaremos esta versão de PaidEmployee:

classe pública PaidEmployee {

id interno privado;

nome da string privada;

salário interno privado; // anual, em dólares inteiros

public PaidEmployee(String nome completo, int id, int salário) {

este.nome = nome completo;

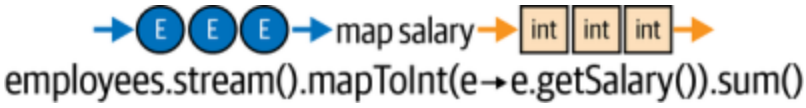
isto.id = id;

```
este.salário = salário;

}

public String getNome() { nome de retorno; }

public int getID() {retorna id; }

public int getSalary() { return salário; }

}
```

Em um fluxo de funcionários, agora podemos usar map() para extrair atributos específicos, como seus nomes. Vamos escrever uma classe de teste que crie alguns exemplos de objetos de funcionários e depois use um fluxo para processar os funcionários:

```
importar java.util.*;

relatório de classe pública {

List<PaidEmployee> funcionários = new ArrayList<>();

void buildEmployeeList() {

funcionários.add(new PaidEmployee("Fozzie", 4,
30_000));

funcionários.add(new PaidEmployee("Gonzo", 2,
50_000));

funcionários.add(new PaidEmployee("Kermit", 1,
60_000));

funcionários.add(new PaidEmployee("Piggy", 3, 80_000));

}
```

```
public void publicarNomes() {

funcionários.stream().map(e ->
e.getName()).forEach(System.out::println);

}

public static void main(String args[]) {

Relatório r = novo Relatório();

r.buildEmployeeList();

r.publishNames();
```




```
}

}
```

O métodopublishNames() usa map() para pegar nosso objeto PaidEmployee e pegar o nome do funcionário. Esse nome (um objeto String simples) continua no fluxo. Com os nomes disponíveis, poderíamos adicionar filtros, como o filtro “nomes com um o” dos exemplos anteriores, ou ficar atentos a registros duplicados de funcionários. Sempre que você precisar massagear seus dados, map() é o método a ser usado.

Mapeando conversões

No exemplo anterior, convertemos silenciosamente nosso fluxo de um com objetos PaidEmployee para um com objetos String. Como ambos os tipos são tipos de referência, não precisamos realmente nos preocupar com

o fato de termos tipos antes e depois diferentes. Se você precisar passar de um tipo de referência para um tipo base — ou vice-versa — você terá que ser um pouco mais explícito sobre a conversão.

Esta é definitivamente uma tarefa comum, então Java fornece algumas variações úteis de map() exatamente para esse propósito. Vamos fazer uma soma dos salários anuais de todos os nossos funcionários para saber qual deve ser o nosso orçamento salarial: public void publicarOrçamento() {

int b = funcionários.stream().mapToInt(e -> e.getSalary()).sum();

System.out.println("O orçamento anual é " + b);

}

Figura 11-2ilustra os dados que passam por esse fluxo de cálculo do orçamento.

Figura 11-2. Convertendo entre objetos e ints em um stream

Existem classes semelhantes para mover para dois outros tipos base: mapToDouble() e mapToLong(). Se você já possui um fluxo de números e deseja passar para um objeto, todos os fluxos de tipo base, como IntStream, incluem a operação mapToObj().

Mapas planos

Queremos apresentar outro tipo de operação de mapeamento comumente usada com streams: o flatmap. A operação flatmap pega entradas irregulares e as suaviza em um único fluxo de elementos (você pode até

dizer plano!). O que queremos dizer com entrada irregular? É principalmente uma maneira fofa de dizer dados

multidimensionais. Considere o tabuleiro de xadrez de arrays que discutimos

em“Matrizes Multidimensionais”. Podemos brincar com uma configuração semelhante no jshell usando valores int simples. O “tabuleiro” é uma matriz de linhas, onde cada linha é uma matriz de números. O que acontece se tentarmos iniciar um fluxo a partir do nosso array bidimensional? Vamos experimentar com uma matriz 4 × 4 reduzida: jshell> int[][] placa = {

...> {2, 0, 4, 2},

...> {0, 3, 0, 1},

...> {5, 0, 1, 0},

...> { 2, 3, 0, 2 } }

placa ==> int[4][] { int[4] ... , int[4] { 2, 3, 4, 2 } }

jshell> Arrays.stream(board).forEach(System.out::println)

[E@5a10411

[I@2ef1e4fa

[Eu@306a30c7

[Eu@b81eda8

Hmm, isso parece ser um fluxo de objetos int[], não números inteiros individuais.

(Esses blobs feios são a maneira padrão do Java de imprimir objetos que não possuem um método toString() bonito. O formato é “tipo de objeto @ endereço de memória”.) E

se tentarmos iniciar um fluxo a partir da primeira linha?

```
jshell>
Arrays.stream(board[0]).forEach(System.out::println)

2

3

4

2
```

Essa saída parece melhor – é uma lista dos valores de nossas peças de xadrez inventadas, mas é apenas uma linha. Poderíamos colocar o fluxo no meio de um loop for e processar um fluxo separado para cada linha, mas isso parece complicado e tornaria qualquer tipo de contagem ou soma muito mais difícil.

Para uma maneira agradável e amigável de colocar todas as nossas peças de xadrez em um fluxo, usaremos flatMap() ou, no nosso caso, de passar de um objeto (cada linha é um objeto de array) para um tipo base (cada peça de xadrez é um int), usaremos flatMapToInt():

```
jshell> Arrays.stream(placa).

...> flatMapToInt(r -> Arrays.stream(r)).

...> forEach(System.out::println)

2
```

3

4

2

0

0

0

0

0

0

0

0

2

3

4

2

Viva, funcionou! Começamos com um fluxo de objetos densos e dividimos esses objetos densos em um único fluxo de partes menores. Você pode usar flatMap() e seus primos do tipo base para lançar quaisquer dados multidimensionais tabulares, cúbicos ou genéricos em um fluxo agradável de elementos individuais.

Vejamos outro exemplo que combina vários tópicos de stream que abordamos até agora. Uma tarefa comum para administradores de sistema e de rede é analisar arquivos de log. Os servidores Web, por exemplo, registram o endereço IP (Internet Protocol) de cada visitante e o recurso solicitado. Aqui está um pequeno exemplo, com as linhas longas truncadas para facilitar a leitura:

54.152.182.118 - - [20/Set/2020:08:28:46 -0400] "OBTER / ...

107.150.59.82 - - [20/Set/2020:09:33:02 -0400] "OBTER / ...

66.249.65.234 - - [20/Set/2020:09:33:54 -0400] "OBTER /robots.txt ...

66.249.65.243 - - [20/Set/2020:09:33:54 -0400] "OBTER /robots.txt ...

Cada linha contém muitas informações: o endereço IP, a data e hora da solicitação, o que foi solicitado, como foi solicitado e (se você espiar o arquivo de log real na pasta ch11/examples) informações sobre qual navegador ou o agente do usuário fez a solicitação. Os arquivos de log do mundo real podem ser enormes e os

administradores geralmente os mantêm compactados no disco.

Vamos usar todas as nossas habilidades de stream. Começaremos com um arquivo GZIP e carregaremos seu conteúdo com alguns fluxos de E/S e, em seguida, dividiremos os dados descompactados em linhas. Podemos usar flatMap() para transformar o fluxo

funcional de linhas em um fluxo de tokens separados por espaço.

Com nossos tokens em mãos, podemos finalmente chegar às informações que realmente desejamos: uma contagem de endereços IP de visitantes únicos: importar java.util.regex.Pattern;

importar java.util.stream.Stream;

importar java.util.zip.GZIPInputStream;

importar java.io.*;

classe pública UniqueIPs {

public static void main(String args[]) {

Separador de padrões = Pattern.compile("\\s+");

Padrão ipAddress = Pattern.compile("\\d+\\.\\d+\\.\\d+\\.\\d+"); tentar {

//Abra o arquivo...

FileInputStream fis = new FileInputStream("sample-access.log.gz");

// Em seguida, descompacte o arquivo com um wrapper...

GZIPInputStream gis = novo GZIPInputStream(fis);

// Em seguida, envolva a entrada descompactada em um leitor

InputStreamReader ir = novo InputStreamReader(gis);

// Em seguida, coloque um leitor em buffer...

```
BufferedReader br = novo BufferedReader(ir);

// Isso finalmente nos dá nosso stream!

// Agora vamos processar esse stream para obter
algumas informações inter essantes

resultado longo = br.lines()

.flatMap(ls -> separator.splitAsStream(ls))

.filter(palavra -> ipAddress.matcher(palavra).matches())

.distinto()

.contar();

System.out.println("Encontrados " + resultado + " IPs
exclusivos.");

} catch (IOException ioe) {

System.err.println("Ah, não! Algo deu errado: " + ioe);

}

}

}
```

Muito legal. Assim que tivermos o fluxo de linhas de nosso arquivo descompactado, podemos obter nossa contagem usando a abordagem funcional para processar dados com uma lista de etapas compacta, eficiente e legível. Novamente, você não precisa usar fluxos funcionais e lambdas, mas cada vez mais programadores estão buscando essa maneira de resolver problemas —

mesmo em linguagens antigas orientadas a objetos, como Java.

Reduzindo e coletando

Vimos vários exemplos usando streams neste capítulo. Cada exemplo finalizou o fluxo com uma das três operações de terminal: um forEach() que normalmente usamos para imprimir os elementos, um count() para saber quantos elementos existem, ou com sum() como forma de somar todos dos elementos numéricos. Contar e somar são exemplos de redução de um fluxo. Você "reduz" todos os elementos do seu fluxo a uma única resposta.

Streams em Java possuem vários redutores integrados, conforme mostrado emTabela 11-1.

Tabela 11-1. Operações de redução de terminais

Operação

Descrição

contar()

Retorna o número de elementos no fluxo

encontre algum()

Retorna um elemento (se existir) do fluxo

encontrarPrimeiro() Retorna o primeiro elemento (se existir) do fluxo combinarTodos()

Retorna verdadeiro se todos os elementos do fluxo

corresponderem aos critérios fornecidos

Operação

Descrição

combinarAny()

Retorna verdadeiro se pelo menos um elemento no fluxo

corresponder aos critérios fornecidos

máximo()

Usando um comparador fornecido, retorna o elemento

“maior” (se houver)

min()

Usando um comparador fornecido, retorna o elemento

“menor” (se houver)

Você pode estar se perguntando por que a operação sum() que usamos algumas vezes não está listada. É definitivamente um redutor, mas a versão integrada está disponível apenas nos fluxos do tipo base: IntStream, LongStream e DoubleStream.

Valores opcionais

Antes de nos aprofundarmos nos redutores (incluindo como criar um redutor personalizado), precisamos estar preparados para um resultado potencialmente terrível: um fluxo vazio.

Sempre que você filtrar um fluxo, é possível que não reste nada do outro lado do filtro.

Como um exemplo rápido, e se o filtro em nosso exemplo anterior de nomes estivesse procurando a letra “a” em vez de “o”? Nenhum de nossos nomes contém “a”, então o filtro acabaria eliminando todos os nomes de nossa lista. A operação count() pode lidar bem com essa situação: ela simplesmente retorna uma resposta zero. Mas e se tivéssemos usado min() ou findFirst()? Esses redutores esperam fornecer um elemento correspondente do seu stream. Se não houver mais elementos, o que um redutor deve retornar? Pode ser aceitável em alguns cenários retornar um valor nulo, mas se o seu fluxo terminar com elementos do tipo base, como valores int, você não poderá usar nulo.

Em vez de forçar você a construir alguma regra estranha ou lançar uma exceção, os fluxos Java suportam a noção de uma resposta opcional. Essas respostas estão agrupadas em uma classe, apropriadamente chamada de Opcional, do pacote java.util.

Um objeto Opcional possui dois métodos principais com os quais trabalharemos nesta seção: isPresent() nos informa se um valor existe ou não, e get() retorna esse valor.

(Se você chamar get() quando nenhum valor estiver presente, você “obterá” uma NoSuchElementException.)

Podemos testar essa ideia opcional revisitando nosso exemplo de filtragem de nomes.

Em vez de contar os resultados, usaremos findFirst() para retornar o primeiro nome correspondente. Como pode não haver nenhuma correspondência, obteremos o resultado agrupado em um Opcional. Sinta-se à vontade para reutilizar a coleção de nomes se ela ainda estiver no

seu jshell, mas aqui vai uma rápida recapitulação: jshell> Lista<String> nomes = new ArrayList<>()

nomes ==> []

jshell> nomes.add("Kermit")

jshell> nomes.add("Fozzie")

jshell> nomes.add("Gonzo")

jshell> nomes.add("Porquinho")

jshell> nomes

nomes ==> [Caco, Fozzie, Gonzo, Piggy]

Agora vamos passar esses nomes por um filtro e procurar a primeira correspondência.

Tentaremos nosso filtro com um “o” (que deveria ter resposta) e depois com um “a”

(que não deveria ter resposta). Observe como usamos o resultado Opcional: jshell> Opcional correspondente =names.stream().

...> filtro(n -> n.indexOf("o") > -1).findFirst()

correspondente ==> Opcional[Fozzie]

jshell> System.out.println(matched.isPresent() ?

...> matched.get() : "N/A")

Fozzie

jshell> Opcional correspondente =names.stream().

...> filtro(n -> n.indexOf("a") > -1).findFirst()

correspondido ==> Opcional.vazio

jshell> System.out.println(matched.isPresent() ?

...> matched.get() : "N/A")

N / D

Embora torne seu código um pouco mais detalhado para testar se seu valor éPresent(), Opcional fornece uma interface clara para lidar com os resultados bons e

“ruins” de seu processamento de fluxo. E como acontece com tantas outras classes e métodos nesta área funcional, você pode usar as classes OpcionalInt, OpcionalLong e Opcional Double para capturar resultados de tipo base potencialmente ausentes.

Criando um redutor personalizado

E se os redutores integrados não atenderem às suas necessidades? Você pode não se surpreender ao saber que pode fornecer um lambda para criar um redutor personalizado. A classe Stream inclui uma operação reduz() que aceita um lambda com a forma BinaryOperator do pacote java.util.function que discutimos

em“Interfaces Funcionais”. BinaryOperator aceita dois argumentos do mesmo tipo e retorna um valor (também do mesmo tipo). Dependendo de suas necessidades, você pode usar reduzir() apenas com o operador binário lambda ou pode usar um segundo formulário que também assume um valor inicial do mesmo tipo usado

pelo operador binário. Vamos experimentar esta segunda forma para criar um redutor fatorial personalizado.

Fatoriais são números grandes – ou podem ser, de qualquer maneira. Se o termo não lhe parecer familiar, é semelhante a uma operação de soma, mas em vez de somar

cada número na sequência, você multiplica. Normalmente você usa o ponto de exclamação para indicar esta operação: 5! (pronuncia-se “cinco fatorial”) multiplicará 5 e 4 perfazendo 20, então 20 × 3 perfaz 60, depois 60 × 2 perfaz 120 e, finalmente, 120 × 1 resulta em 120. Pode parecer bastante simples, mas os números fatoriais ficam muito grandes, muito rapidamente:

shell> 5 * 4 * 3 * 2 * 1

US$ 17 ==> 120

// já sabemos quais são 5! é, então reutilize esse valor

shell> 10*9*8*7*6*120

US$ 18 ==> 3628800

shell> 12 * 11 * 3628800

$ 19 ==> 479001600

Se você observar atentamente o resultado de 12!, notará que é pouco menos de meio bilhão, portanto ainda se enquadra na faixa (positiva) de valores para o tipo int. Mas 13! seria aproximadamente 6,5 bilhões, então não podemos armazenar essa resposta com inteiros. Poderíamos calculá-lo com posições compradas, mas mesmo esse tipo não consegue segurar nada depois de

20!. Felizmente, Java está pronto com algumas classes divertidas de java.math: BigInteger e BigDecimal. Essas classes podem abrigar valores arbitrariamente grandes, perfeitos para remover os limites dos tipos base em nosso trabalho fatorial.

Podemos usar um iterador simples como fonte, já que a multiplicação não requer uma ordem específica de operações. Nosso redutor fatorial sempre produzirá uma resposta semelhante a count() ou sum(),3então usaremos o segundo formulário com valor inicial de 1. Podemos tentar isso em jshell:

// Primeiro crie um alias rápido para nosso valor "1"

jshell> um = BigInteger.ONE

um ==> 1

// Teste com 12!

jshell> Stream.iterate(um, contagem ->
contagem.add(um)).

...> limit(12).reduce(one, (a, b) -> a.multiply(b))

$ 32 ==> 479001600

// Corresponde. Yay! Podemos conseguir 13!?

jshell> Stream.iterate(um, contagem ->
contagem.add(um)).

...> limit(13).reduce(one, (a, b) -> a.multiply(b))

$ 33 ==> 6227020800

// Viva. Grande teste a seguir, podemos conseguir 21!?

```
jshell> Stream.iterate(um, contagem ->
contagem.add(um)).

...> limit(21).reduce(one, (a, b) -> a.multiply(b))

$ 36 ==> 51090942171709440000

// Claro que sim! Agora, só para ser bobo, tente 99!

jshell> Stream.iterate(um, contagem ->
contagem.add(um)).

...> limit(99).reduce(one, (a, b) -> a.multiply(b))

$ 37 ==>
933262154439441526816992388562667004907159682
643

816214685929638952175999932299156089414639761
565

182862536979208272237582511852109168640000000
000

000000000000
```

A saída de 99! é tão grande que tivemos que cortá-lo arbitrariamente para caber na edição impressa deste livro.4Mas nosso redutor personalizado funcionou!

Dica

Você pode implementar a interface BinaryOperator em uma classe se sua lógica de redução for muito complexa para o lambda simples e embutido. Então você pode fornecer uma instância dessa classe para reduzir() em vez dos lambdas que usamos em nossos exemplos.

Colecionadores

As respostas que os redutores produzem costumam ser muito úteis. Quantas linhas você processou? Quantas vezes você viu uma palavra específica? Qual é a média de alguma coluna em dados tabulares? Mas e se você quiser mais do que uma única resposta? Ao filtrar, por exemplo, você pode querer manter todos os itens correspondentes do fluxo, em vez de contá-los ou somá-los.

E se você quiser uma nova lista apenas com os nomes que contêm a letra “o”?

Podemos usar um coletor.

A interface java.util.stream.Collector permite uma flexibilidade impressionante na maneira como você coleta e organiza os resultados do processamento do seu fluxo.

Não abordaremos coletores personalizados neste livro, mas, felizmente, a classe Collectors relacionada inclui vários coletores comuns como métodos estáticos. Por exemplo, podemos usar um desses métodos estáticos para obter aquela lista de nomes com o que nos interessa:

```
jshell> List<String> onames = nomes.stream().

...> filtro(n -> n.indexOf("o") > -1).

...> coletar(Collectors.toList())

onnames ==> [Fozzie, Gonzo]
```

Excelente. Agora onames é um objeto List<String> regular que podemos usar em qualquer outro lugar onde precisarmos dele. Existem muitos, muitos outros métodos de coleta que encorajamos você a dar uma olhada nodocumentação on-line para

colecionadores. Os exercícios de código no final deste capítulo lhe dão a oportunidade de experimentar outro coletor popular, groupingBy(), mas não temos tempo para cobrir todas as outras opções maravilhosas disponíveis.

Usando Lambdas diretamente

Queremos destacar outro recurso dos lambdas em Java: você pode usá-los em seu próprio código. Embora você provavelmente comece usando lambdas em algumas tarefas, como classificar coleções ou filtrar longos fluxos de dados, eventualmente você pode querer escrever métodos que aceitem um lambda como argumento a ser usado no corpo desse método. Como as expressões lambda são apenas instâncias de alguma interface funcional, Java torna a aceitação de lambdas bastante simples.

Considere um sensor digital: talvez algum dispositivo conectado a uma porta USB.

Muitos desses sensores são estáveis e consistentes, mas estão consistentemente desligados por algum fator. Talvez um termômetro pense que seu escritório em casa está sempre três graus mais quente do que realmente é, ou talvez um sensor de luz subestime a luz ambiente em 10%. Você pode escrever métodos de ajuste separados que “adicionem 3” a uma leitura ou que “reduzam em 10%” cada valor, mas também podem usar lambdas para

criar um método de ajuste genérico e permitir que o chamador forneça a lógica de ajuste.

Vamos ver como você pode escrever tal método. Para fazer seu método aceitar um lambda, você precisa decidir qual formato ele deve ter. Você sempre pode criar sua própria forma, é claro, mas muitas vezes você pode simplesmente usar uma das interfaces do pacote java.util.function. Para nossos ajustes de leitura do sensor, usaremos a forma DoubleUnaryOperator. (Um operador unário opera em um valor da mesma forma que um operador binário funciona em dois.) Aceitaremos um argumento duplo e retornaremos um duplo ajustado como resultado. Podemos colocar nosso ajustador incrivelmente flexível em um equipamento de teste simples para testar:

```
importar java.util.function.DoubleUnaryOperator;

ajustador de classe pública {

ajuste duplo estático público (duplo val,

Ajuste DoubleUnaryOperator)

{

retornar ajuste.applyAsDouble(val);

}

public static void main(String args[]) {

amostra dupla = 70,2;

System.out.println("Leitura inicial: " + amostra);

System.out.print("Adicionando 3: ");
```

```
System.out.println(ajustar(amostra, s -> s + 3));

System.out.print("Reduzindo em 10%: ");

System.out.println(ajustar(amostra, s -> s * 0,9));

}

}
```

Você pode ver que nosso método Adjust() aceita dois argumentos: o valor que queremos ajustar e o lambda que fará o ajuste. (E sim, você poderia implementar o DoubleUnaryOperator em uma classe e fornecer uma instância dessa implementação como alternativa.) Quando chamamos Adjust(), podemos usar a mesma sintaxe compacta que vimos em outras partes do arquivo oficial. JDK. É um pouco como usar magia proibida, mas é totalmente encorajado!

Se você compilar e executar esta demonstração, deverá ver um resultado semelhante a este:

```
$ ajustador java

Leitura inicial: 70,2

Adicionando 3: 73,2

Reduzindo em 10%: 63,1800000000001
```

Exatamente o que esperávamos. E poderíamos escrever outros ajustes sem ter que reescrever nosso método Adjust() real. Você provavelmente não precisará desse tipo de lógica dinâmica para todos os problemas que resolver em Java, mas vale a pena colocar esse truque

em sua caixa de ferramentas para que você possa utilizá-lo quando o fizer.

Próximos passos

Tal como acontece com tantos recursos do Java, poderíamos escrever um livro inteiro apenas sobre expressões ou fluxos lambda.Outros sim!Esperamos que esta introdução desperte seu apetite para aprender mais sobre os tópicos de PF. Se você quiser uma prática mais interativa com esses tópicos, recomendamos fortemente os laboratórios disponíveis na plataforma on-line. Nosso próprio Marc Loy criou duas séries, uma

emLambdas Javae outro emFluxos Java, ambos com exemplos práticos dos temas que abordamos neste capítulo

Perguntas de revisão

1. Qual pacote contém a maioria das interfaces funcionais introduzidas no Java 8?

2. Você precisa usar algum sinalizador especial ao compilar ou executar aplicativos Java que usam recursos funcionais como lambdas?

3. Como você cria expressões lambda com múltiplas instruções no corpo?

4. As expressões lambda podem ser nulas? Eles podem retornar valores?

5. Você pode reutilizar um fluxo depois de processá-lo?

6. Como você poderia pegar um fluxo de objetos e convertê-lo em um fluxo de números inteiros?

7. Se você tiver um fluxo que filtra linhas vazias de um arquivo, que operação você poderia usar para informar quantas linhas tinham algum conteúdo?

Exercícios de código

1. Nossa demonstração do Ajustador nos permite passar qualquer lambda que aceite e retorne um valor duplo. Não estamos restritos a mudanças simples, como adicionar um valor fixo. Adicione mais uma linha de saída que converte o número de uma leitura de Fahrenheit em Celsius. (Para relembrar

rapidamente, C = (F – 32) * 5/9. Nossa leitura de 70,2 deve sair em torno de 21,2.)

2. Usando as classes PaidEmployee e Report d[e“Mapeando atributos de objetos”,](#)

adicione um relatório simples semelhante a subscribeBudget() que exibe o salário médio de todos os funcionários.

Exercícios Avançados

1. Vamos explorar mais os colecionadores que mencionamos no final do capítulo.

Adicione um atributo “role” (do tipo String) à classe PaidEmployee. Certifique-se de atualizar o método buildEmployeeList() na classe Report para atribuir funções também. Sinta-se à vontade para escolher as funções que desejar, mas certifique-se de que pelo menos dois funcionários compartilhem a mesma função (para fins de teste).

Agora olhe para odocumentação para o coletor groupingBy(). Ele retorna um mapa dos grupos e seus membros. No nosso exemplo, as chaves deste mapa serão as funções que você criou. Os valores associados serão listas de todos os funcionários que compartilham essa função. Você pode adicionar mais um

“relatório” à classe Report que cria este mapa e depois imprime as funções e seus funcionários associados.

1Na verdade, Alan Turing, estudante de Church e pioneiro da computação, provou que o cálculo lambda era equivalente ao próprio sistema de Turing (a máquina de Turing fundamental) para realizar computação.

2Mencionamos Clojure em vez de inúmeras outras linguagens funcionais modernas porque ele roda na JVM e pode ser integrado a classes e métodos Java. Organizado!

3Uma peculiaridade do processo fatorial é que 0! é definido como “o número de maneiras de organizar itens em um conjunto vazio” – que é exatamente uma. Mesmo que nosso fluxo não tenha elementos, ainda podemos retornar corretamente o valor inicial.

4Isso é 9.3e+155 se a notação científica for mais fácil de analisar nesse tamanho. As estimativas populares para o número de átomos no universo conhecido giram em torno de 10e+82, caso você esteja se perguntando quão grande é 99! realmente é.

Capítulo 12. Aplicativos de Desktop

Java saltou para a fama e glória com o poder dos miniaplicativos – elementos interativos incríveis em uma página da web. Parece mundano hoje em dia, mas na

época era nada menos que uma maravilha. Java também tinha suporte

multiplataforma na manga e podia executar o mesmo código em sistemas Windows, Unix e macOS. Os primeiros JDKs tinham um conjunto rudimentar de componentes gráficos conhecidos coletivamente como Abstract Window Toolkit (AWT). O

“abstrato” no AWT vem do uso de classes comuns (Button, Window, etc.) com implementações nativas. Você escreve aplicativos AWT com código abstrato e multiplataforma; seu computador executa seu aplicativo e fornece componentes nativos concretos.

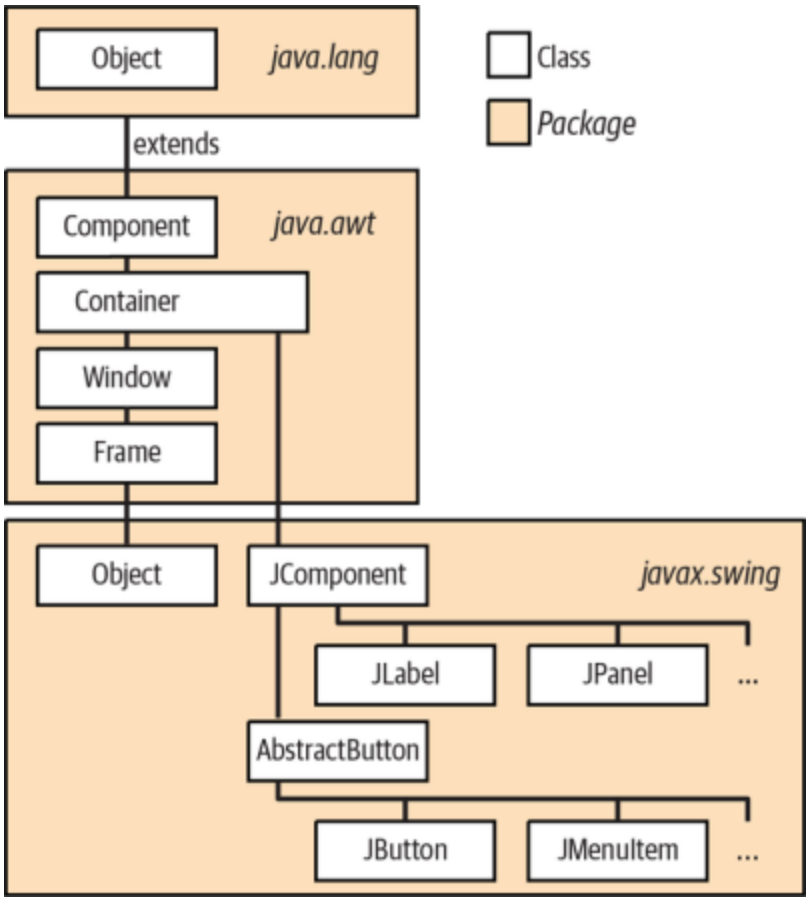
Infelizmente, essa combinação bacana de abstrato e nativo apresenta algumas limitações bastante sérias. No reino abstrato, você encontra designs de “menor denominador comum” que só dão acesso aos recursos disponíveis em todas as plataformas suportadas pelo JDK. Nas implementações nativas, até mesmo alguns recursos disponíveis em todos os lugares eram distintamente diferentes quando renderizados na tela. Muitos desenvolvedores de desktop que trabalhavam com Java naqueles primeiros dias brincavam que o slogan “escreva uma vez, execute em qualquer lugar” era na verdade “escreva uma vez, depure em qualquer lugar”. O

pacote Java Swing foi criado para melhorar esse estado lamentável. Embora o Swing não tenha resolvido todos os problemas de entrega de aplicativos multiplataforma, ele tornou possível o desenvolvimento sério de aplicativos de desktop em Java. Você pode encontrar muitos projetos de código aberto de qualidade e até mesmo alguns aplicativos comerciais escritos em Swing.

Na verdade, o IDE que detalhamos emApêndice A, IntelliJ IDEA, é um aplicativo Swing! Ele claramente vai de igual para igual com IDEs nativos em desempenho e usabilidade.1

Se você olhar a documentação do javax.swing2pacote, você verá que ele contém uma infinidade de classes. Você ainda precisará de algumas partes do domínio java.awt original. Existem livros inteiros sobre AWT (*Referência Java AWT*, Zukowski, e no Swing (*Balanço Java*, Loy, et al. e até mesmo em subpacotes do AWT, como gráficos 2D

(*Gráficos Java 2D*, Knudsen, Neste capítulo, nos concentraremos em alguns componentes populares, como botões e campos de texto. Veremos como organizá-los na janela do seu aplicativo e como interagir com eles. Você pode se surpreender com o quão sofisticado seu aplicativo pode ficar com esses tópicos iniciais simples. Se você desenvolver mais desktops depois de ler este livro, também poderá se surpreender com a quantidade de conteúdo de interface gráfica do usuário (GUI, ou apenas UI) disponível para Java. Queremos aguçar seu apetite e ao mesmo tempo reconhecer que há muitas, muitas outras discussões sobre UI que devemos deixar para você descobrir mais tarde. Com isso dito, vamos começar a turnê turbulenta!



Botões, controles deslizantes e campos de texto, meu Deus!

Então, por onde começar? Temos um problema do tipo “o ovo e a galinha”: precisamos discutir as “coisas” para colocar na tela, como os objetos JLabel que usamos

em“OláJava”. Mas também precisamos discutir em que você investe essas coisas. E

onde você coloca essas coisas também merece discussão, pois não é um processo trivial. Na verdade, parece que temos um problema do ovo, da galinha e do brunch.

Pegue uma xícara de café ou uma mimosa e começaremos. Abordaremos primeiro alguns

componentes populares (as “coisas”), depois seus contêineres e, finalmente, o tópico de disposição de seus componentes nesses contêineres. Depois que você conseguir colocar um bom conjunto de widgets na tela, discutiremos como interagir com eles e também como lidar com interfaces de usuário em um mundo multithread.

Hierarquias de componentes

Como discutimos nos capítulos anteriores, as classes Java são projetadas e estendidas de forma hierárquica. JComponent e JContainer ficam no topo da hierarquia de classes Swing, conforme mostrado emFigura 12-1. Não abordaremos essas duas classes com muitos detalhes, mas lembre-se de seus nomes. Você encontrará vários atributos e métodos comuns nessas classes ao ler a documentação do Swing. À medida que você avança em seus esforços de programação, provavelmente desejará criar seu próprio componente. JComponent é um ótimo ponto de partida. Usamos JComponent ao construir nosso componente gráfico Hello emCapítulo 2.

Figura 12-1. Hierarquia de classes Swing parcial (muito parcial)

Estaremos cobrindo a maioria das outras classes mencionadas na hierarquia resumida acima, mas você definitivamente vai querer visitar odocumentação on-linepara ver os muitos componentes que tivemos que deixar de fora.

Arquitetura do controlador Model View

Na base da noção de “coisas” do Swing está um padrão de design conhecido como Model View Controller (MVC). Os autores do pacote Swing trabalharam duro para

aplicar consistentemente esse padrão para que, quando você encontrar novos

componentes, seu comportamento e uso pareçam familiares. A arquitetura MVC visa compartimentar o que você vê (a visão) do estado dos bastidores (o modelo) e da coleção de interações (o controlador) que alteram essas partes. Essa separação de preocupações permite que você se concentre em acertar cada peça. O tráfego de rede pode atualizar o modelo nos bastidores. A visualização pode ser sincronizada em intervalos regulares que parecem suaves e responsivos ao usuário. MVC fornece uma estrutura poderosa, porém gerenciável, para usar na construção de qualquer aplicativo de desktop.

Ao examinarmos nossa pequena seleção de componentes, destacaremos o modelo e os elementos da vista. Em seguida, entraremos em mais detalhes sobre os controladores

em“Eventos”. Se você acha a noção de padrões de programação intrigante, *Padrões de*

Projeto: Elementos de Software Orientado a Objetos Reutilizáveis(Addison-Wesley) de Gamma, Helm, Johnson e Vlissides (a renomada Gangue dos Quatro) é a obra clássica.

Para obter mais detalhes sobre o uso do padrão MVC especificamente no Swing, consulte o capítulo introdutório do *Balanço Java*, por Loy et al.

Etiquetas e botões

O componente de UI mais simples não é surpreendentemente um dos mais populares.

Etiquetas são usadas em todos os lugares para indicar funcionalidade, exibir status e chamar o foco. Usamos um rótulo para nosso primeiro aplicativo gráfico emCapítulo 2.

Usaremos muito mais rótulos à medida que continuarmos a construir programas mais interessantes.

O componente JLabel é uma ferramenta versátil. Vejamos alguns exemplos de como usar JLabel e personalizar seus diversos atributos. Começaremos revisitando nosso programa “Hello, Java” com alguns ajustes preparatórios:

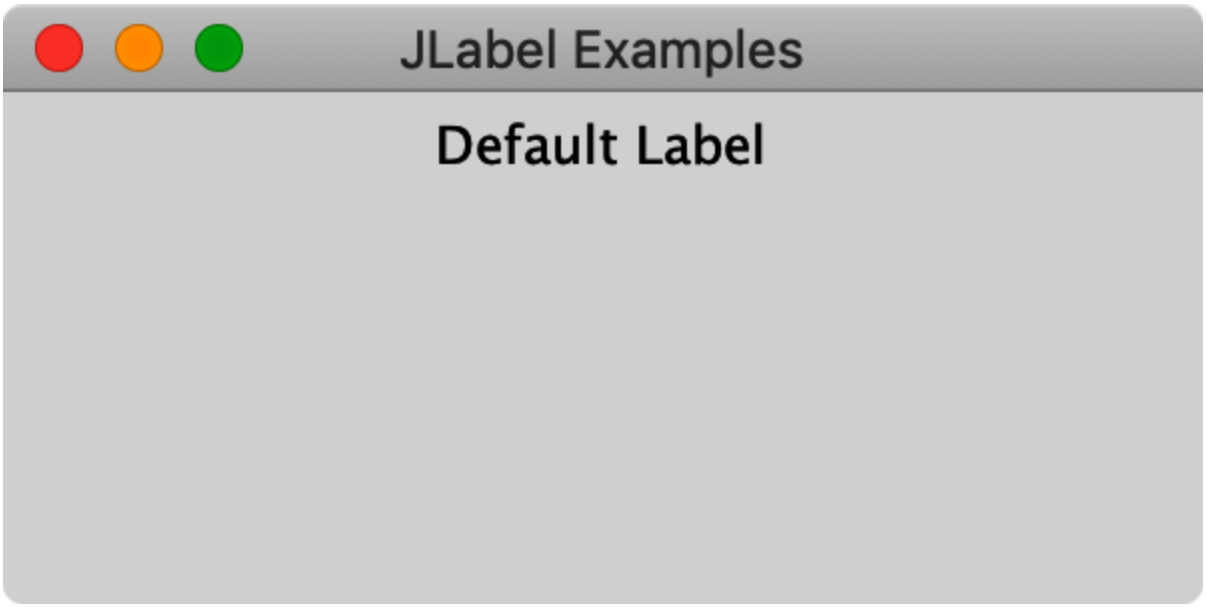
```
importar javax.swing.*;

importar java.awt.*;

rótulos de classe pública {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Exemplos JLabel");

frame.setLayout(novo FlowLayout());

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setSize(300, 150);

JLabel básico = new JLabel("Etiqueta Padrão");

quadro.add(básico);

frame.setVisible (verdadeiro);

}

}
```

Resumidamente, as partes interessantes são:

❶

❷

❸



Configurando o gerenciador de layout para uso pelo quadro.

Definir a ação realizada ao utilizar o botão “fechar” do sistema operacional (neste caso, o ponto vermelho no canto superior esquerdo da janela). A ação que selecionamos aqui sai do aplicativo.

Criando nossa etiqueta simples e adicionando-a à moldura.

Você declara e inicializa o rótulo e depois o adiciona ao quadro. Isso deveria ser familiar. O que provavelmente é novo é o uso de uma instância FlowLayout, que nos ajuda a produzir a captura de tela mostrada emFigura 12-2.

Figura 12-2. Um JLabel único e simples

Veremos os gerenciadores de layout com muito mais detalhes em“Contêineres e

Layouts”, mas precisamos de algo para nos tirar do papel e que também nos permita adicionar vários componentes a um único contêiner. A classe FlowLayout preenche um contêiner centralizando horizontalmente os componentes na parte superior, adicionando da esquerda para a direita até que a “linha” fique sem espaço e, em seguida, continuando em uma nova linha abaixo. Esse tipo de arranjo não será muito útil em aplicações maiores, mas é ideal para colocar diversas coisas na tela rapidamente.

Vamos provar isso adicionando mais alguns rótulos ao quadro. Confira os resultados mostrados emFigura 12-3:

rótulos de classe pública {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Exemplos JLabel");

frame.setLayout(novo FlowLayout());

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

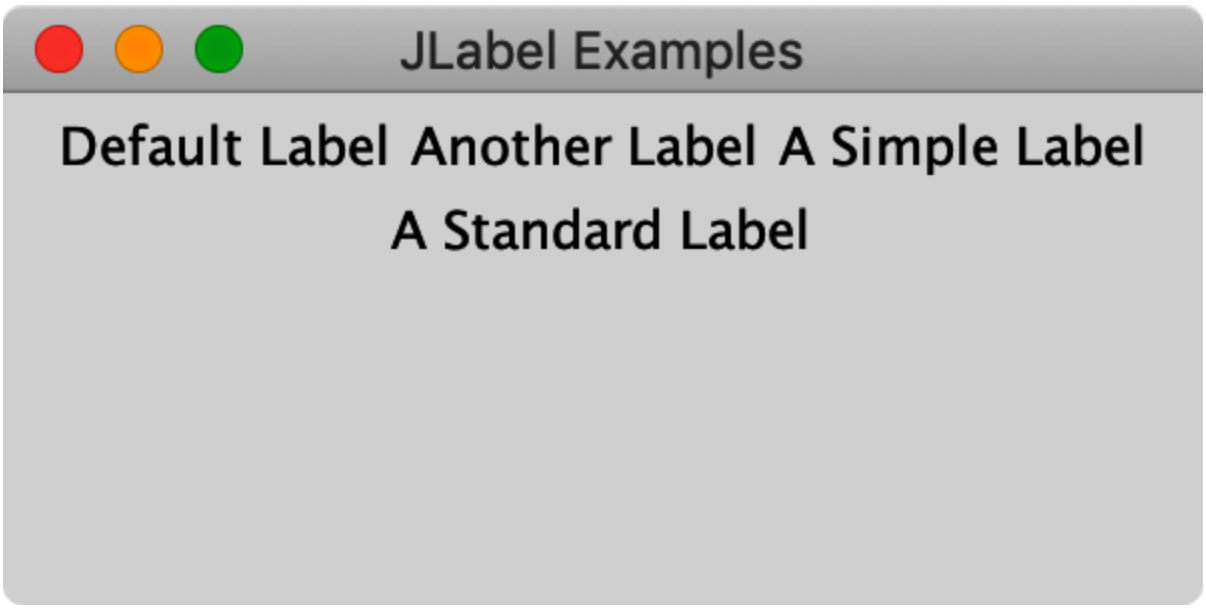
frame.setSize(300, 150);

JLabel básico = new JLabel("Etiqueta Padrão");

JLabel outro = new JLabel("Outro rótulo");

JLabel simples = new JLabel("Um rótulo simples");

Padrão JLabel = new JLabel("Um rótulo padrão");




```
quadro.add(básico);

quadro.add(outro);

quadro.add(simples);

frame.add(padrão);

frame.setVisible (verdadeiro);

}

}
```

Figura 12-3. Vários objetos JLabel básicos

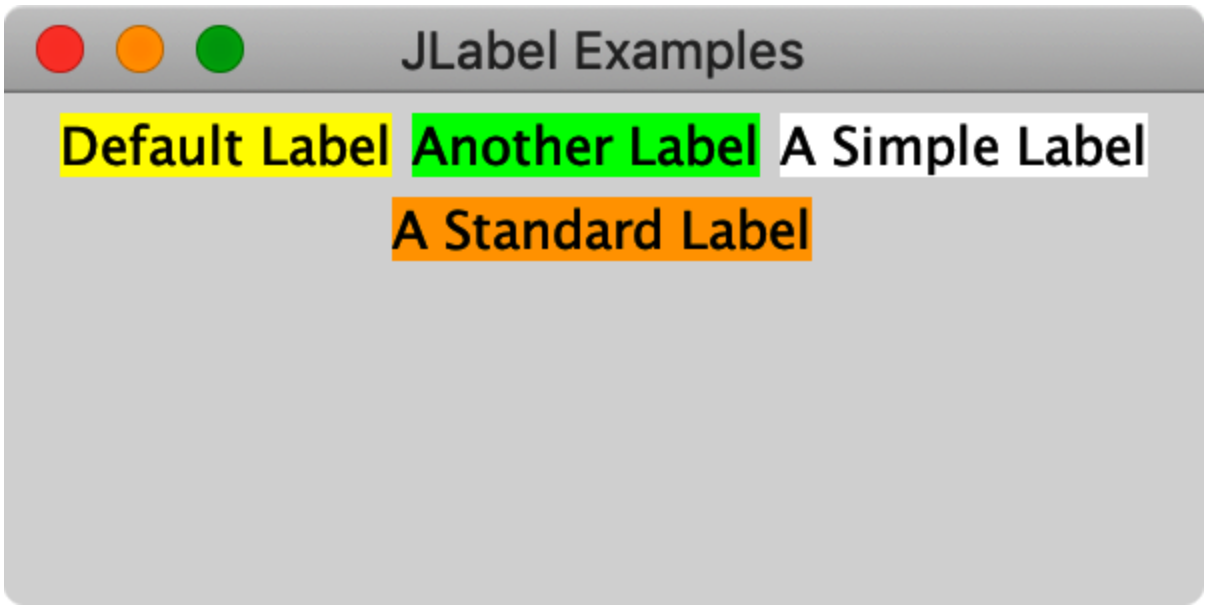
Legal, certo? Novamente, esse layout simples não se destina à maioria dos tipos de conteúdo encontrados em aplicativos de produção, mas é definitivamente útil quando você começa. Mais um ponto sobre layouts que queremos fazer, pois você encontrará essa ideia mais tarde: FlowLayout também trata do tamanho dos rótulos. Isso pode ser difícil de perceber neste exemplo porque os

rótulos têm um fundo transparente por padrão. Se importarmos a classe java.awt.Color, podemos usar essa classe para ajudar a torná-los opacos e dar-lhes uma cor de fundo específica:

```
JLabel básico = new JLabel("Etiqueta Padrão");

básico.setOpaque(true);

basic.setBackground(Color.AMARELO);

JLabel outro = new JLabel("Outro rótulo");

outro.setOpaque(true);

outro.setBackground(Color.GREEN);

quadro.add(básico);

quadro.add(outro);

// outra configuração de quadro
```

Se fizermos o mesmo para todos os nossos rótulos, podemos agora ver seus verdadeiros tamanhos e as lacunas entre eles.Figura 12-4. Mas se pudermos controlar a cor de fundo dos rótulos, o que mais podemos fazer? Podemos mudar a cor do primeiro plano? (Sim.) Podemos mudar a fonte? (Sim.) Podemos mudar o

alinhamento? (Sim.) Podemos adicionar ícones? (Sim.) Podemos criar rótulos autoconscientes que eventualmente construam a Skynet e provoquem o fim da humanidade? (Talvez, mas provavelmente não, e certamente não facilmente. Ainda bem.)Figura 12-5mostra alguns desses possíveis ajustes.



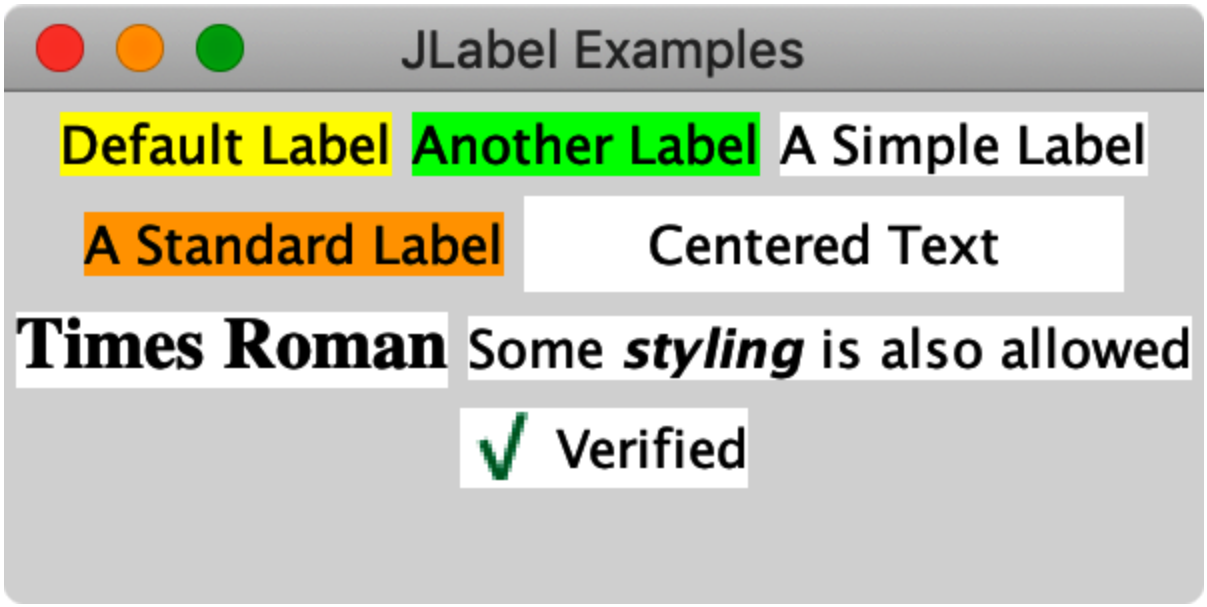


Figura 12-4. Etiquetas opacas e coloridas

Figura 12-5. Mais rótulos com opções mais sofisticadas

E aqui está o código-fonte relevante que criou essa variedade adicional:

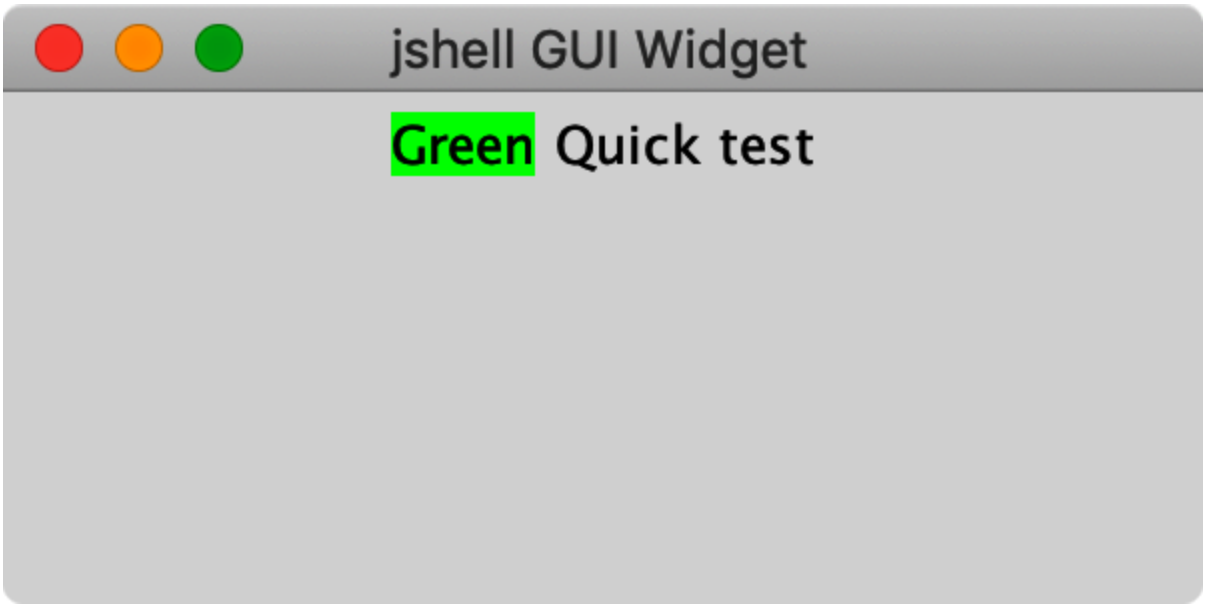
// uma etiqueta branca com tamanho forçado e texto centralizado dentro JLabel centralizado = new

```
JLabel("Texto Centralizado", JLabel.CENTER);
centered.setPreferredSize(nova Dimensão(150, 24));

centrado.setOpaque(true);

centralizado.setBackground(Color.WHITE);

// uma etiqueta branca com uma fonte alternativa maior

JLabel vezes = new JLabel("Times Roman");

vezes.setOpaque(true);

tempos.setBackground(Color.WHITE);

times.setFont(new Font("TimesRoman", Font.BOLD, 18));

// uma etiqueta em branco usando HTML embutido para
estilização

JLabel estilizado = new JLabel("<html>Alguns <b>
<i>estilos</i></b>"

+ " também é permitido</html>");

estilizado.setOpaque(true);

estilizado.setBackground(Color.WHITE);

// um rótulo com um ícone e um texto

Ícone JLabel = novo JLabel("Verificado",

novo ImageIcon("ch10/examples/check.png"),
JLabel.LEFT);

icon.setOpaque(true);
```

icon.setBackground(Color.BRANCO);

// finalmente, adicionamos todos os nossos novos rótulos ao quadro

frame.add(centralizado);



frame.add(vezes);

frame.add (estilizado);

quadro.add(ícone);

Usamos algumas outras classes para ajudar, como java.awt.Font e

javax.swing.ImageIcon. Poderíamos analisar muitas outras opções, mas precisamos examinar alguns outros componentes. Se você quiser brincar com esses rótulos e experimentar mais opções que você vê na documentação Java, tente importar um auxiliar que construímos para jshell e brincar.3Os resultados de nossas poucas linhas são mostrados emFigura 12-6:

```
$ javac ch12/examples/Widget.java

$jshell

| Bem-vindo ao JShell - Versão 21-ea

| Para uma introdução digite: /help intro

jshell> importar javax.swing.*

jshell> importar java.awt.*

jshell> importar ch12.examples.Widget

jshell> Widget w = novo Widget()

w ==>
ch10.Widget[frame0,0,23,300x150,layout=java.awt.B ...
capaz = true]

jshell> JLabel rótulo1 = new JLabel("Verde")

rótulo1 ==>
javax.swing.JLabel[,0,0,0x0,inválido,alignmentX=0. ... íon
=

CENTRO]

jshell> label1.setOpaque(true)

jshell> label1.setBackground(Color.GREEN)

jshell> w.add(rótulo1)

$8 ==> javax.swing.JLabel[,0,0,0x0,...]

jshell> w.add(new JLabel("Teste rápido"))
```

$9 ==> javax.swing.JLabel[,0,0,0x0,...]

Figura 12-6. Usando nossa classe Widget em jshell

Esperamos que você veja como é fácil criar um rótulo (ou outros componentes, como os botões que exploraremos a seguir) e ajustar seus parâmetros de forma interativa.

Essa é uma ótima maneira de se familiarizar com os blocos de construção que você tem à sua disposição para criar aplicativos de desktop Java. Se você usa muito nosso widget, poderá achar seu método reset() útil. Este método remove todos os componentes atuais e atualiza a janela para que você possa recomeçar rapidamente.

Botões

O outro componente quase universal necessário para aplicativos gráficos é o botão. A classe JButton é o seu botão preferido no Swing. (Você também encontrará outros tipos de botões populares, como JCheckbox e JToggleButton, na documentação.) A criação de um botão é muito semelhante à criação de um rótulo, conforme mostrado

emFigura 12-7.

```
importar javax.swing.*;

importar java.awt.*;

botões de classe pública {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Exemplos JButton");

frame.setLayout(novo FlowLayout());

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setSize(300, 150);

JButton básico = new JButton("Experimente!");

quadro.add(básico);

frame.setVisible (verdadeiro);

}

}
```

Figura 12-7. Um JButton simples

Você pode controlar as cores, o alinhamento do texto e da imagem, a fonte e assim por diante dos botões, da mesma forma que faz com os rótulos. A diferença, claro, é que você pode clicar em um botão e reagir a esse clique em seu programa, enquanto os rótulos são, em sua maioria, exibições estáticas. Tente executar este exemplo e clicar no botão. Ele deve mudar de cor e parecer “pressionado”, embora ainda não execute

nenhuma outra função em nosso programa. Queremos passar por mais alguns componentes antes de abordar a noção de “reagir” a um clique de botão (um “evento”

na linguagem Swing), mas você pode pular p[ara“Eventos”se voc](#)ê não pode esperar!

Componentes de texto

Seria impossível imaginar hoje um aplicativo desktop ou web sem os campos de entrada de texto. Esses elementos de entrada permitem a entrada de informações de forma livre e são quase onipresentes em formulários on-line. Você pode obter nomes, endereços de e-mail, números de telefone e números de cartão de crédito. Você pode fazer tudo isso em idiomas que compõem seus personagens, ou em outros que são lidos da direita para a esquerda. Swing possui três grandes componentes de texto: JTextField, JTextArea e JTextPane; todos estendidos de um pai comum,

JTextComponent. JTextField é um campo de texto clássico destinado à entrada breve, de uma única palavra ou de uma única linha. JTextArea permite muito mais entrada espalhada por várias linhas. JTextPane é um componente especializado destinado à edição de rich text.

Campos de texto

Vejamos um exemplo de entrada de texto em execução em nosso aplicativo simples e fluido. Reduziremos as coisas a dois rótulos e campos de texto correspondentes:
importar javax.swing.*;

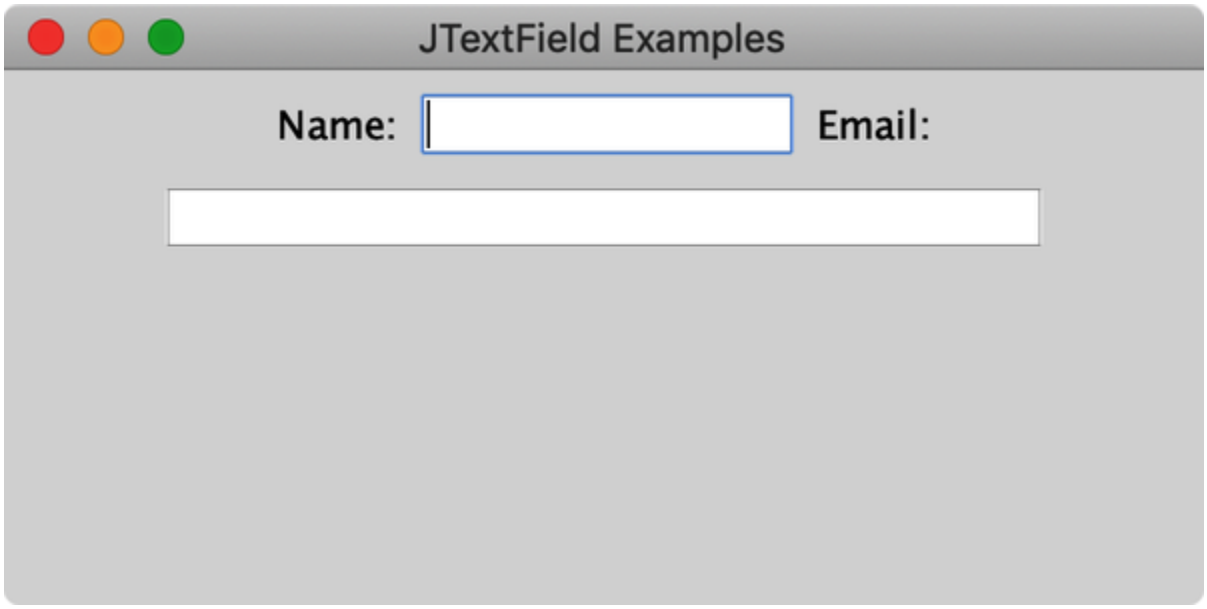
importar java.awt.*;

classe pública TextInputs {

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

Quadro JFrame = new JFrame("Exemplos JTextField");

frame.setLayout(novo FlowLayout());

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setSize(400, 200);

JLabel nomeLabel = new JLabel("Nome:");

JTextField nomeCampo = new JTextField(10);

JLabel emailLabel = new JLabel("E-mail:");

JTextField emailField = new JTextField(24);

frame.add(nomeLabel);

frame.add(nomeCampo);

frame.add(emailLabel);

frame.add(emailField);

frame.setVisible (verdadeiro);

}

}
```

Aviso emFigura 12-8que o tamanho de um campo de texto é determinado pelo número de colunas que especificamos em seu construtor. Essa não é a única maneira de inicializar um campo de texto, mas é útil quando não há outros mecanismos de layout que determinem a largura do campo. (Aqui, o FlowLayout

falhou um pouco – o rótulo “Email:” não ficou na mesma linha do campo de texto do email. Corrigiremos isso assim que aprendermos mais sobre layouts.) Vá em frente e digite algo! Você



pode inserir e excluir texto; destaque coisas dentro do campo com o mouse; e recorte, copie e cole conforme

esperado.

Figura 12-8. Rótulos simples e JTextFields

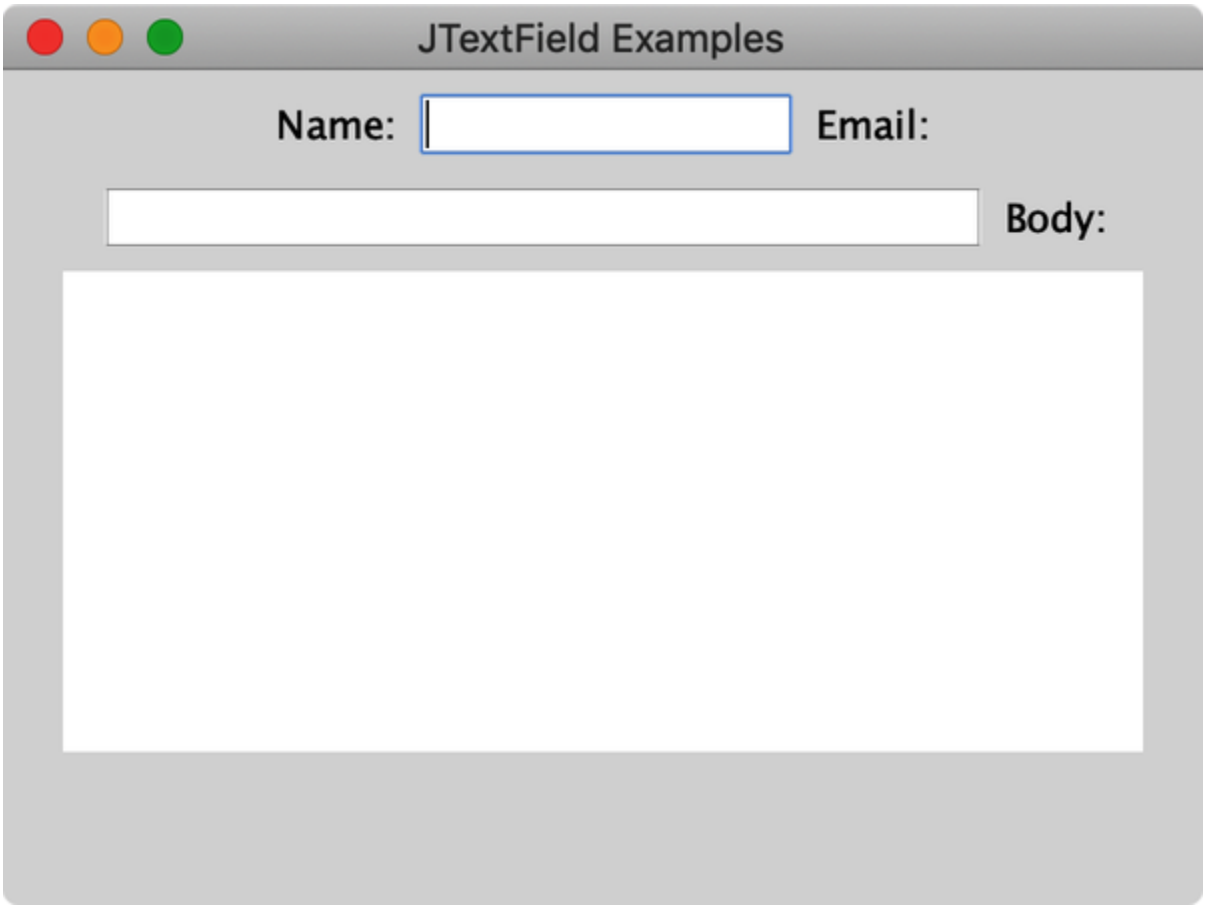
Se você adicionar um campo de texto ao nosso aplicativo de demonstração em jshell, conforme mostrado em Figura 12-9, você pode chamar seu método getText() para ver se o conteúdo está realmente disponível para você.

Figura 12-9. Recuperando o conteúdo de um JTextField

```
jshell> w.reset()

jshell> JTextField emailField = novo JTextField(15)

emailField ==> javax.swing.JTextField[,0,0,0x0, ...
lignment=LEADING]

jshell> w.add(new JLabel("E-mail:"))

$12 ==> javax.swing.JLabel[,0,0,0x0, ...
situação=CENTRO]

jshell> w.add(emailField)

$ 13 ==> javax.swing.JTextField[,0,0,0x0, ...
lignment=LEADING]

// Insira um endereço de exemplo, digitamos "
me@some.company "

jshell> emailField.getText()

$ 14 ==> " eu@alguma.empresa "
```

Observe que a propriedade text é leitura-gravação. Você pode chamar setText() em seu campo de texto para alterar seu conteúdo programaticamente. Isso pode ser

ótimo para definir valores padrão, formatar automaticamente itens como números de telefone ou preencher previamente um formulário com informações coletadas pela rede. Experimente em jshell.



Áreas de texto

Quando você precisar de espaço para mais do que palavras simples ou até mesmo entradas de URL longas, provavelmente recorrerá ao JTextArea para fornecer ao usuário várias linhas de espaço de entrada. Você pode criar uma área de texto vazia com um construtor semelhante ao JTextField. Para JTextArea, você especifica o número de linhas além do número de colunas. Dê uma olhada no código para adicionar uma área de texto ao

nosso aplicativo de demonstração de entrada de texto:
JLabel bodyLabel = new JLabel("Corpo:");

JTextArea bodyArea = new JTextArea(10,30);

frame.add(bodyLabel);

frame.add(bodyArea);

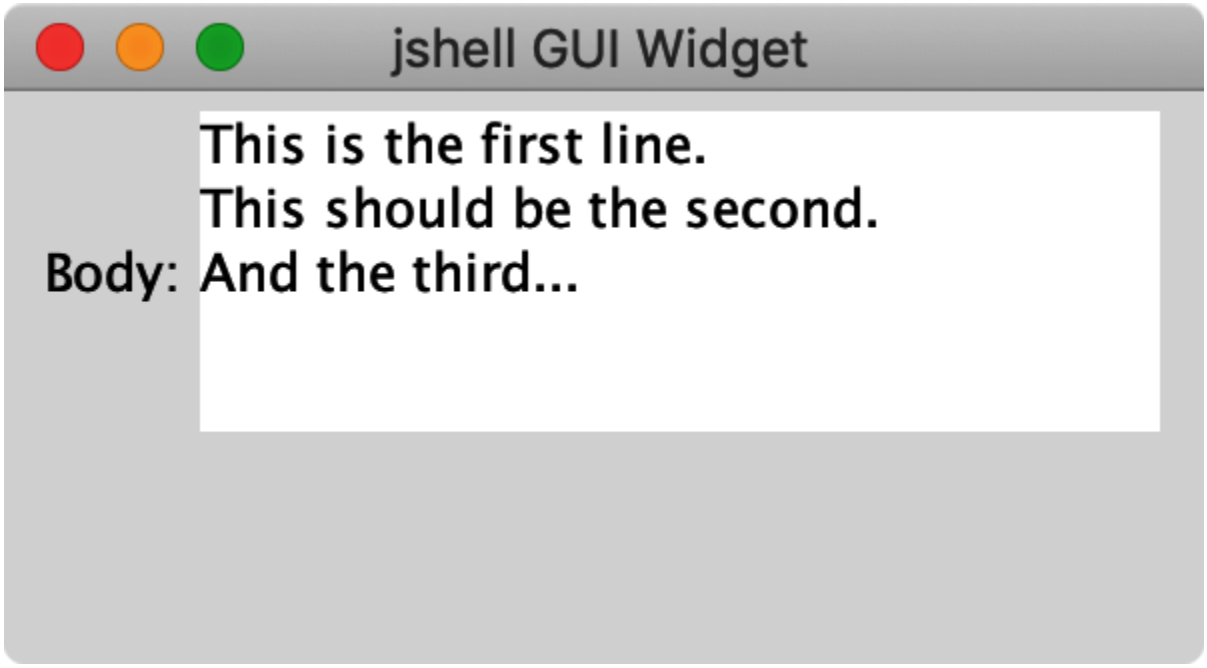
Os resultados são mostrados em Figura 12-10. Você pode ver que temos espaço para várias linhas de texto. Vá em frente e execute esta nova versão e experimente você mesmo. O que acontece se você digitar além do final de uma linha? O que acontece quando você pressiona a tecla Return? Você percebe os comportamentos com os quais está familiarizado? Você ainda terá acesso ao seu conteúdo da mesma forma que faz com um campo de texto.

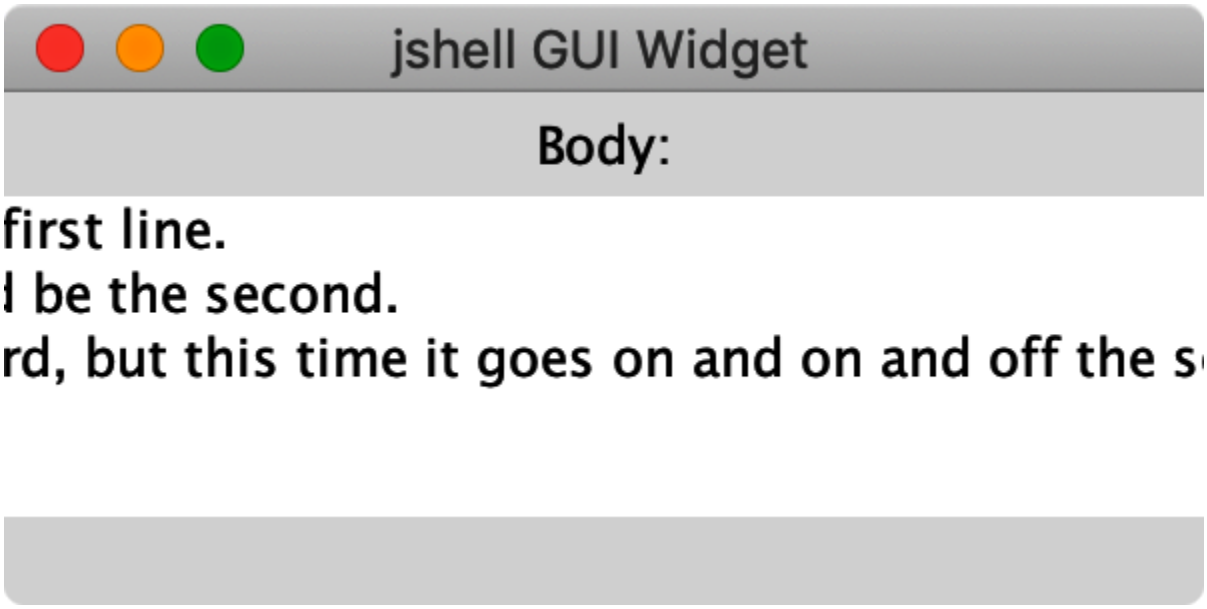
Figura 12-10. Adicionando um JTextArea

Vamos adicionar uma área de texto ao nosso widget em jshell para que possamos brincar com suas propriedades:

```
jshell> w.reset()

jshell> w.add(new JLabel("Corpo:"))

$ 16 ==> javax.swing.JLabel[,0,0,0x0, ... ition=CENTER]

jshell> JTextArea bodyArea = new JTextArea(5,20)

bodyArea ==> javax.swing.JTextArea[,0,0,0x0, ...
palavra=false,wrap=false

]

jshell> w.add(bodyArea)
```

$ 18 ==> javax.swing.JTextArea[,0,0,0x0, ...
lse,wrap=false]





jshell> bodyArea.getText()

$19 ==> "Esta é a primeira linha.\nEsta deve ser a
segunda.\nE a terceira

..."

Ótimo! Podemos ver que a tecla Return que digitamos para produzir nossas três linhas

emFigura 12-11é codificado como o caractere \n na string que recuperamos.

Figura 12-11. Recuperando o conteúdo de um JTextArea

Mas o que acontece se você tentar digitar uma frase longa e contínua que ultrapasse o final da linha? Você pode obter uma área de texto estranha que se expandiu até o tamanho da sua janela e além, conforme mostrado emFigura 12-12.

Figura 12-12. Uma linha excessivamente longa em um JTextArea simples

Podemos corrigir esse comportamento incorreto de dimensionamento observando um par de propriedades de JTextArea, mostradas emTabela 12-1.

Tabela 12-1. Quebrar propriedades de JTextArea

Propriedade

Padrão Descrição

linhaWrap

falso

Se as linhas mais longas que a tabela devem ser

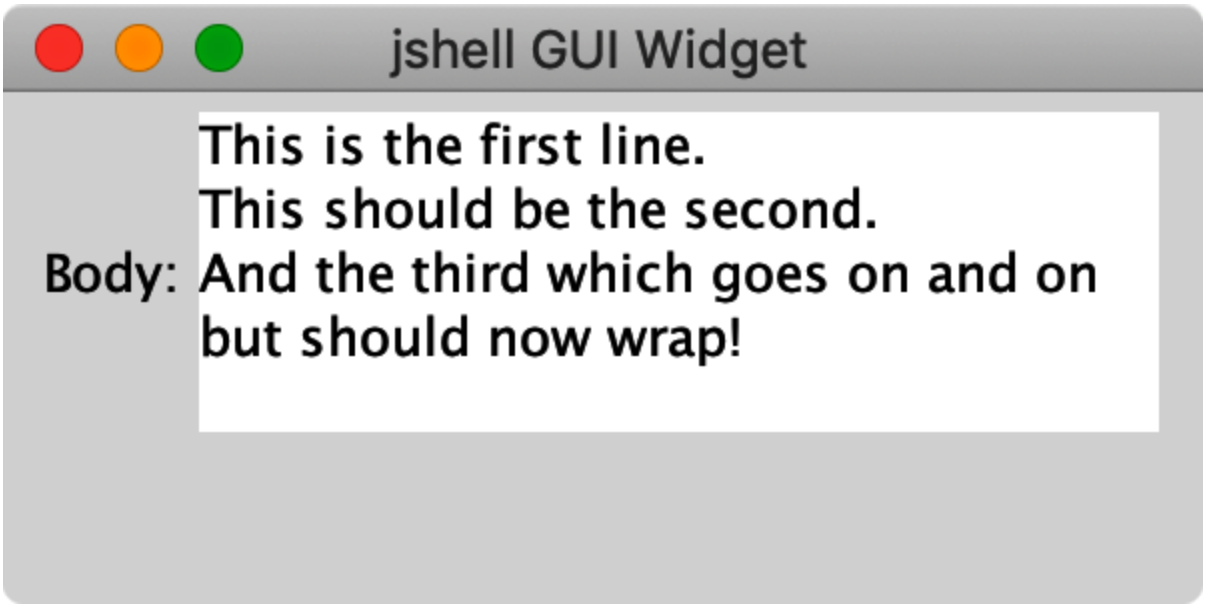
quebradas

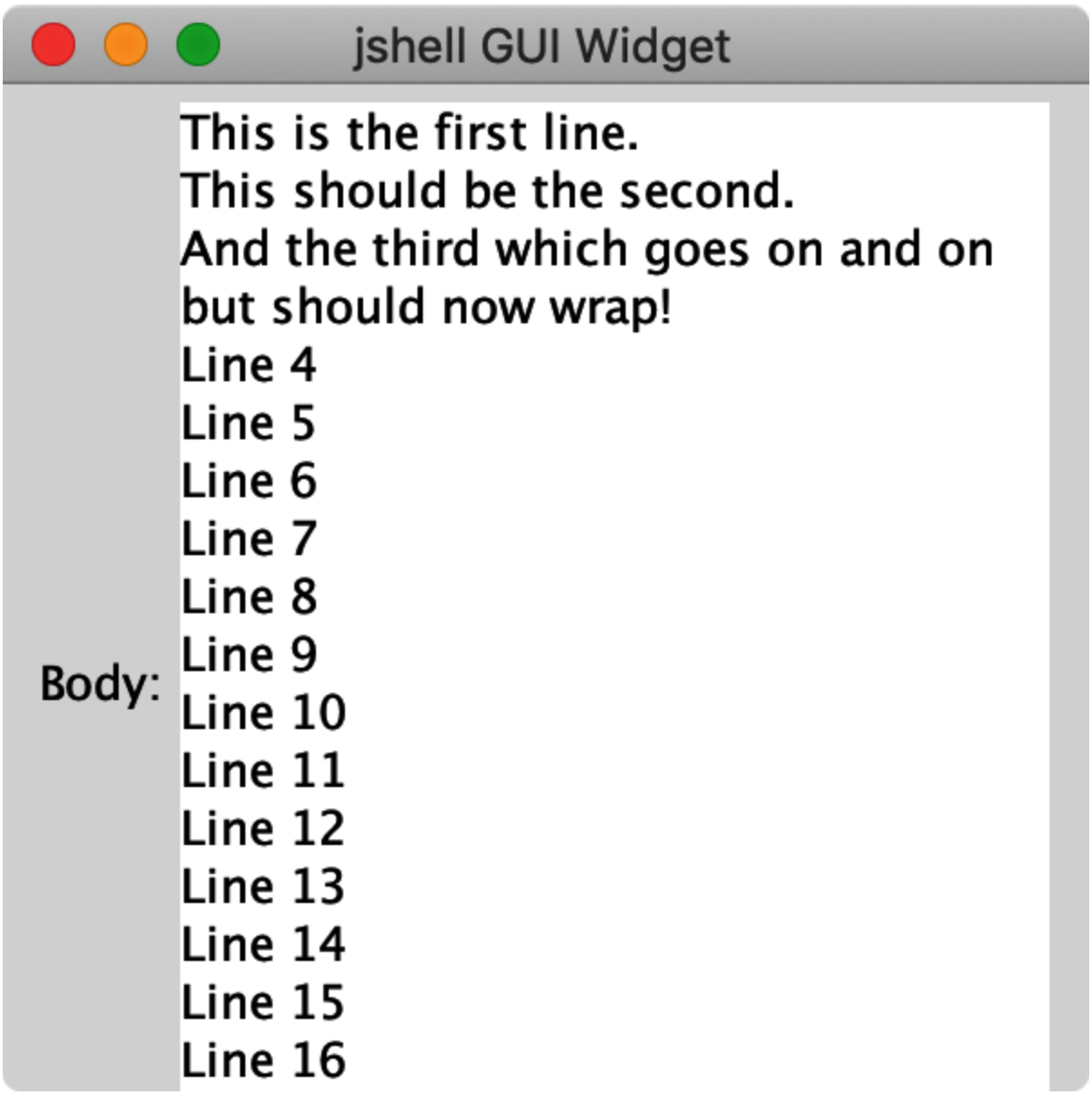
wrapStyleWord falso

Se as linhas quebrarem, se as quebras de linha devem

estar nos limites das palavras ou dos caracteres

Então, vamos começar do zero e ativar a quebra automática de palavras. Podemos usar setLineWrap(true) para garantir que o texto seja quebrado. Mas isso provavelmente não é suficiente. Adicionaremos uma chamada para





setWrapStyleWord(true) para garantir que a área de texto não apenas quebre as palavras no meio. Isso deve ser semelhante aFigura 12-13.

Figura 12-13. Uma linha de quebra em um JTextArea simples

Você pode tentar isso no jshell ou em seu próprio aplicativo. Ao recuperar o texto do objeto bodyArea, você

não deverá ver uma quebra de linha (\n) na linha três entre o segundo “on” e o “but”.

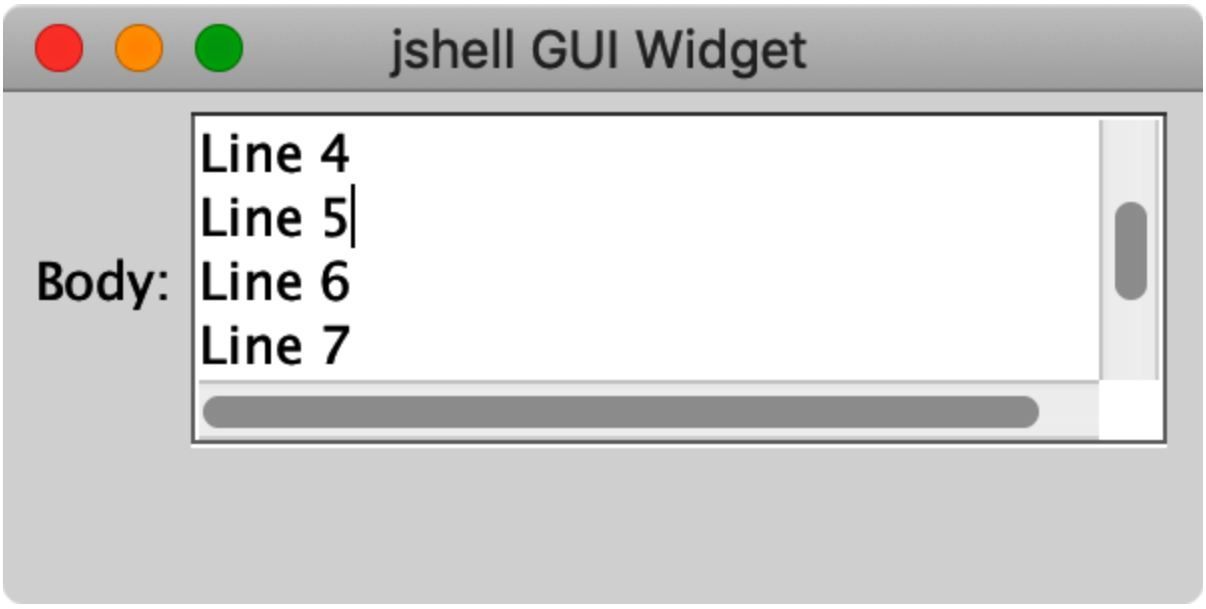
Rolagem de texto

O que acontece se tivermos muitas linhas? Por si só, JTextArea faz aquele truque estranho de “crescer até não poder”, como mostrado emFigura 12-14.

Figura 12-14. Muitas linhas em um JTextArea simples

Para corrigir esse problema, precisamos chamar algum suporte de um componente auxiliar padrão do Swing: JScrollPane. Este é um contêiner de uso geral que facilita a apresentação de grandes componentes em espaços confinados. Para mostrar como isso é fácil, vamos corrigir nossa área de texto:4

jshell> w.remove(bodyArea); // Então podemos começar com uma nova área de texto



jshell> bodyArea = new JTextArea(5,20)

bodyArea ==> javax.swing.JTextArea[,0,0,0x0,inval...
palavra=false,wrap=f alse]

jshell> w.add(novo JScrollPane(bodyArea))

$ 17 ==> javax.swing.JScrollPane[,47,5,244x84, ...
ortBorder=]

Você pode ver emFigura 12-15que a área de texto não ultrapasse os limites do quadro.

Você também pode ver as barras de rolagem padrão nas laterais e na parte inferior. Se você só precisa de uma rolagem simples, pronto! Mas, como a maioria dos outros componentes do Swing, o JScrollPane possui muitos detalhes que você pode ajustar conforme necessário. Não cobriremos a maioria deles aqui, mas queremos mostrar como lidar com uma configuração comum para áreas de texto: quebra de linha (quebra de palavras) com rolagem vertical - ou seja, sem rolagem horizontal.

Figura 12-15. Muitas linhas em um JTextArea incorporado em um JScrollPane JLabel bodyLabel = new JLabel("Corpo:");

JTextArea bodyArea = new JTextArea(10,30);

bodyArea.setLineWrap(true);

bodyArea.setWrapStyleWord(true);

JScrollPane bodyScroller = novo JScrollPane(bodyArea);

bodyScroller.setHorizontalScrollBarPolicy(

JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

bodyScroller.setVerticalScrollBarPolicy(

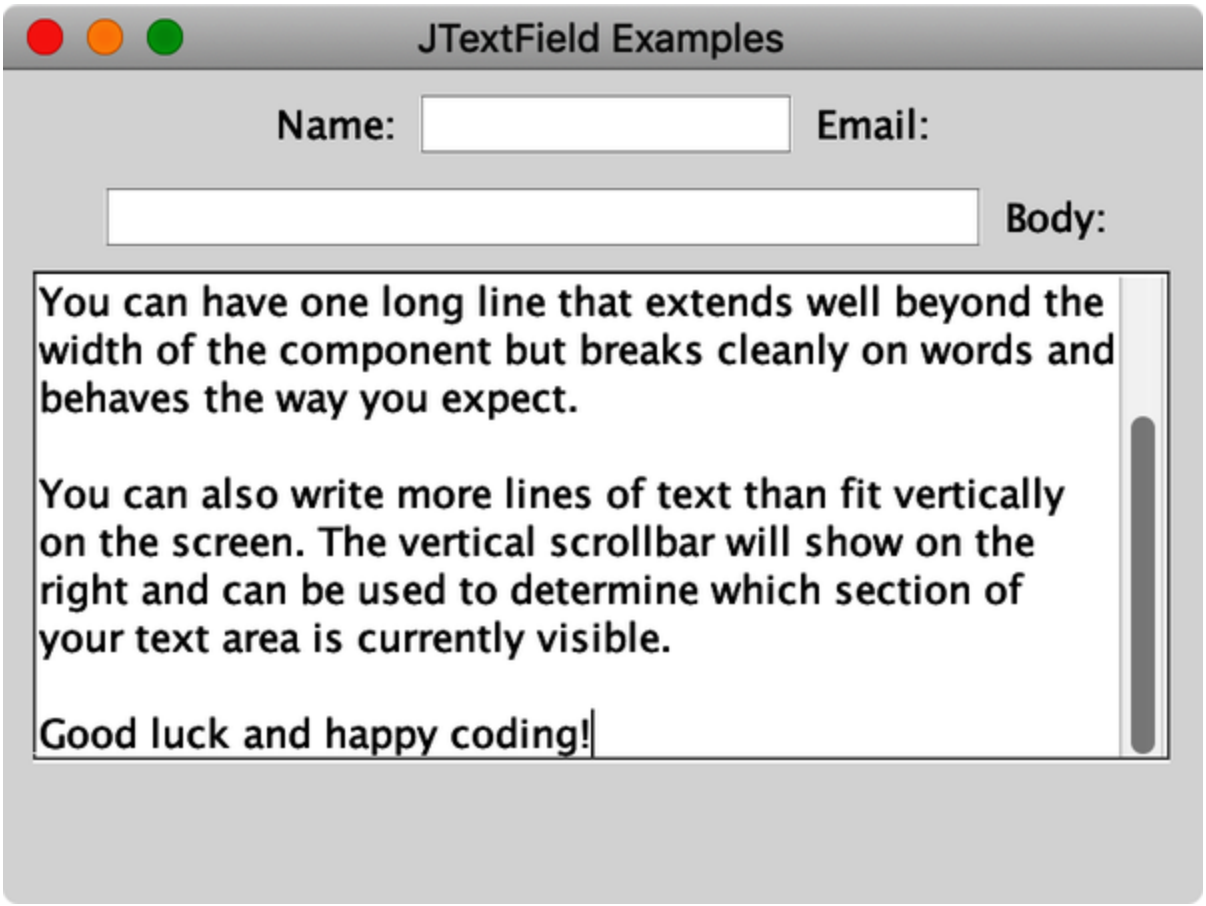
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

frame.add(bodyLabel);

// observe que não adicionamos bodyArea, já está em bodyScroller

frame.add(bodyScroller);

Você deve acabar com uma área de texto como a mostrada emFigura 12-16.



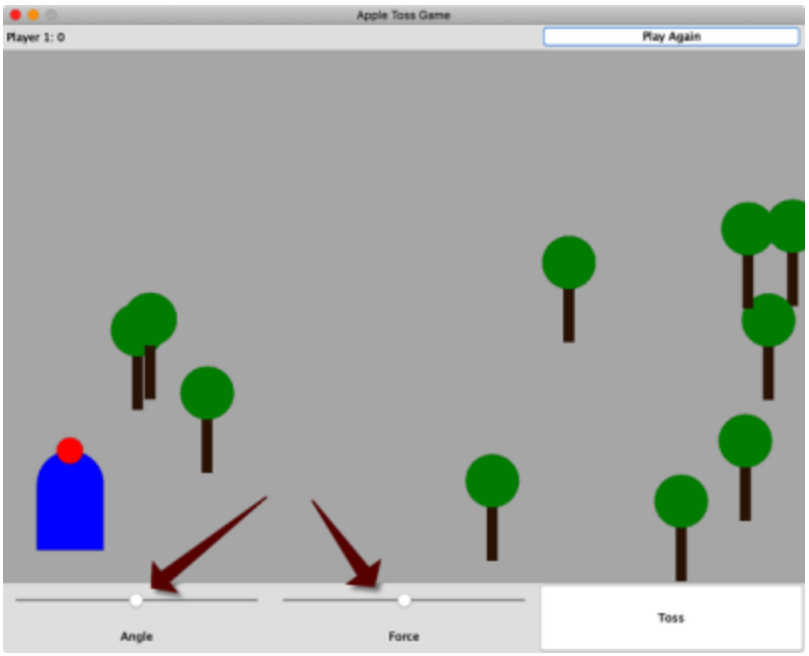


Figura 12-16. Um JTextArea bem formado em um JScrollPane

Viva! Agora você tem uma ideia dos componentes Swing mais comuns, incluindo rótulos, botões e campos de texto. Mas nós realmente apenas arranhamos a superfície desses componentes. Consulte a documentação Java e experimente cada um desses componentes no jshell ou em seus próprios miniaplicativos.

Ficar confortável com o design da UI requer prática. Recomendamos que você procure outros livros e recursos on-line se estiver criando aplicativos de desktop, mas nada supera o tempo gasto no teclado.

Outros componentes

Se você já consultou a documentação do pacote javax.swing, sabe que várias dezenas de outros componentes estão disponíveis. Dentro dessa grande lista, há alguns que queremos destacar.5

JSlider

Os controles deslizantes são um componente de entrada bacana e eficiente quando você deseja que o usuário escolha entre uma variedade de valores: por exemplo, itens como seletores de tamanho de fonte, seletores de cores e seletores de zoom. Os controles deslizantes são perfeitos para os valores de ângulo e força que precisamos em nosso jogo de lançamento de maçãs. Nossos ângulos variam de 0 a 180, e nosso valor de força varia de 0 a 20 (um máximo arbitrário). Figura 12-17 mostra esses controles deslizantes no lugar (ignore como alcançamos o layout por enquanto).

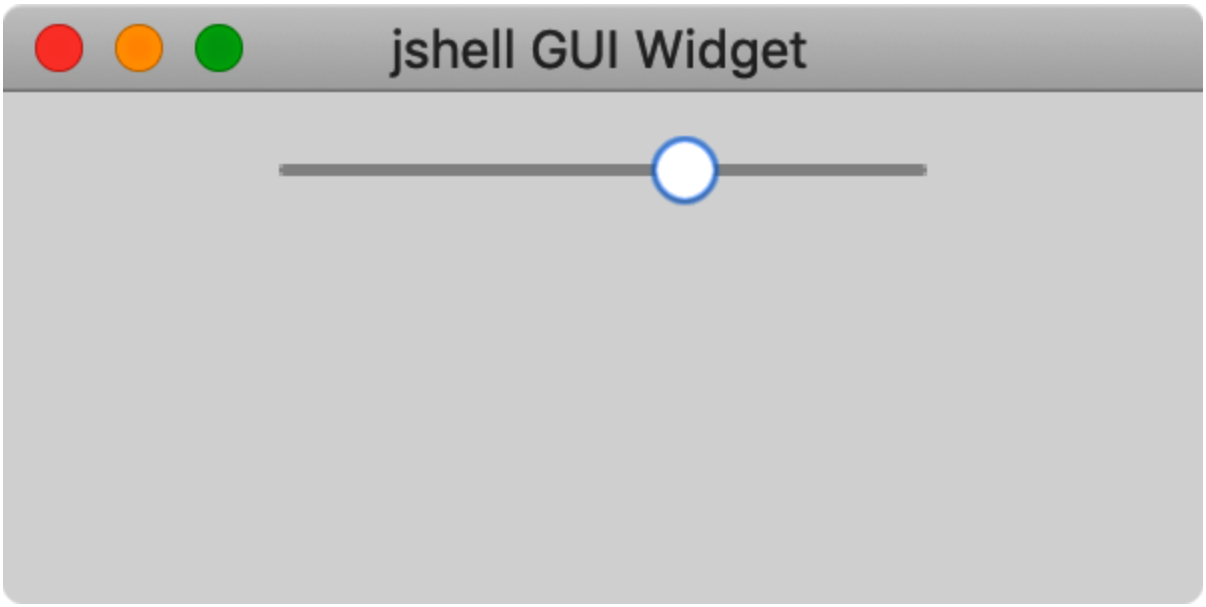


Figura 12-17. Usando JSlider em nosso jogo de lançar maçãs

Para criar um novo controle deslizante, você fornece três valores: o mínimo (0 para nosso controle deslizante de ângulo), o máximo (180) e o valor inicial (comece no meio para o jogo em 90). Você pode adicionar esse controle deslizante ao nosso playground jshell assim:

```
//reinicia o widget

jshell> w.reset()

jshell> controle deslizante JSlider = novo JSlider(0, 180,
90);

controle deslizante ==> javax.swing.JSlider[,0,0,0x0, ...
ks=false,snapTo Value=true]

jshell> w.add(controle deslizante)

$20 ==> javax.swing.JSlider[,0,0,0x0, ... alue=true]
```

Mova o controle deslizante como você vê emFigura 12-18e observe seu valor atual usando o método getValue():

```
jshell> slider.getValue()

$ 21 ==> 112
```

Figura 12-18. Um JSlider simples em jshell

Em“Eventos”, veremos como receber esses valores em tempo real conforme o usuário os altera.

Os construtores JSlider usam números inteiros para os valores mínimo e máximo, e getValue() retorna um número inteiro. Se você precisar de valores fracionários, isso cabe a você. O controle deslizante de força em nosso jogo, por exemplo, se beneficiaria ao suportar mais de 21 níveis distintos. Podemos resolver isso construindo o controle deslizante com um intervalo maior de números inteiros e, em seguida, dividindo o valor atual por um fator de escala apropriado:

```
jshell> JSlider força = novo JSlider(0, 200, 100)
```

força ==> javax.swing.JSlider[,0,0,0x0, ...
ks=false,snapToValue=true]

jshell> w.add(força)

$ 23 ==> javax.swing.JSlider[,0,0,0x0,inválido ...
alue=true]

jshell>force.getValue()

$ 24 ==> 68

jshell> float minhaForce = force.getValue() / 10.0f;

minhaForça ==> 6,8

JList

Se você tiver um conjunto discreto de valores, mas esses valores não forem um intervalo de números simples e contíguo, o elemento da interface do usuário “lista” é uma ótima escolha. JList é a implementação Swing deste tipo de entrada. Você pode configurá-lo para permitir

seleções únicas ou múltiplas e, se se aprofundar nos recursos do Swing, poderá produzir visualizações personalizadas que exibem os itens da sua lista com informações ou detalhes extras. (Por exemplo, você pode criar listas de ícones, ou ícones e texto, ou texto multilinha, e assim por diante.) Ao contrário dos outros componentes que vimos até agora, o JList requer um pouco mais de informações para começar. Para criar um componente de lista útil, você precisa usar um dos construtores que pega os dados que você pretende mostrar. O

construtor mais simples aceita uma matriz Object. Embora você possa passar um array de objetos de qualquer tipo, o comportamento padrão do JList será mostrar a saída do método toString() dos seus objetos na lista. Usar um array de objetos String é muito comum e produz os resultados esperados.Figura 12-19mostra uma lista simples de cidades.

Figura 12-19. Uma JList simples de quatro cidades em jshell

```
jshell> w.reset()

jshell> String[] cidades = new String[] { "Atlanta",
"Boston",

...> "Chicago", "Denver" };

cidades ==> String[4] { "Atlanta", ..., "Denver" }

jshell> JList cidadeList = new JList<String>(cidades);

cityList ==> javax.swing.JList[,0,0,0x0, ...entation=0]

jshell> w.add(listacidade)
```

$ 29 ==> javax.swing.JList[,0,0,0x0,inválido... ação=0]

Usamos as mesmas informações de tipo <String> com o construtor que usamos ao criar objetos de coleção parametrizados, como ArrayList (consulte“Limitações de tipo”). Como o Swing foi adicionado bem antes dos genéricos, você pode encontrar

exemplos online ou em livros que não adicionam as informações de tipo. Omiti-lo não impede a compilação ou execução do seu código, mas você receberá a mesma mensagem de aviso desmarcada em tempo de compilação que viu com as classes de coleção.

Semelhante a obter o valor atual de um controle deslizante, você pode recuperar o item ou itens selecionados em uma lista a qualquer momento usando um dos quatro métodos:

getSelectedIndex()

Para listas de seleção única, retorna um int

getSelectedIndices()

Para listas multisselecionadas, retorna uma matriz de int

getSelectedValue()

Para listas de seleção única, retorna um objeto

getSelectedValues()

Para listas de seleção múltipla, retorna uma matriz de objetos

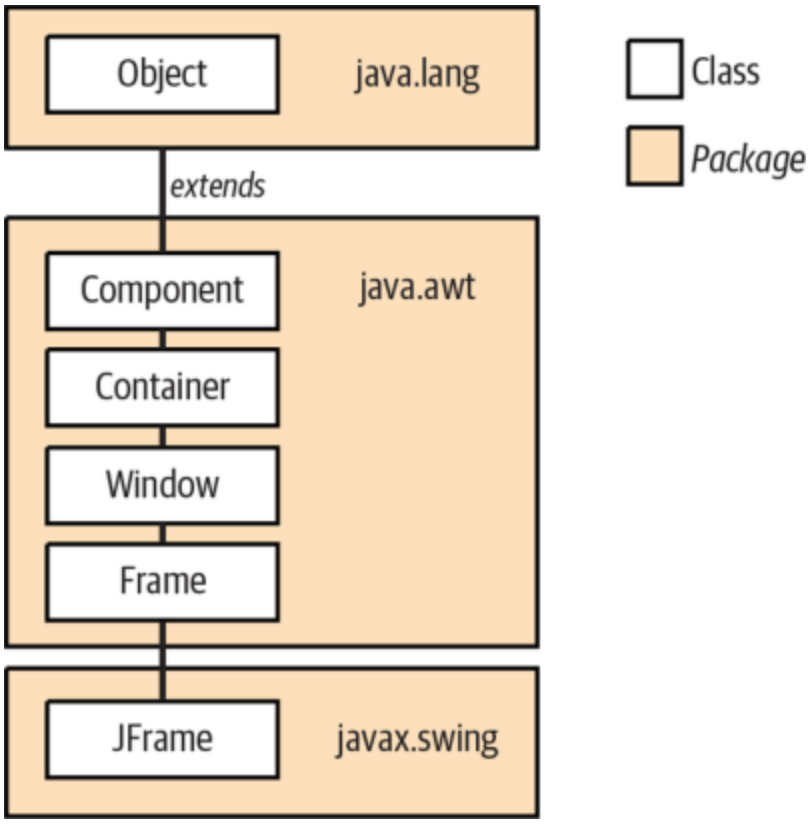
A principal diferença é se o índice do(s) item(s) selecionado(s) ou o(s) valor(es) real(is) é mais útil para você. Brincando com nossa lista de cidades em jshell, podemos extrair uma cidade selecionada assim:

```
jshell> cityList.getSelectedIndex()

$ 31 ==> 2

jshell> cityList.getSelectedIndices()

$ 32 ==> int[1] { 2 }

jshell> cityList.getSelectedValue()

$ 33 ==> "Chicago"

jshell> cidades[cityList.getSelectedIndex()]

$ 34 ==> "Chicago"
```

Para listas grandes, você provavelmente desejará uma barra de rolagem. O Swing promove a reutilização em seu código, então você pode usar um JScrollPane com JList assim como fizemos para áreas de texto.

Contêineres e Layouts

Essa formidável lista de componentes é apenas um subconjunto dos widgets disponíveis. Nesta seção, você organizará os componentes que discutimos em arranjos úteis. Esses arranjos acontecem dentro de um contêiner, que é o termo Java para um componente que pode ter (ou “conter”) outros componentes. Vamos começar examinando os contêineres mais comuns.



Molduras e Janelas

Todo aplicativo de desktop precisa de pelo menos uma janela. Este termo é anterior ao Swing e é usado pela maioria das interfaces gráficas disponíveis nos três grandes sistemas operacionais – incluindo o Windows (sem relação). O Swing fornece uma classe JWindow de baixo nível se você precisar, mas provavelmente você construirá seu aplicativo dentro de um JFrame.Figura 12-20ilustra a hierarquia de classes do JFrame. Manteremos seus recursos básicos, mas à medida que seus aplicativos se tornam mais ricos, você pode querer criar janelas personalizadas usando elementos superiores na hierarquia.

Figura 12-20. A hierarquia de classes JFrame

Vamos revisitar a criação daquela primeira aplicação gráfica deCapítulo 2e concentre-se um pouco mais no objeto JFrame:

```
importar javax.swing.*;

classe pública OláJavaAgain {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Olá, Java!");

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setSize(300, 150);

Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

frame.add(rótulo);

frame.setVisible (verdadeiro);

}

}
```

A string que passamos para o construtor JFrame se torna o título da janela. Em seguida, definimos algumas propriedades específicas em nosso objeto. Garantimos que quando o usuário fechar a janela, saiamos do nosso programa. (Isso pode parecer óbvio, mas aplicativos complexos podem ter múltiplas janelas, como paletas de ferramentas ou suporte para vários documentos. Fechar uma janela nesses aplicativos pode não significar “sair”.)

Em seguida, escolhemos um tamanho inicial para a janela e adicionamos nosso componente de rótulo real ao quadro (que por sua vez coloca o rótulo em seu painel de conteúdo, mais sobre isso em um minuto). Depois que o componente é adicionado, tornamos a janela visível e o resultado é Figura 12-21.

Figura 12-21. Um JFrame simples com um rótulo adicionado

Este processo básico é a base de toda aplicação Swing. A parte interessante do seu aplicativo vem do que você faz com esse painel de conteúdo.

Mas o que é esse painel de conteúdo? A estrutura utiliza seu próprio conjunto de contêineres que contêm diversas partes de aplicações típicas. Você pode definir seu próprio painel de conteúdo como qualquer objeto descendente de java.awt.Container, mas manteremos o padrão por enquanto.

Também estamos usando um atalho para adicionar nosso rótulo. A versão JFrame de add() delegará ao add() do

painel de conteúdo. O trecho a seguir mostra como adicionar o rótulo sem o atalho:

Rótulo JLabel = new JLabel("Olá, Java!", JLabel.CENTER);

frame.getContentPane().add(label);

A classe JFrame não possui atalhos para tudo, entretanto. Leia a documentação e use um atalho, se existir. Caso contrário, não hesite em obter uma referência via getContentPane() e então configurar ou ajustar esse contêiner conforme necessário.

JPanel

O painel de conteúdo padrão é um JPanel, o contêiner ideal no Swing. É um componente como JButton ou JLabel, portanto seus painéis podem conter outros painéis. Esse aninhamento geralmente desempenha um papel importante no layout do aplicativo. Por exemplo, você pode criar um JPanel para abrigar os botões de formatação de um editor de texto em uma “barra de ferramentas” e, em seguida, adicionar essa barra de ferramentas ao painel de conteúdo. Esse arranjo torna mais fácil para os usuários mostrar, ocultar ou movê-lo.

JPanelpermite adicionar e remover componentes da tela. (Os métodos são herdados da classe Container, mas nós os acessamos através de nossos objetos JPanel.) Você também pode repaint() um painel se algo mudou e você deseja atualizar sua UI.

Podemos ver os efeitos dos métodos add() e remove() mostrados emFigura 12-

22usando o nosso widget playground em jshell:


```
jshell> Widget w = novo Widget()

w ==> ch10.Widget[frame0,0,23,300x300, ...
kingEnabled=true]

jshell> JLabel emailLabel = new JLabel("E-mail:")

emailLabel ==> javax.swing.JLabel[,0,0,0x0 ...
ition=CENTER]

jshell> JTextField emailField = novo JTextField(12)

emailField ==> javax.swing.JTextField[,0,0,0x0, ...
LEADING]

jshell> JButton submitButton = new JButton("Enviar")

submitButton ==> javax.swing.JButton[,0,0,0x0, ...
ble=true]

jshell> w.add(emailLabel);

$8 ==> javax.swing.JLabel[,0,0,0x0, ... ition=CENTER]

//Captura de tela à esquerda na imagem acima

jshell> w.add(emailField)

$9 ==> javax.swing.JTextField[,0,0,0x0, ...
nment=LEADING]

jshell> w.add(submitButton)

$10 ==> javax.swing.JButton[,0,0,0x0, ... pable=true]
```

// Agora temos a captura de tela do meio

jshell> w.remove(emailLabel)

// E finalmente a captura de tela certa

Tente você mesmo! A maioria dos aplicativos não adiciona e remove componentes à toa. Normalmente, você construirá sua interface adicionando o que precisa e depois deixará como está. Você pode ativar ou desativar alguns botões ao longo do caminho, mas tente não surpreender o usuário com partes desaparecendo ou novos elementos aparecendo.

Figura 12-22. Adicionando e removendo componentes em um JPanel

Gerentes de layout

Contêineres como JPanel são responsáveis por organizar os componentes que você adiciona. Java fornece vários gerenciadores de layout para ajudá-lo a alcançar os resultados desejados.

BorderLayout

Você já viu o FlowLayout em ação. Você estava usando outro gerenciador de layout sem realmente saber: o painel de conteúdo de um JFrame usa BorderLayout por padrão.Figura 12-23mostra as cinco áreas controladas pelo BorderLayout, juntamente com suas regiões. Observe que as regiões NORTE e SUL são tão largas quanto a janela

do aplicativo, mas apenas tão altas quanto necessário para caber no rótulo. Da mesma forma, as regiões LESTE

e OESTE preenchem a lacuna vertical entre as regiões NORTE

e SUL, mas têm apenas a largura necessária, deixando o espaço restante a ser preenchido tanto horizontal como verticalmente pela região CENTRO:

```
importar java.awt.*;

importar javax.swing.*;

classe pública BorderLayoutDemo {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Demonstração de BorderLayout");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setSize(400, 200);

JLabel norteLabel = new JLabel("Topo - Norte", JLabel.CENTER); JLabel sulLabel = new JLabel("Inferior - Sul", JLabel.CENTER); JLabel eastLabel = new JLabel("Direita - Leste", JLabel.CENTER); JLabel westLabel = new JLabel("Esquerda - Oeste", JLabel.CENTER); JLabel centerLabel = new JLabel("Centro (todo o resto)",

JLabel.CENTRO);

// Pinte os rótulos para que possamos ver melhor seus limites

norteLabel.setOpaque(true);

norteLabel.setBackground(Color.GREEN);

sulLabel.setOpaque(true);
```

```
sulLabel.setBackground(Color.GREEN);
eastLabel.setOpaque(true);
eastLabel.setBackground(Color.RED);
westLabel.setOpaque(true);
westLabel.setBackground(Color.RED);
centerLabel.setOpaque(true);
centerLabel.setBackground(Color.YELLOW);
frame.add(northLabel, BorderLayout.NORTH);
frame.add(southLabel, BorderLayout.SOUTH);
frame.add(eastLabel, BorderLayout.EAST);
frame.add(westLabel, BorderLayout.WEST);
frame.add(centerLabel, BorderLayout.CENTER);
frame.setVisible (verdadeiro);
}
}
```

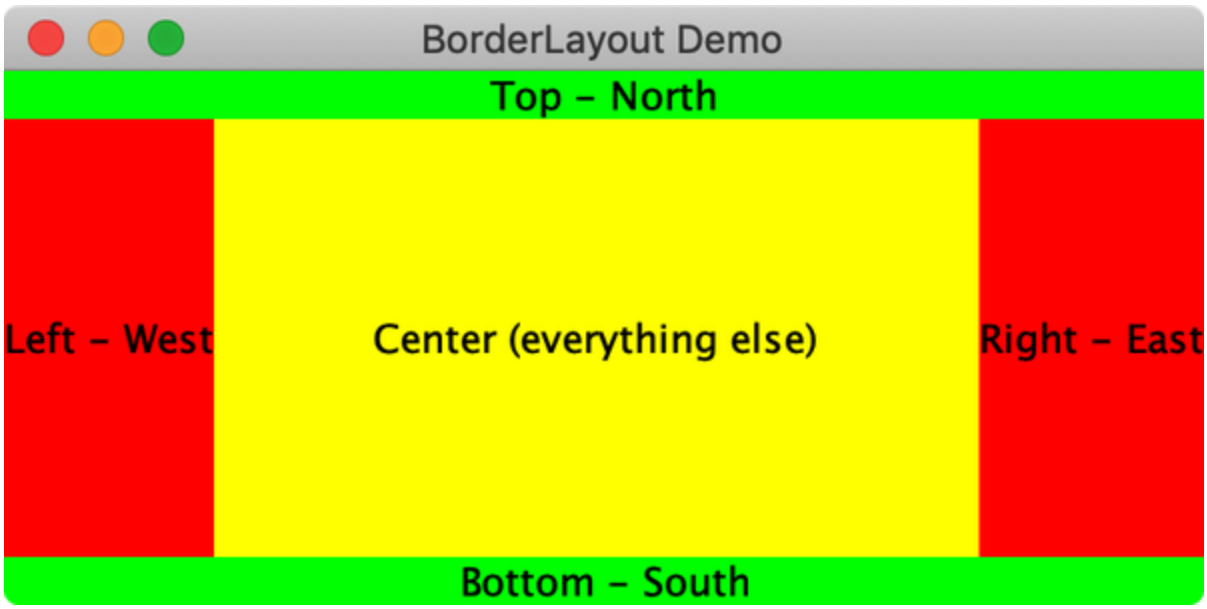


Figura 12-23. As regiões disponíveis com BorderLayout

O método add() neste caso pega um argumento extra e o passa para o gerenciador de layout. (Nem todos os gerentes precisam desse argumento, como você viu no FlowLayout.)

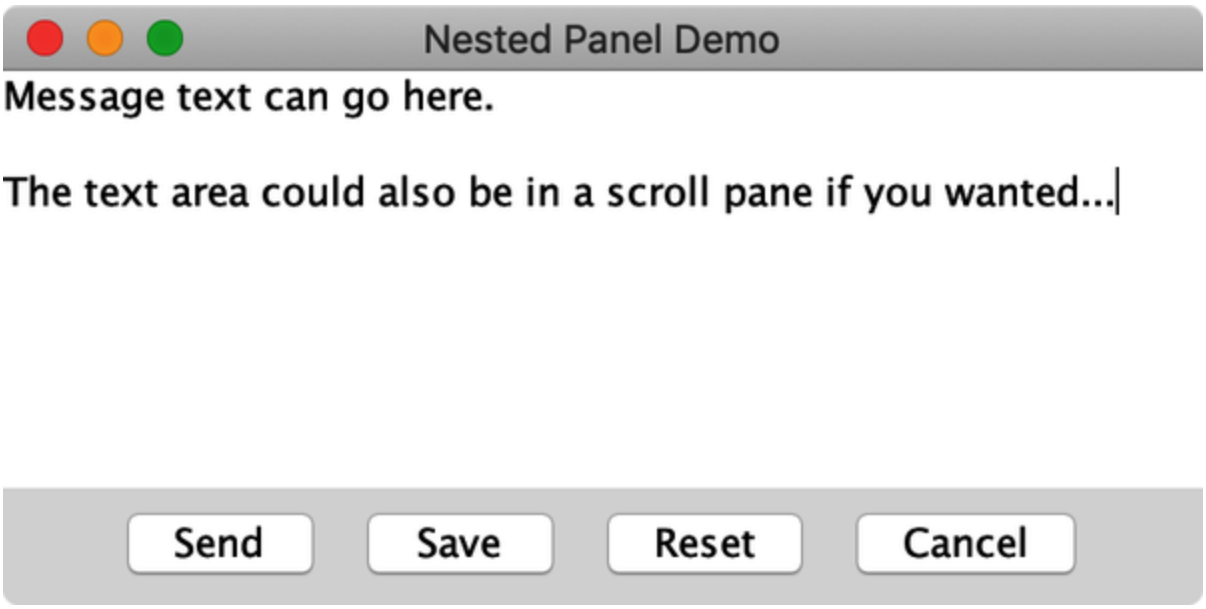
Figura 12-24mostra um exemplo de aninhamento de objetos JPanel em um aplicativo.

Usamos uma área de texto para uma mensagem grande no centro e, em seguida, adicionamos alguns botões de ação a um painel na parte inferior. Novamente, sem os eventos que abordaremos na próxima seção, nenhum desses botões faz nada, mas queremos mostrar como trabalhar com vários contêineres. E você poderia continuar aninhando objetos JPanel se quisesse.

Às vezes, uma melhor escolha de layout de nível superior torna seu aplicativo mais sustentável e com melhor desempenho:

classe pública NestedPanelDemo {

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

Quadro JFrame = new JFrame("Demonstração do painel
aninhado");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setSize(400, 200);

// Cria a área de texto e adiciona-a ao centro

JTextArea mensagemArea = new JTextArea();

frame.add(messageArea, BorderLayout.CENTER);

// Cria o contêiner do botão

JPanel buttonPanel = new JPanel(new FlowLayout());

// Cria os botões

JButton sendButton = new JButton("Enviar");

JButton saveButton = new JButton("Salvar");

JButton resetButton = new JButton("Redefinir");

JButton cancelarButton = new JButton("Cancelar");

//Adiciona os botões ao seu contêiner

buttonPanel.add(sendButton);

buttonPanel.add(saveButton);

buttonPanel.add(resetButton);

buttonPanel.add(cancelarButton);
```




```
// E finalmente, adicione esse contêiner na parte inferior
do aplicativo frame.add(buttonPanel,
BorderLayout.SOUTH);

frame.setVisible (verdadeiro);

}

}
```

Figura 12-24. Um exemplo simples de contêiner aninhado

Duas coisas a serem observadas neste exemplo. Primeiro, você pode ver que não especificamos o número de linhas ou colunas ao criar nosso objeto JTextArea. Ao contrário do FlowLayout, o BorderLayout definirá o tamanho de seus componentes quando possível. Para a parte superior e inferior, isso significa usar a própria altura do componente, semelhante ao funcionamento do FlowLayout, mas depois definir a largura do componente para preencher o quadro. Os lados usam a largura de seus componentes, mas o gerenciador de layout define a

altura. BorderLayout define a largura e a altura do componente no centro.

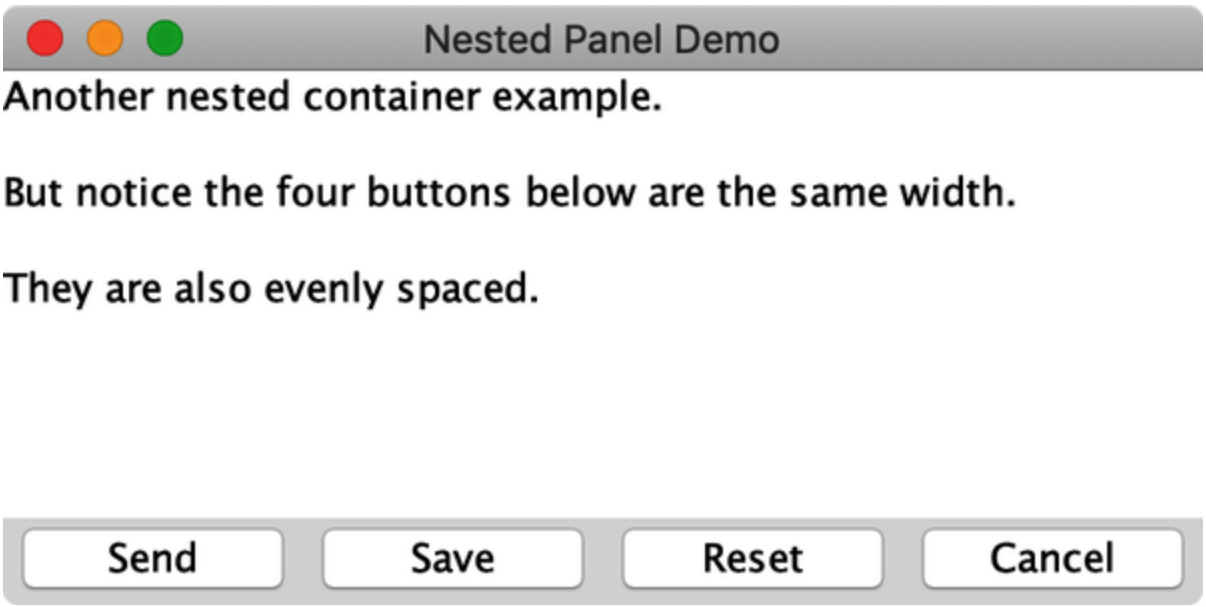
Segundo, quando adicionamos os objetos messageArea e buttonPanel ao quadro, especificamos o argumento “where” extra ao método add() do quadro. No entanto, quando adicionamos os próprios botões ao buttonPanel, usamos a versão mais simples de add() apenas com o argumento do componente. O gerenciador de layout do contêiner determina qual variação de add() precisamos. Portanto, embora o buttonPanel esteja na região SUL do quadro usando BorderLayout, o saveButton e seus compatriotas estão em um contêiner próprio e não sabem ou se importam com o que está acontecendo fora desse contêiner.

Layout de grade

Muitas vezes você precisa (ou deseja) que seus componentes ou rótulos ocupem espaços simétricos. Pense nos botões Sim, Não e Cancelar na parte inferior de uma caixa de diálogo de confirmação. (O Swing também pode criar esses diálogos; mais

sobre isso em“Modais e Pop-Ups”.) A classe GridLayout pode ajudar com esse espaçamento uniforme. Vamos tentar usar GridLayout para esses botões no exemplo anterior. Tudo o que precisamos fazer é alterar uma linha:

```
// Cria o contêiner do botão. Versão antiga:

// JPanel buttonPanel = new JPanel(new FlowLayout());

JPanel buttonPanel = new JPanel(new GridLayout(1,0));
```



As chamadas para add() permanecem exatamente as mesmas; nenhum argumento de restrição separado é necessário.

Como você pode ver emFigura 12-25, os botões GridLayout são do mesmo tamanho, embora o texto do botão Cancelar seja um pouco mais longo que os outros.

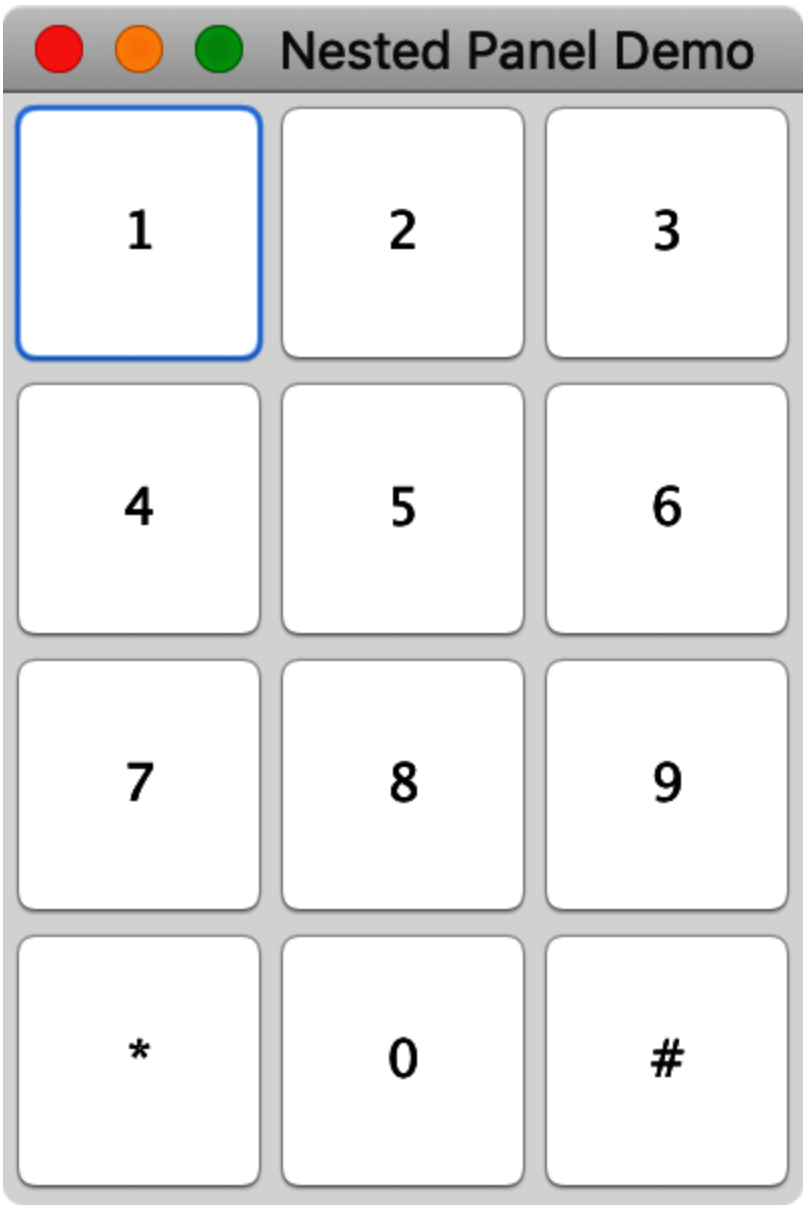
Figura 12-25. Usando GridLayout para uma linha de botões

Ao criar o gerenciador de layout, dissemos que queremos exatamente uma linha, sem restrições de quantas colunas (1, 0). As grades também podem ser bidimensionais com múltiplas linhas e colunas.Figura 12-26mostra o layout clássico do teclado do telefone como exemplo.

classe pública PhoneGridDemo {

public static void main(String[] args) {

```
Quadro JFrame = new JFrame("Demonstração do painel aninhado");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setSize(200, 300);

// Cria o contêiner do phone pad

JPanel phonePad = novo JPanel(novo GridLayout(4,3));

// Cria e adiciona os 12 botões, do canto superior esquerdo ao canto infe rior direito

phonePad.add(novo JButton("1"));

phonePad.add(novo JButton("2"));

phonePad.add(novo JButton("3"));

phonePad.add(novo JButton("4"));

phonePad.add(novo JButton("5"));

phonePad.add(novo JButton("6"));

phonePad.add(novo JButton("7"));

phonePad.add(novo JButton("8"));

phonePad.add(novo JButton("9"));

phonePad.add(novo JButton("*"));

phonePad.add(novo JButton("0"));

phonePad.add(novo JButton("#"));

// E finalmente, adicione o pad ao centro do aplicativo
```




```
frame.add(phonePad, BorderLayout.CENTER);

frame.setVisible (verdadeiro);

}

}
```

Adicionar os botões na ordem da esquerda para a direita, de cima para baixo, deve resultar no aplicativo que você

vê emFigura 12-26.

Figura 12-26. Um layout de grade bidimensional para um teclado telefônico Muito prático e muito fácil se você precisar de elementos perfeitamente simétricos.

Mas e se você quiser um layout principalmente simétrico? Pense em formulários web populares com uma coluna de rótulos à esquerda e uma coluna de campos de texto à direita. GridLayout poderia lidar com um formulário básico de duas colunas como esse, mas muitas vezes seus rótulos são curtos e simples, enquanto seus campos de texto são mais largos, dando ao usuário mais espaço para digitar. Como o Java acomoda esses layouts?

GridBagLayout

Se você precisa de um layout mais interessante, mas não quer aninhar muitos painéis, considere GridBagLayout. É mais complexo de configurar, mas permite layouts complexos que ainda mantêm os elementos esteticamente alinhados e dimensionados.

Semelhante ao BorderLayout, você adiciona componentes com um argumento extra. O

argumento para GridBagLayout, entretanto, é um objeto GridBagConstraints rico em vez de uma simples String.

A “grade” no GridBagLayout é exatamente isso, um contêiner retangular dividido em várias linhas e colunas. A parte da “bolsa”, porém, vem de uma noção de como você usa as células criadas por essas linhas e colunas. As linhas e colunas podem ter altura ou largura próprias e os componentes podem ocupar qualquer conjunto retangular de células. Podemos aproveitar essa

flexibilidade para construir nossa interface de jogo com um único JPanel em vez de vários painéis aninhados.Figura 12-27mostra uma maneira de dividir a tela em quatro linhas e três colunas e depois posicionar os componentes.

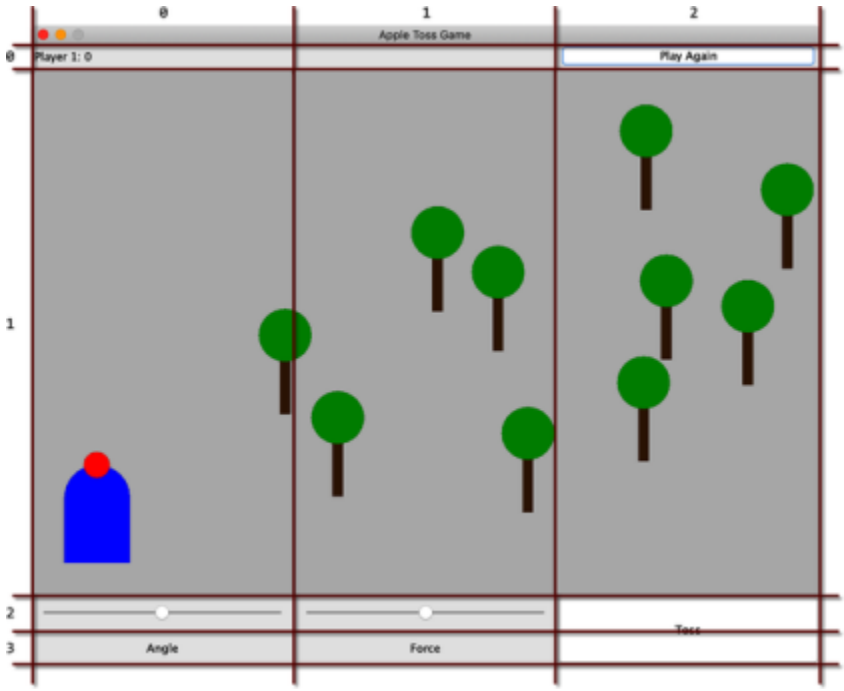


Figura 12-27. Um exemplo de grade para uso com GridBagLayout

Você pode ver as diferentes alturas das linhas e larguras das colunas. Alguns componentes ocupam mais de uma célula. Esse tipo de arranjo não funciona para todos os aplicativos, mas é poderoso e funciona para muitas UIs que precisam de mais do que simples layouts.

Para construir um aplicativo com GridBagLayout, você precisa manter algumas referências ao adicionar componentes. Vamos configurar a grade primeiro: público estático final int SCORE_HEIGHT = 30;

público estático final int CONTROL_WIDTH = 300;

público estático final int CONTROL_HEIGHT = 40;

público estático final int FIELD_WIDTH = 3 * CONTROL_WIDTH;

público estático final int FIELD_HEIGHT = 2 * CONTROL_WIDTH;

flutuador final estático público FORCE_SCALE = 0,7f;

GridBagLayout gameLayout = new GridBagLayout();

gameLayout.columnWidths = novo int[]

{ CONTROL_WIDTH, CONTROL_WIDTH, CONTROL_WIDTH };

gameLayout.rowHeights = novo int[]

{SCORE_HEIGHT, FIELD_HEIGHT, CONTROL_HEIGHT, CONTROL_HEIGHT };

JPanel gamePane = new JPanel(gameLayout);

Esta etapa requer um pouco de planejamento de sua parte, mas é fácil de ajustar quando você coloca alguns componentes na tela. Para adicionar esses componentes, você precisa criar e configurar um objeto GridBagConstraints. Felizmente, você pode reutilizar o mesmo objeto para todos os seus componentes – basta repetir a parte de configuração antes de adicionar cada elemento. Aqui está um exemplo de como você pode adicionar o componente principal do campo de jogo:

GridBagConstraints gameConstraints = new GridBagConstraints();

gameConstraints.fill = GridBagConstraints.BOTH;

```
gameConstraints.gridy = 1;

gameConstraints.gridx = 0;

gameConstraints.gridheight = 1;

gameConstraints.gridwidth = 3;

Campo campo = new Campo();

gamePane.add(campo, gameConstraints);
```

Observe como definimos quais células o campo ocupará. Especificamos o canto superior esquerdo de um retângulo fornecendo a linha (gridy) e a coluna (gridx). Em seguida especificamos quantas linhas nosso campo irá ocupar (gridheight) e quantas colunas (gridwidth). Este é o núcleo da configuração das restrições do grid bag.

Você também pode ajustar coisas como a forma como um componente preencherá as células que ocupa (preencher) e quanto de margem cada componente obtém.

Decidimos simplesmente preencher todo o espaço disponível em um grupo de células (“ambos” um preenchimento horizontal e um preenchimento vertical), mas você pode ler sobre mais opções nodocumentação para GridBagConstraints.

Vamos adicionar um rótulo de pontuação na parte superior:

```
gameConstraints.fill = GridBagConstraints.BOTH;

gameConstraints.gridy = 0;
```

```
gameConstraints.gridx = 0;

gameConstraints.gridheight = 1;

gameConstraints.gridwidth = 1;

JLabel scoreLabel = new JLabel("Jogador 1: 0");

gamePane.add(scoreLabel, gameConstraints);
```

Para este segundo componente, você percebe como a configuração das restrições é semelhante à forma como lidamos com o campo do jogo? Sempre que você encontrar semelhanças como essa, considere incluir essas etapas semelhantes em uma função que você possa reutilizar:

```
private GridBagConstraints buildConstraints(int linha, int col,

int rowspan, int colspan)

{

// Use nossa referência global para o objeto gameConstraints

gameConstraints.fill = GridBagConstraints.BOTH;

gameConstraints.gridy = linha;

gameConstraints.gridx = col;

gameConstraints.gridheight = rowspan;

gameConstraints.gridwidth = colspan;

retornar gameConstraints;
```

}

Então você poderia reescrever os blocos de código anteriores para o rótulo de pontuação e o campo de jogo, assim:

GridBagConstraints gameConstraints = new GridBagConstraints();

JLabel scoreLabel = new JLabel("Jogador 1: 0");

Campo campo = new Campo();

gamePane.add(scoreLabel, buildConstraints(0,0,1,1));

gamePane.add(campo, buildConstraints(1,0,1,3));

Com essa função instalada, você pode adicionar rapidamente vários outros componentes e rótulos para completar a interface do jogo. Por exemplo, o botão de lançamento no canto inferior direito d[oFigura 12-27p](#)ode ser configurado assim: JLabel lançarButton = new JButton("Lançar");

gamePane.add(tossButton, buildConstraints(2,2,2,1));

Muito mais limpo! Simplesmente continuamos criando nossos componentes e colocando-os nas linhas e colunas corretas, com os intervalos apropriados. No final temos um conjunto razoavelmente interessante de componentes dispostos em um único container.

Tal como acontece com outras seções deste capítulo, não temos tempo para cobrir todos os gerenciadores de layout, ou mesmo todos os recursos dos gerenciadores de layout que discutimos. Certifique-se de verificar a documentação do Java e tente criar alguns aplicativos

fictícios para brincar com os diferentes layouts. Como ponto de partida, BoxLayout é uma boa atualização para a ideia de grade, e GroupLayout pode produzir formulários de entrada de dados. Por enquanto, porém, vamos seguir em frente. É hora de “conectar” todos esses componentes e responder a todas as digitações, cliques e pressionamentos de botões – ações que são codificadas em Java como eventos.

Eventos

Como discutido em“Arquitetura do controlador Model View”, os elementos de modelo e visualização dos designs MVC são diretos. Mas e quanto ao aspecto do controlador?

No Swing (e no Java em geral), a interação entre usuários e componentes é comunicada por meio de eventos. Um evento contém informações gerais, como quando a interação ocorreu, bem como informações específicas do tipo de evento, como o ponto da tela onde você clicou com o mouse ou qual tecla você digitou no teclado. Um ouvinte (ou manipulador) capta a mensagem e pode responder de alguma forma útil. Conectar componentes a ouvintes é o que permite aos usuários controlar seu aplicativo.

Eventos de mouse

A maneira mais fácil de começar é gerar e manipular um evento. Vamos seguir os passos de nossos primeiros aplicativos rápidos com um aplicativo HelloMouse e focar no tratamento de eventos de mouse. Quando clicarmos com o mouse, usaremos esse evento de clique para determinar a posição do nosso JLabel. (A propósito, isso

exigirá a remoção do gerenciador de layout. Queremos definir as coordenadas de nossa etiqueta manualmente.)

❶

Ao observar o código-fonte deste exemplo, preste atenção a alguns itens específicos, anotados com legendas numeradas:

```
//nome do arquivo: ch12/examples/HelloMouse.java

pacote ch10.examples;

importar java.awt.*;

importar javax.swing.*;

importar java.awt.event.MouseEvent;

importar java.awt.event.MouseListener;

classe pública HelloMouse estende JFrame implementa
MouseListener {

Etiqueta JLabel;

public OláMouse() {

super("Demonstração de MouseEvent");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//remove o gerenciador de layout

setLayout(nulo);

setSize(300, 100);
```

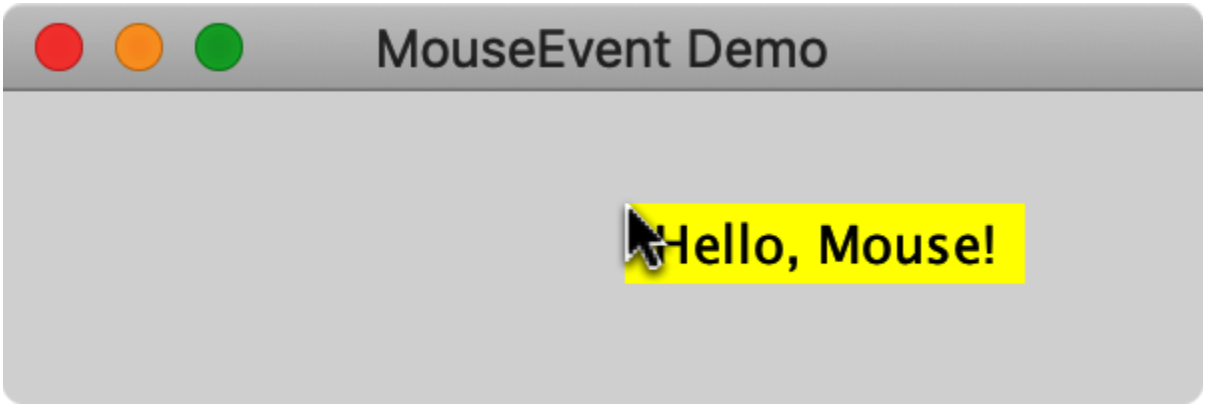
```
label = new JLabel("Olá, Mouse!", JLabel.CENTER);
rótulo.setOpaque(true);
label.setBackground(Color.AMARELO);
rótulo.setSize(100,20);
rótulo.setLocation(100,100);
adicionar(rótulo);
getContentPane().addMouseListener(this);
}
public void mouseClicked(MouseEvent e) {
label.setLocation(e.getX(), e.getY());
}
public void mousePressed(MouseEvent e) { }
public void mouseReleased(MouseEvent e) { }
public void mouseEntered(MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
public static void main(String[] args) {
Moldura HelloMouse = new HelloMouse();
frame.setVisible (verdadeiro);
}
}
```

Conforme você clica, o Java coleta eventos de baixo nível do seu hardware (computador, mouse, teclado) e os entrega a um ouvinte apropriado. Ouvintes são

❷

❸

❹



interfaces. Você pode criar classes especiais apenas para implementar a interface ou pode implementar ouvintes como parte da classe principal do aplicativo, como fizemos aqui. O local onde você escolhe lidar com os eventos realmente depende das ações que você precisa tomar em resposta a eles. Você verá vários exemplos de ambas as abordagens ao longo deste livro.

Implementamos a interface MouseListener além de estender o JFrame. Tivemos que fornecer um corpo para cada método listado em MouseListener, mas fazemos nosso trabalho real em mouseClicked(). Este método pega as coordenadas do clique do objeto de evento e as utiliza para alterar a posição do rótulo. A classe MouseEvent contém muitas informações sobre o evento: quando ocorreu, em qual componente ocorreu, qual botão do mouse estava envolvido, a coordenada (x,y)

onde o evento ocorreu e assim por diante. Tente imprimir algumas dessas informações em alguns métodos não implementados, como mouseDown().

Adicionamos alguns métodos para outros tipos de eventos de mouse que não usamos.

Isso é comum com eventos de nível inferior, como eventos de mouse e teclado. As interfaces do listener são projetadas para fornecer um ponto central de coleta de eventos relacionados. Você deve implementar todos os métodos na interface, mas pode responder aos eventos específicos de seu interesse e deixar os outros métodos vazios.

A outra parte crítica do novo código é a chamada para addMouseListener() para nosso painel de conteúdo. A sintaxe pode parecer um pouco estranha, mas é uma abordagem padrão. Usar getContentPane() diz “este é o componente que gera eventos” e usar this como argumento diz “esta é a classe que recebe (manipula) os eventos”. Neste exemplo, os eventos do painel de conteúdo do quadro serão entregues de volta à mesma classe, onde colocamos todo o código de manipulação do mouse.

Vá em frente e execute o aplicativo. Você obterá uma variação do conhecido aplicativo gráfico “Hello, World”, mostrado naFigura 12-28. A mensagem amigável deve seguir o mouse enquanto você clica.

Figura 12-28. Usando um MouseEvent para posicionar um rótulo

Adaptadores de mouse

Se quiser tentar a abordagem da classe auxiliar, você pode adicionar uma classe separada ao arquivo e implementar MouseListener nessa classe. Nesse caso, você pode aproveitar um atalho que o Swing oferece para muitos ouvintes. A classe

MouseAdapter é uma implementação simples da interface MouseListener, com métodos vazios escritos para cada evento. Ao estender esta classe, você substitui apenas os métodos de seu interesse. Isso cria um manipulador limpo e conciso:

```
//nome do arquivo:
ch12/examples/HelloMouseHelper.java

pacote ch12.examples;

importar java.awt.*;

importar java.awt.event.MouseEvent;

importar java.awt.event.MouseAdapter;

importar javax.swing.*;

classe pública HelloMouseHelper {

public static void main(String[] args) {

Quadro JFrame = new JFrame("MouseEvent Demo");

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setLayout(nulo);

frame.setSize(300, 300);
```

```
Rótulo JLabel = new JLabel("Olá, Mouse!",
JLabel.CENTER);

rótulo.setOpaque(true);

label.setBackground(Color.AMARELO);

rótulo.setSize(100,20);

rótulo.setLocation(100,100);

frame.add(rótulo);

Movedor LabelMover = new LabelMover(rótulo);

frame.getContentPane().addMouseListener(mover);

frame.setVisible (verdadeiro);

}

}

/**

* Classe auxiliar para mover um rótulo para a posição de
um clique do mou se.

* Lembre-se do Capítulo 5 que as classes secundárias
incluídas no mesmo

* a classe pública não deve ser pública. Eles podem ser
protegidos,

* arquivo como privado ou pacote privado (sem
qualificador).

*/
```

```
classe LabelMover estende MouseAdapter {

JLabel rótuloToMove;

public LabelMover(rótulo JLabel) {

labelToMove = etiqueta;

}

public void mouseClicked(MouseEvent e) {

labelToMove.setLocation(e.getX(), e.getY());

}

}
```

Lembre-se de que as classes auxiliares precisam ter uma referência para cada objeto que tocam. Passamos nosso rótulo para o construtor do nosso adaptador. Essa é uma maneira popular de estabelecer as conexões necessárias, mas você certamente poderia adicionar o acesso necessário posteriormente — desde que o manipulador tenha uma referência para cada objeto necessário antes de começar a receber eventos.

Eventos de ação

Embora eventos de mouse e teclado de baixo nível estejam disponíveis em quase todos os componentes do Swing, eles podem ser um pouco entediantes. A maioria das bibliotecas de UI fornece eventos de nível superior que são mais simples de pensar. O

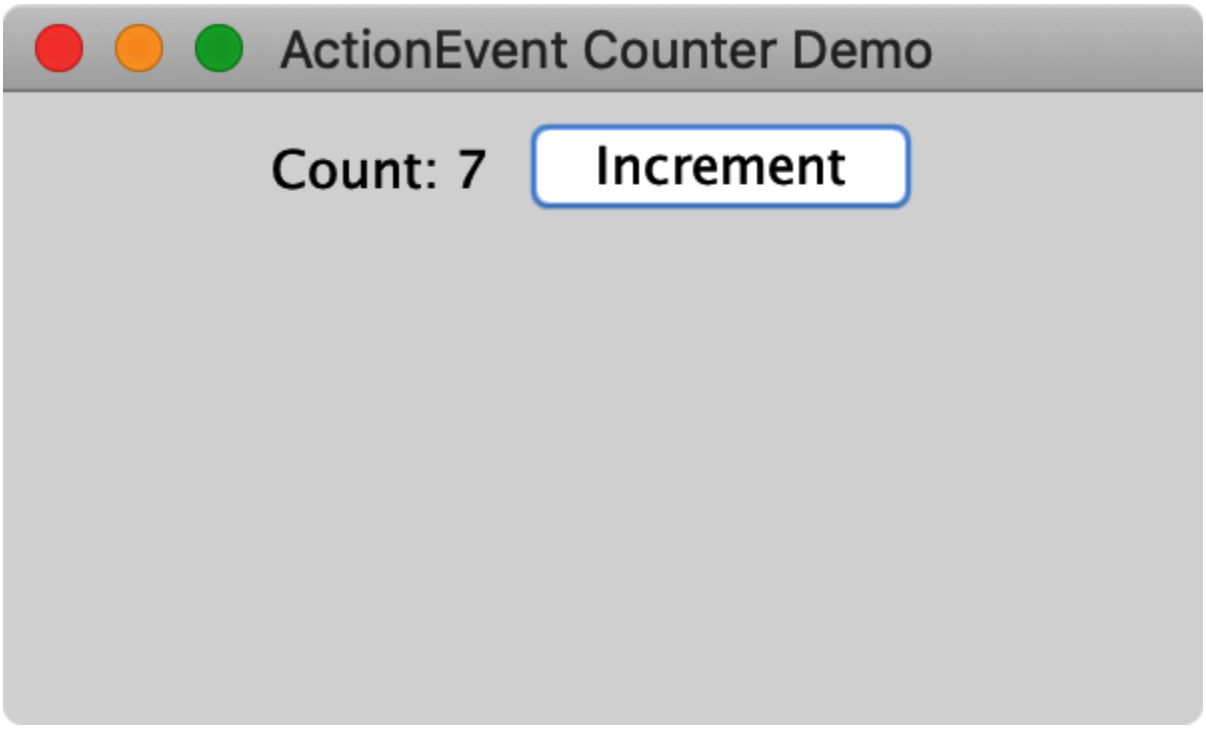
balanço não é exceção. A classe JButton, por exemplo, suporta um ActionEvent que informa que o botão foi clicado. Na maioria das vezes, é exatamente isso que

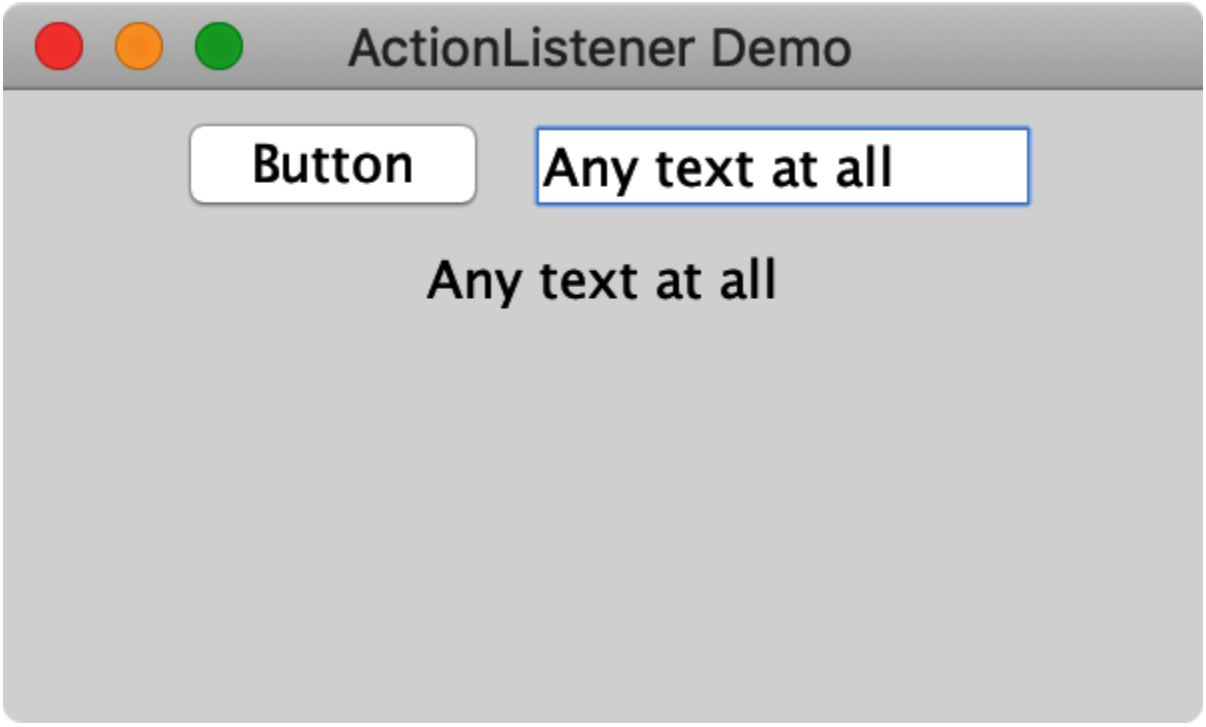
você deseja. Mas os eventos do mouse ainda estarão disponíveis se você precisar de algum comportamento especial, como reagir a cliques de diferentes botões do mouse ou distinguir entre um toque longo e um toque curto em uma tela sensível ao toque.

Uma maneira popular de demonstrar o evento de clique do botão é construir um contador simples, como o que você vê emFigura 12-29. Cada vez que você clica no botão, o programa atualiza o rótulo. Esta simples prova de conceito mostra que você pode receber e responder a eventos de UI. Vamos ver a fiação necessária para esta demonstração:

```
pacote ch12.examples;

importar javax.swing.*;

importar java.awt.*;

importar java.awt.event.ActionEvent;

importar java.awt.event.ActionListener;

classe pública ActionDemo1 estende JFrame implementa ActionListener {

int contadorValor = 0;

JLabel contadorLabel;

public ActionDemo1() {

super("Demonstração do contador de ActionEvent");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

setLayout(novo FlowLayout());
```

```
setSize(300, 180);

counterLabel = new JLabel("Contagem: 0",
JLabel.CENTER);

add(contadorLabel);

Incrementador JButton = new JButton("Incremento");

incrementer.addActionListener(this);

adicionar(incrementador);
```






```
}
public void actionPerformed(ActionEvent e) {
contadorValor++;
counterLabel.setText("Contagem: " + counterValue);
}
public static void main(String[] args) {
Demonstração ActionDemo1 = new ActionDemo1();
demo.setVisible(verdadeiro);
}
}
```

Figura 12-29. Usando ActionEvent para incrementar um contador

Atualizamos uma variável de contador simples e exibimos o resultado dentro do método actionPerformed(), que é onde os objetos ActionListener recebem seus eventos. Usamos a abordagem de implementação de ouvinte direto, mas poderíamos facilmente ter criado uma classe auxiliar como fizemos com o exemplo LabelMover

em“Eventos do rato”.

Os eventos de ação são diretos; eles não têm tantos detalhes disponíveis quanto os eventos de mouse, mas carregam uma propriedade de “comando”. O comando em questão é apenas uma string arbitrária. Isso não significa nada para Java, mas você pode personalizar essa propriedade para seu próprio uso. Para botões, o padrão do Java é usar o texto do rótulo do botão. A classe JTextField também gera um evento de ação se você pressionar a tecla Return enquanto digita no campo de texto. Neste caso, entretanto, o texto atualmente no campo é usado para o comando.Figura 12-30mostra como conectar um botão e um campo de texto a um rótulo.

Figura 12-30. Usando ActionEvents de diferentes fontes

```
classe pública ActionDemo2 {

public static void main(String[] args) {

Quadro JFrame = new JFrame("Demonstração de
ActionListener");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
frame.setLayout(novo FlowLayout());

frame.setSize(300, 180);

Etiqueta JLabel = new JLabel("Os resultados vão aqui",
JLabel.CENTER); Ajudante ActionCommandHelper = novo
ActionCommandHelper(rótulo);

JButton simpleButton = new JButton("Botão");

simpleButton.addActionListener(ajudante);

JTextField campo simples = new JTextField(10);

simpleField.addActionListener(ajudante);

frame.add(simpleButton);

frame.add(simpleField);

frame.add(rótulo);

frame.setVisible (verdadeiro);

}

}

/**

* Classe auxiliar para mostrar a propriedade de comando
de qualquer Actio nEvent em um determinado rótulo.

*/

classe ActionCommandHelper implementa ActionListener
{
```

```
JLabel resultadoLabel;

public ActionCommandHelper(rótulo JLabel) {

resultadoLabel = rótulo;

}

public void actionPerformed(ActionEvent ae) {

resultLabel.setText(ae.getActionCommand());

}

}
```

Observe que usamos um objeto ActionListener para manipular os eventos do botão e do campo de texto. Este é um ótimo recurso da abordagem de ouvinte do Swing para lidar com eventos: qualquer componente que gere um determinado tipo de evento pode reportar a qualquer ouvinte que receba esse tipo. Às vezes, seus manipuladores de eventos são únicos e você criará um manipulador separado para cada componente.

Mas muitos aplicativos oferecem diversas maneiras de realizar a mesma tarefa. Muitas vezes você pode lidar com essas diferentes fontes de entrada com um único ouvinte. E

quanto menos código você tiver, menos coisas podem dar errado!

Alterar eventos

Outro tipo de evento que aparece em vários componentes Swing é ChangeEvent. Este é um evento simples que permite que você saiba que algo mudou. A

classe JSlider usa esse mecanismo para relatar alterações na posição do controle deslizante. A classe ChangeEvent possui uma referência ao componente que foi alterado (a origem do evento), mas não contém detalhes sobre o que pode ter sido alterado nesse componente. Cabe a você solicitar esses detalhes ao componente. Esse processo de ouvir e consultar pode parecer tedioso, mas permite notificações eficientes de que atualizações são necessárias, sem criar centenas de classes com milhares de métodos para cobrir todas as variações de eventos que possam surgir.

Não reproduziremos o aplicativo inteiro aqui, mas vamos dar uma olhada em como a classe AppleToss usa ChangeListener para mapear o controle deslizante de mira para nosso físico:

```
// arquivo: ch12/examples/game/AppleToss.java

gamePane.add(buildAngleControl(), buildConstraints(2, 0, 1, 1));

//outras coisas de configuração...

JSlider privado buildAngleControl() {

// Nossa mira pode variar de 0 a 180 graus

Controle deslizante JSlider = novo JSlider (0,180);

// mas o 0 trigonométrico está no lado direito, não no esquerdo

slider.setInverted(true);

// Sempre que o valor do controle deslizante mudar, atualize o player slider.addChangeListener(new
```

```
ChangeListener() {

public void stateChanged(ChangeEvent e) {

player1.setAimingAngle((float)slider.getValue());

field.repaint();

}

});

controle deslizante de retorno;

}
```

Neste trecho, usamos um padrão de fábrica para criar nosso controle deslizante e retorná-lo para uso no método add() de nosso contêiner gamePane. Criamos uma classe interna anônima simples. Alterar nosso controle deslizante de mira tem um efeito e só há uma maneira de mirar a maçã. Como não há possibilidade de reutilização de classe, escolhemos uma classe interna anônima. Não há nada de errado em criar uma classe auxiliar completa e passar para ela os elementos player1 e field como argumentos para um construtor ou método de inicialização, mas você encontrará a abordagem usada acima com bastante frequência.

Queremos apontar outra opção para lidar com eventos simples como ChangeEvent e ActionEvent. Os ouvintes desses eventos possuem um único método abstrato. Essa

frase lhe lembra alguma coisa? É assim que a Oracle descreve suas interfaces funcionais. Então podemos usar um lambda!

```
// E agora, sempre que o valor do controle deslizante
mudar, devemos atua lizar

slider.addChangeListener(e -> {

player1.setAimingAngle((float)slider.getValue());

field.repaint();

});
```

Infelizmente, muitos ouvintes lidam com uma variedade de eventos relacionados.

Você não pode usar um lambda com qualquer interface de ouvinte que tenha mais de um método. Mas lambdas funcionam com botões e itens de menu, então eles ainda podem desempenhar um papel importante em sua aplicação gráfica se você os achar atraentes.

Nosso widget não é muito bom para testar código relacionado a eventos em jshell.

Embora você certamente possa escrever uma classe interna anônima ou um lambda multilinha em uma linha de comando, isso pode ser tedioso e sujeito a erros que não são fáceis de corrigir na mesma linha de comando. Geralmente é mais simples escrever aplicativos de demonstração pequenos e focados, como muitos dos exemplos deste capítulo. Embora encorajemos você a iniciar o jogo de lançar maçãs para brincar com o controle deslizante mostrado no código acima, você também deve experimentar alguns aplicativos originais.

Outros eventos

Existem dezenas de outros eventos e ouvintes espalhados pelos pacotes java.awt.event e javax.swing.event. Vale a pena dar uma olhada na documentação apenas para ter uma ideia dos outros tipos de eventos que você pode

encontrar.Tabela 12-2mostra os eventos e ouvintes associados aos componentes que discutimos até agora neste capítulo, bem como alguns que valem a pena conferir à medida que você trabalha mais com o Swing.

Tabela 12-2. Eventos Swing e AWT e ouvintes associados

S/A Classe de evento

Interface do ouvinte

Gerando componentes

A

Evento de ação

ActionListener

Botão J, JMenuItem,

JTextField

S

Evento de mudança

AlterarListener

JSlider

A

ItemEvento

ItemListener

JCheckBox, JRadioButton

A

Evento-chave

KeyListener

Descendentes do

Componente

S

ListSelectionEvent ListSelectionListener JList

S/A Classe de evento

Interface do ouvinte

Gerando componentes

A

Evento do mouse

MouseListener

Descendentes do

Componente

A

Evento MouseMotion MouseMotionListener

Descendentes do

Componente

Eventos AWT (A) de java.awt.event, eventos Swing (S) de javax.swing.event Se você não tiver certeza de quais eventos um componente específico suporta, verifique sua documentação para métodos semelhantes a addXYZListener(). O que quer que signifique XYZ é uma pista sobre onde mais procurar na documentação.

Lembre-se de que nosso controle deslizante usa addChangeListener(). Portanto, XYZ é Mudança neste caso. Você pode inferir o nome do evento (ChangeEvent) e a interface do ouvinte (ChangeListener) dessa pista. Depois de ter a documentação do ouvinte, tente implementar todos os métodos e simplesmente imprimir qual evento é relatado.

Você pode aprender muito sobre como os vários componentes do Swing reagem aos eventos do teclado e do mouse dessa maneira.

Considerações sobre rosqueamento

Se você leu alguma documentação do JDK sobre Swing enquanto trabalhava neste capítulo, você pode ter se deparado com um aviso de que os componentes do Swing não são thread-safe. Como você aprendeu emCapítulo 9, Java oferece suporte a vários threads de execução para aproveitar o poder de processamento do computador moderno. Aplicativos multithread correm o risco de permitir que dois threads lutem pelo mesmo recurso ou atualizem a mesma variável ao mesmo tempo, mas com valores diferentes. Não saber se seus dados estão corretos pode diminuir drasticamente sua capacidade de depurar um programa ou até mesmo

confiar em sua saída. Para componentes Swing, este aviso lembra aos programadores que seus elementos de UI podem estar sujeitos a este tipo de corrupção.

Para ajudar a manter uma UI consistente, o Swing incentiva você a atualizar seus componentes no thread de envio de eventos AWT. Este é o tópico que lida naturalmente com coisas como cliques em botões. Se você atualizar um componente em resposta a um evento (como nosso botão de contador e rótulo no“Eventos de Ação”), estás pronto. A ideia é que, se todos os outros threads em seu aplicativo enviarem atualizações de UI para o único thread de envio de eventos, nenhum componente poderá ser afetado negativamente por alterações simultâneas e possivelmente conflitantes.

Um exemplo comum de threading em aplicativos gráficos é o botão giratório animado que fica na tela enquanto você espera o download de um arquivo grande. Mas e se você ficar impaciente? E se parecer que o download falhou, mas o botão giratório ainda estiver funcionando? Se a sua tarefa de longa duração estiver usando o thread de expedição de eventos, o usuário não poderá clicar no botão Cancelar ou realizar

qualquer ação. Tarefas de longa duração devem ser tratadas por threads separados que podem ser executados em segundo plano, deixando seu aplicativo responsivo e disponível. Mas então como atualizamos a IU quando o thread em segundo plano termina? Swing tem uma classe auxiliar pronta para você.

SwingUtilities e atualizações de componentes

Você pode usar a classe SwingUtilities de qualquer thread para realizar atualizações nos componentes da UI de

maneira segura e estável. Existem dois métodos estáticos que você pode usar para se comunicar com sua IU:

•

invocarAndWait()

•

invocarMais tarde()

Como seus nomes indicam, o primeiro método executa algum código de atualização da IU e faz o thread atual aguardar a conclusão da atualização antes de continuar. O

segundo método transfere algum código de atualização da UI para o thread de envio de eventos e então retoma imediatamente a execução no thread atual. (O

encadeamento de despacho de eventos às vezes é chamado de fila de despacho de eventos. Você pode anexar eventos ou atualizações e o encadeamento de despacho de eventos chegará até eles aproximadamente na ordem em que foram adicionados, como em uma fila.) Qual deles você usa realmente depende se seu thread em segundo plano precisa saber o estado da IU antes de continuar. Por exemplo, se você estiver adicionando um novo botão à sua interface, você pode querer usar invocaAndWait() para que, quando o thread em segundo plano continuar, você possa ter certeza de que atualizações futuras realmente terão um botão para atualizar.

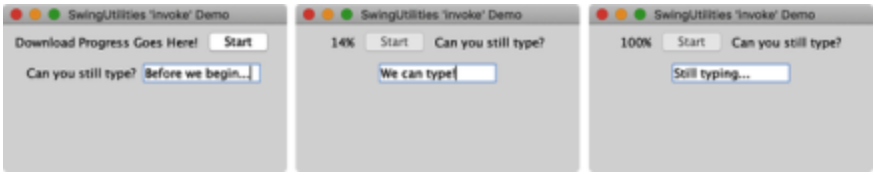
Se você não está tão preocupado com a atualização de algo, mas ainda deseja que seja tratado com segurança

pelo encadeamento de despacho, invocaLater() é perfeito.

Pense em atualizar uma barra de progresso enquanto um arquivo grande está sendo baixado. Seu código pode disparar várias atualizações à medida que o download é concluído. Você não precisa esperar que essas atualizações gráficas terminem antes de retomar o download. Se uma atualização de progresso for atrasada ou muito próxima de uma segunda atualização, não haverá nenhum dano real. Mas você não quer que uma interface gráfica ocupada interrompa seu download – especialmente se o servidor for sensível a pausas.

Veremos vários exemplos desse tipo de interação rede/UI emCapítulo 13, mas vamos falsificar algum tráfego de rede e atualizar um pequeno rótulo para mostrar o SwingUtilities. Podemos configurar um botão Iniciar que atualizará um rótulo de status com uma simples exibição de porcentagem e iniciará um thread em segundo plano que ficará suspenso por um segundo e, em seguida, aumentará o progresso.

Cada vez que o thread for ativado, ele atualizará o rótulo usando InvokeLater() para definir corretamente o texto do rótulo. Primeiro, vamos configurar nossa demonstração:



pacote ch12.examples;

classe pública ProgressDemo {

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

Quadro JFrame = new JFrame("SwingUtilities 'invoca' Demo");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setLayout(novo FlowLayout());

frame.setSize(300, 180);

Etiqueta JLabel = new JLabel("O progresso do download vai aqui!", JLabel.CENTRO);

Pretendente de thread = new Thread(new ProgressPretender(label));

JButton simpleButton = new JButton("Iniciar");

simpleButton.addActionListener(e -> {

simpleButton.setEnabled(falso);

pretender.start();

});

JLabel checkLabel = new JLabel("Você ainda consegue digitar?"); JTextField checkField = new JTextField(10);

frame.add(rótulo);

frame.add(simpleButton);

frame.add(checkLabel);

frame.add(checkField);

frame.setVisible (verdadeiro);
```

}

}

A maior parte disso deve parecer familiar, mas veja como criamos nosso tópico.

Passamos um novo ProgressPretender() como argumento para nosso construtor Thread. Poderíamos ter dividido essa chamada em partes separadas, mas como não nos referimos diretamente ao nosso objeto ProgressPretender novamente, podemos continuar com essa abordagem mais organizada e densa. No entanto, nos referimos ao próprio thread, então criamos uma variável adequada para ele. Podemos então iniciar nosso thread em execução no ActionListener para nosso botão. Também

desabilitamos nosso botão Iniciar nesse ponto. Não queremos que o usuário tente (re)iniciar um thread que já está em execução!

Adicionamos um campo de texto para você digitar. Enquanto o progresso está sendo atualizado, seu aplicativo deve continuar respondendo à entrada do usuário, como digitação. Tente! O campo de texto não está conectado a nada, mas você poderá inserir e excluir texto enquanto observa o contador de progresso subir lentamente, conforme mostrado emFigura 12-31.

Figura 12-31. Atualizações thread-safe para um rótulo de progresso Então, como atualizamos esse rótulo sem bloquear o aplicativo? Vejamos a classe ProgressPretender e inspecionamos o método run():

pacote ch12.examples;

classe ProgressPretender implementa Runnable {

```
Etiqueta JLabel;

progresso interno;

public ProgressPretender(rótulo JLabel) {

este.label = rótulo;

progresso = 0;

}

execução nula pública() {

enquanto (progresso <= 100) {

SwingUtilities.invokeLater(

() -> label.setText(progresso + "%");

);

tentar {

Thread.sleep(1000);

} catch (InterruptedException, ou seja) {

System.err.println("Alguém nos interrompeu. Ignorando o
download."); quebrar;

}

progresso++;

}

}
```

```
}
```

Nesta classe, armazenamos o rótulo passado ao nosso construtor para sabermos onde exibir nosso progresso atualizado. O método run() tem três etapas básicas: 1) atualizar o rótulo, 2) suspender por 1.000 milissegundos e 3) incrementar nosso progresso.

Na etapa 1, o argumento lambda que passamos para invocaLater() é baseado na interface Runnable emCapítulo 9. Poderíamos ter usado uma classe interna ou uma classe interna anônima, mas para uma tarefa tão simples, um lambda é perfeito. O

corpo lambda atualiza o rótulo com nosso valor de progresso atual. O thread de envio de eventos executará o lambda. Essa é a mágica que deixa o campo de texto responsivo, mesmo que nosso thread de “progresso” esteja inativo a maior parte do tempo.

A etapa 2 é o thread padrão suspenso. O método sleep() sabe que pode ser interrompido, então o compilador garantirá que você forneça um bloco try/catch como fizemos acima. Há muitas maneiras de lidar com a interrupção, mas neste caso optamos simplesmente por sair do loop.

Finalmente, incrementamos nosso contador de progresso e reiniciamos todo o processo. Assim que atingirmos 100, o loop termina e nosso rótulo de progresso deve parar de mudar. Se você esperar pacientemente, verá o valor final. O próprio aplicativo deve permanecer ativo. Você ainda pode digitar no campo de texto. Nosso download foi concluído e está tudo bem com o mundo!

Temporizadores

A biblioteca Swing inclui um temporizador projetado para funcionar no espaço da UI.

A classe javax.swing.Timer é bastante direta. Ele espera um período de tempo especificado e então dispara um evento de ação (o mesmo tipo de evento que ocorre ao clicar em um botão). Ele pode disparar essa ação uma vez ou repetidamente. Você encontrará muitos motivos para usar temporizadores com aplicativos gráficos. Além de fornecer uma maneira alternativa de conduzir um loop de animação, você pode querer cancelar automaticamente alguma ação, como carregar um recurso de rede se estiver demorando muito. Por outro lado, você pode colocar um pequeno botão giratório ou caixa de diálogo “aguarde” para informar ao usuário que a operação está em andamento. Talvez você queira desativar um prompt de diálogo se o usuário não responder dentro de um intervalo de tempo especificado. O Timer do Swing pode lidar com todos esses cenários.

Animação com Temporizador

Vamos modificar nossas maçãs voadoras de“Revisitando Animação com Threads”e

tente implementar a animação com uma instância de Timer. A classe Timer cuida desse detalhe para nós. Ainda podemos usar nosso método step() na classe Apple desde nossa primeira passagem na animação. Precisamos apenas alterar o método start e manter uma variável adequada para o cronômetro:

público estático final int STEP = 40; //duração do quadro em milissegundo s

Animação do temporizadorTimer;

```
// outras declarações de membros ...

void startAnimation() {

if (animationTimer == null) {

animaçãoTimer = novo Timer (STEP, este);

AnimationTimer.setActionCommand("repintar");

animaçãoTimer.setRepeats(true);

animaçãoTimer.start();

} else if (!animationTimer.isRunning()) {

animaçãoTimer.restart();

}

}

// Outros métodos ...

public void actionPerformed(evento ActionEvent) {

if (animando &&
event.getActionCommand().equals("repaint")) {

System.out.println("Temporização de passos " +
apples.size() + "maçãs"); para (maçã a: maçãs) {

um passo();

detectarColisões(a);

}

repintar();
```

```
cullFallenApples();

}

}
```

Há duas coisas interessantes nessa abordagem. É definitivamente mais fácil de ler porque não somos responsáveis pelas pausas entre as ações. Criamos o Timer passando para o construtor o intervalo de tempo entre os eventos e um ActionListener para receber os eventos – nossa classe Field neste caso. Damos ao cronômetro um comando de ação simples, mas único, transformamos-o em um cronômetro de repetição e iniciamos!

A outra coisa interessante é específica para Swing e aplicações gráficas: javax.swing.Timer dispara seus eventos de ação no thread de envio de eventos. Você não precisa agrupar nenhuma de suas respostas emvocationAndWait() ou

invocaLater(). Basta colocar seu código baseado em timer no método

actionPerformed() de um ouvinte anexado e pronto!

Como vários componentes geram objetos ActionEvent, tomamos algumas precauções contra colisões, definindo o atributo actionCommand para nosso temporizador. Esta etapa não é estritamente necessária em nosso caso, mas deixa espaço para a classe Field lidar com outros eventos no futuro sem interromper nossa animação.

Outros usos do temporizador

Aplicativos maduros e sofisticados têm uma variedade de pequenos momentos em que ajuda ter um cronômetro único. Nosso jogo da maçã é simples em comparação com a maioria dos aplicativos ou jogos comerciais, mas mesmo aqui podemos adicionar um pouco de “realismo” com um cronômetro: depois de lançar uma maçã, podemos fazer uma pausa antes de permitir que o físico atire outra maçã. Talvez o físico tenha que se abaixar e pegar outra maçã de um balde antes de mirar ou jogar.

Esse tipo de atraso é outro local perfeito para um Timer.

Podemos adicionar esta pausa ao trecho de código da classe Field onde jogamos a maçã:

```
public void startTossFromPlayer(Físico físico) {

if (!animando) {

System.out.println("Iniciando animação!");

animando = verdadeiro;

startAnimation();

}

if (animando) {

// Verifique se temos uma maçã para jogar

if (físico.aimingApple! = null) {

Maçã maçã = físico.takeApple();

apple.toss(físico.aimingAngle, físico.aimingForce);

maçãs.add(maçã);
```

```
Timer appleLoader = novo Timer(800, físico);

appleLoader.setActionCommand("Nova Apple");

appleLoader.setRepeats(falso);

appleLoader.start();

}

}

}
```

Observe que desta vez definimos o cronômetro para ser executado apenas uma vez com a chamada setRepeats(false). Isto significa que depois de pouco menos de um segundo, um único evento será disparado para o nosso físico. A classe Physicist, por sua vez, precisa adicionar a parte implementa Action Listener à definição da classe e incluir uma função actionPerformed() apropriada, assim:

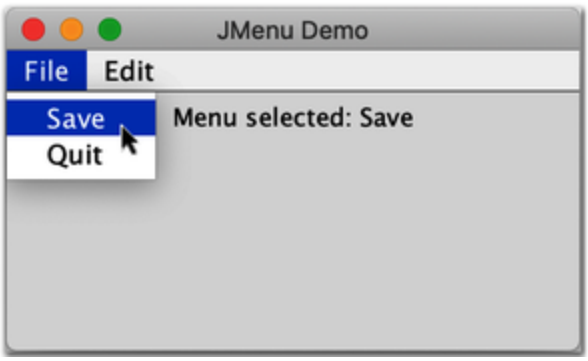
```
// outras importações ...

importar java.awt.event.ActionEvent;

importar java.awt.event.ActionListener;

classe pública Físico implementa ActionListener {

// Coisas atuais de Físico...

// Novo manipulador de eventos para obter uma nova maçã

public void actionPerformed(ActionEvent e) {
```

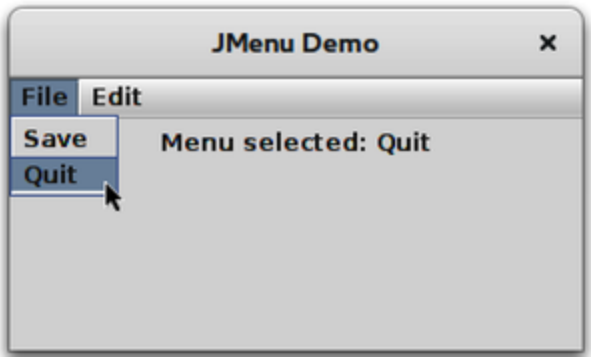
```
if (e.getActionCommand().equals("Nova Apple")) {

getNovaApple();

if (campo! = nulo) {

field.repaint();

}

}

}

}
```

Usar o Timer não é a única maneira de realizar tais tarefas, mas no Swing, a combinação de eventos cronometrados eficientes e o uso automático do thread de despacho de eventos faz com que valha a pena considerá-lo. No mínimo, facilita a prototipagem. Você sempre pode voltar e refatorar seu aplicativo para usar código de threading personalizado, se necessário.

Mas espere, há mais

Como observamos no início do capítulo, há muito mais discussões, tópicos e explorações disponíveis no mundo dos aplicativos gráficos Java. Java tem um pacote





inteiro dedicado ao armazenamentoPreferências de usuário, por exemplo. tem um livro inteiro de Jonathan Knudsen dedicado a *Gráficos Java 2D*. Deixaremos que você faça essa exploração, mas gostaríamos de abordar pelo menos alguns tópicos principais nos quais vale a pena focar primeiro, se você tiver planos para um aplicativo de desktop.

Cardápios

Embora não seja tecnicamente necessário, a maioria dos aplicativos de desktop possui um menu de tarefas comuns para todo o aplicativo, como salvar arquivos alterados ou definir preferências. Aplicativos com recursos específicos, como planilhas, podem ter menus para classificar os dados em uma coluna ou seleção. As classes JMenu, JMenuBar e JMenuItem ajudam você a adicionar essa funcionalidade aos seus aplicativos Swing. Os menus ficam dentro de uma barra de menus e os itens de menu vão dentro dos menus. Swing tem três classes de itens de menu pré-construídas: JMenuItem para entradas básicas de menu, JCheckBoxMenuItem para itens de opção e JRadioButtonMenuItem para itens de menu agrupados, como a fonte ou tema de cores atualmente selecionado. A classe JMenu é em si um item de menu válido para que você possa construir menus aninhados. JMenuItem se comporta como um botão (assim como seus compatriotas de rádio e caixa de seleção), e você pode capturar eventos de menu usando os mesmos ouvintes.

Figura 12-32mostra um exemplo de uma barra de menu simples preenchida com alguns menus e itens.

Figura 12-32. JMenu e JMenuItem no macOS e Linux

Aqui está o código fonte desta demonstração:

```
pacote ch12.examples;

importar javax.swing.*;

importar java.awt.*;

importar java.awt.event.ActionEvent;

importar java.awt.event.ActionListener;

classe pública MenuDemo estende JFrame implementa
ActionListener {

Resultados JLabelLabel;

public MenuDemo() {

super("Demonstração JMenu");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

setLayout(novo FlowLayout());

setSize(300, 180);

resultsLabel = new JLabel("Clique em um item de
menu!");

add(resultadosLabel);

// Agora vamos criar alguns menus e preenchê-los

JMenu arquivoMenu = new JMenu("Arquivo");

JMenuItem saveItem = new JMenuItem("Salvar");

saveItem.addActionListener(this);
```

```
arquivoMenu.add(saveItem);
JMenuItem quitItem = new JMenuItem("Sair");
quitItem.addActionListener(este);
arquivoMenu.add(quitItem);
JMenu editMenu = new JMenu("Editar");
JMenuItem cutItem = new JMenuItem("Cortar");
cutItem.addActionListener(this);
editMenu.add(cortarItem);
JMenuItem copyItem = new JMenuItem("Copiar");
copyItem.addActionListener(this);
editMenu.add(copiarItem);
JMenuItem pasteItem = new JMenuItem("Colar");
pasteItem.addActionListener (este);
editMenu.add(pasteItem);
// E finalmente construa um JMenuBar para a aplicação
JMenuBar mainBar = new JMenuBar();
mainBar.add(arquivoMenu);
mainBar.add(editarMenu);
setJMenuBar(mainBar);
}
```

```
public void actionPerformed(evento ActionEvent) {

resultsLabel.setText("Menu selecionado: " +
event.getActionCommand());

}

public static void main(String args[]) {

Demonstração MenuDemo = new MenuDemo();

demo.setVisible(verdadeiro);

}

}
```

Obviamente não fazemos muito com as ações dos itens de menu aqui, mas elas ilustram como você pode começar a construir as partes esperadas de um aplicativo profissional.

Modais e pop-ups

Os eventos permitem que o usuário chame sua atenção, ou pelo menos a atenção de algum método em sua aplicação. Mas e se você precisar chamar a atenção do usuário?

Um mecanismo de UI popular para esta tarefa é a janela pop-up. Freqüentemente, você ouvirá essa janela ser chamada de modal, caixa de diálogo ou até mesmo caixa de diálogo modal. O termo diálogo vem do fato de que

esses pop-ups apresentam algumas informações ao usuário e esperam – ou exigem – uma resposta. Talvez este processo rápido de perguntas e respostas não seja tão grandioso quanto um simpósio socrático, mas ainda assim. O nome modal refere-se ao fato de que algumas dessas caixas de diálogo que exigem uma resposta irão, na verdade, desabilitar o resto do aplicativo – colocando-o em um modo restrito – até que você forneça essa resposta.

Você pode ter experimentado essa caixa de diálogo em outros aplicativos de desktop.

Se o seu software exige que você se mantenha atualizado com a versão mais recente, por exemplo, ele pode “esmaecer” o aplicativo, indicando que você não pode usá-lo, e então mostrar uma caixa de diálogo modal com um botão que inicia o processo de atualização.

O termo pop-up é mais geral. Embora você certamente possa ter pop-ups modais, também pode ter pop-ups simples (“sem janela restrita”) que não impedem o uso do restante do aplicativo. Pense em uma caixa de diálogo de pesquisa em um aplicativo de processamento de texto que você pode deixar disponível e simplesmente deslizar para o lado da janela principal.

Swing fornece uma classe JDialog simples que você pode usar para criar janelas de diálogo personalizadas. Para interações pop-up típicas com seus usuários, incluindo alertas, confirmações e caixas de diálogo de entrada, a classe JOptionPane possui alguns atalhos realmente úteis. Para alertas simples e mensagens de erro, você pode usar a chamada showMessageDialog(). Este tipo de caixa de diálogo inclui um título personalizável, algum texto e um botão para confirmar (e descartar) o pop-up.

Se você precisa que o usuário escolha sim ou não, showConfirmDialog() é perfeito. E se você precisar de respostas curtas e baseadas em texto do usuário, você vai querer usar showInputDialog().Figura 12-33mostra um exemplo dessas três caixas de diálogo.

Figura 12-33. As principais variações das janelas pop-up JOptionPane

Para criar uma caixa de diálogo de mensagem, você deve fornecer quatro argumentos.

O primeiro argumento refere-se ao quadro ou janela “proprietário” do pop-up.

JOptionPane tentará centralizar a caixa de diálogo sobre seu proprietário quando mostrado. Você também pode especificar null para este argumento, o que informa ao JOptionPane que não há janela primária, portanto centralize o pop-up na tela do usuário. O segundo e terceiro argumentos são Strings para a mensagem e o título do diálogo, respectivamente. O argumento final indica o “tipo” de pop-up, que afeta principalmente o ícone que você vê. Você pode especificar vários tipos:

•

MENSAGEM DE ERRO, ícone vermelho Parar

•

INFORMATION_MESSAGE, Duque6ícone

•

MENSAGEM DE AVISO, ícone de triângulo amarelo

•

QUESTION_MESSAGE, ícone do duque

•

PLAIN_MESSAGE, nenhum ícone

Volte para o seu jshell e tente criar um pop-up de mensagem. Você pode usar a útil opção nula para o proprietário:

jshell> importar javax.swing.*

jshell> JOptionPane.showMessageDialog(null, "Olá", "Alerta jshell",

...> JOptionPane.ERROR_MESSAGE)

Outra tarefa comum para pop-ups é verificar a intenção do usuário. Muitos aplicativos perguntam se você tem certeza de que deseja encerrar ou excluir algo. JOptionPane ajuda você. Você pode experimentar esta caixa de diálogo de confirmação no jshell: jshell> JOptionPane.showConfirmDialog(null, "Tem certeza?") US$ 18 ==> 0

A versão de dois argumentos do método showConfirmDialog() produz um pop-up com os botões Sim, Não e Cancelar. Você pode determinar qual resposta o usuário selecionou mantendo o valor de retorno (um int). Clicamos no botão Sim, que retorna 0, mas você não precisa memorizar os valores de retorno. JOptionPane possui constantes que cobrem as diversas respostas:

•

SIM_OPÇÃO

•

NENHUMA OPÇÃO

•

CANCEL_OPTION

•

OK_OPÇÃO

•

CLOSED_OPTION

JOptionPaneretorna o valor CLOSED_OPTION se o usuário fechar a caixa de diálogo usando os controles da janela em vez de clicar em qualquer um dos botões disponíveis na caixa de diálogo. Você também tem um mínimo de controle sobre esses botões.

Você pode usar uma versão de quatro argumentos que tenha um título e uma opção de botão com um dos seguintes valores:

•

SIM_NÃO_OPÇÃO

•

OK_CANCEL_OPTION

•

YES_NO_CANCEL_OPTION

Na maioria das situações, ter “Não” e “Cancelar” na mesma caixa de diálogo confunde os usuários. Recomendamos usar uma das duas primeiras opções. (O usuário sempre

pode fechar a caixa de diálogo usando os controles padrão da janela se não quiser fazer uma seleção.) Vamos tentar criar uma caixa de diálogo Sim/Não:

jshell> int resposta = JOptionPane.showConfirmDialog(null,

...> "Tem certeza?", "Confirmar", JOptionPane.YES_NO_OPTION) resposta ==> 1

jshell> if (resposta == JOptionPane.NO_OPTION)

...> System.out.println("Eles recusaram")

Eles recusaram

Alguns pop-ups pedem uma entrada rápida. Você pode usar o método

showInputDialog() para fazer uma pergunta e permitir que o usuário digite uma resposta. Essa resposta (uma String) pode ser armazenada da mesma forma que você mantém a opção de confirmação. Vamos tentar mais um pop-up em jshell: jshell> String pin = JOptionPane.showInputDialog(null, "Por favor, insira seu PIN:")

pino ==> "1234"

As caixas de diálogo de entrada são úteis para solicitações únicas, mas não as recomendamos se você tiver uma série de perguntas a fazer ao usuário. Você deve manter os modais confinados a tarefas rápidas e pouco frequentes. Eles interrompem o usuário – intencionalmente. Às vezes, essa interrupção é exatamente o que você precisa. No entanto, se você abusar da atenção do usuário, provavelmente irá incomodá-lo e ele aprenderá a simplesmente ignorar todos os pop-ups do seu aplicativo.

O arquivo ch12/examples/ModalDemo.java contém um pequeno aplicativo que pode criar uma variedade de caixas de diálogo modais. Sinta-se à vontade para brincar e experimentar os diferentes tipos de mensagens ou opções de botões de confirmação. E

não tenha medo de modificar esses aplicativos de exemplo! Às vezes, ajustar um aplicativo simples e recompilar é mais fácil do que tentar digitar exemplos de múltiplas linhas no jshell.

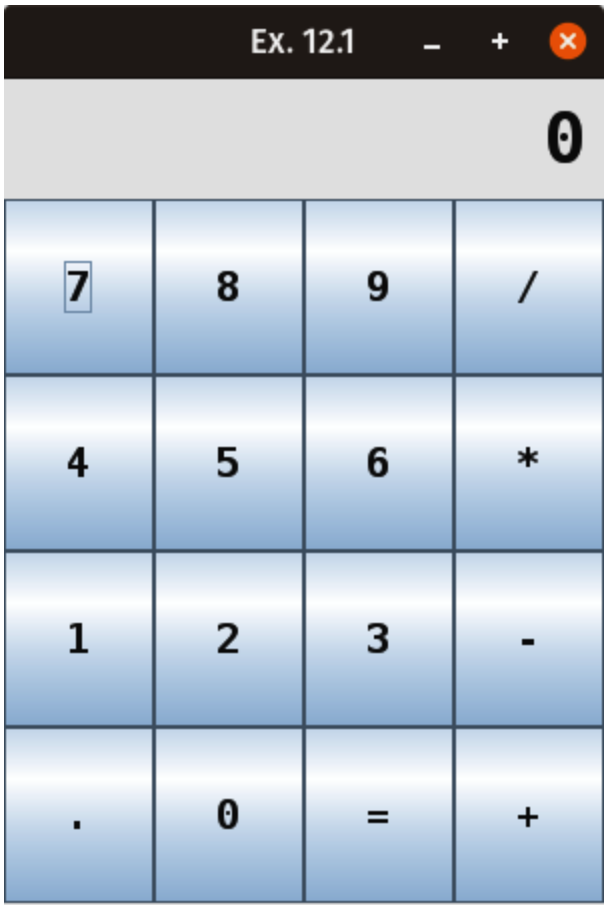
Interface do usuário e experiência do usuário

Este foi um tour rápido por alguns dos elementos de UI mais comuns para aplicativos de desktop, como JButton, JLabel e JTextField. Discutimos como organizar esses componentes usando gerenciadores de layout em contêineres e introduzimos vários outros componentes.

É claro que os aplicativos de desktop são apenas parte da história.Capítulo 13aborda conceitos básicos de rede, incluindo obtenção de conteúdo da Web e aplicativos cliente/servidor simples.

Perguntas de revisão

1. Qual componente você usaria para exibir algum texto ao usuário?



2. Quais componentes você usaria para permitir que o usuário insira texto?

3. Que evento o clique em um botão gera?

4. Qual ouvinte você deve anexar ao JList se quiser saber quando o usuário altera o item selecionado?

5. Qual é o gerenciador de layout padrão para JPanel?

6. Qual thread é responsável pelo processamento de eventos em Java?

7. Qual método você usaria para atualizar um componente como JLabel após a conclusão de uma tarefa em segundo plano?

8. Qual contêiner contém objetos JMenuItem?

Exercícios de código

1. Crie uma interface de calculadora com botões e exibição de texto. Você pode usar a classe inicial Calculadora na pasta ch12/exercises. Estende o JFrame e implementa a interface ActionListener. O elemento de exibição deve ficar na parte superior da calculadora e mostrar o texto justificado à direita. Os botões devem incluir os dígitos de 0 a 9, ponto decimal, adição, subtração,

multiplicação, divisão e “igual” para mostrar os resultados. Você pode ver como deve ser emFigura 12-34.

Figura 12-34. Um exemplo de interface de calculadora

Não se preocupe em conectar os botões para fazer a calculadora funcionar –

ainda. Abordaremos isso no exercício avançado.

2. O jogo de lançamento de maçãs na pasta ch12/exercises/game possui controles deslizantes e botões para mirar e lançar maçãs. Neste momento, essas maçãs simplesmente voam em arco e eventualmente saem dos limites da

nossa janela. Adicione o código necessário para capturar colisões entre uma maçã e um obstáculo, como uma

árvore ou uma cerca viva. Sua solução deve remover a maçã e o obstáculo e depois atualizar a tela.

Exercícios Avançados

1. Pegue o shell visual que você criou para uma calculadora no primeiro exercício de código e conecte os botões para torná-lo funcional. Clicar nos botões numéricos deve colocar o dígito correspondente no display. Clicar em um botão de operação (como adição ou divisão) deve armazenar a operação a ser executada e permitir que o usuário insira um segundo número. Clicar no botão

"=" deve mostrar os resultados da operação.

Este exercício reúne diversas discussões de capítulos anteriores. Faça alterações incrementais e não tenha medo de olhar"Exercícios Avançados"para obter dicas sobre como proceder.

1Se você está curioso sobre este tópico e deseja ver os bastidores de um aplicativo Java comercial para desktop, a JetBrains publica oCódigo fontepara a edição da comunidade.

2O prefixo do pacote javax foi introduzido anteriormente pela Sun para acomodar pacotes que foram distribuídos com Java, mas não eram "principais". A decisão foi modestamente controversa, mas o javax permaneceu e também foi usado com outros pacotes.

3Você precisará iniciar o jshell no diretório de nível superior que contém os exemplos compilados para o livro. Se você estiver usando o IntelliJ IDEA, poderá iniciar seu terminal e alternar diretórios usando cd

out/production/LearningJava6e e, em seguida, iniciar o jshell.

4À medida que criamos componentes Swing para uso nesses exemplos de jshell, estaremos omitindo grande parte da saída resultante por questão de espaço. jshell imprime muitas informações sobre cada componente, embora também use reticências quando as coisas ficam muito extremas. Não se assuste se você vir detalhes extras sobre os atributos de um elemento enquanto estiver experimentando.

5Devemos também observar que existem muitos projetos de código aberto com componentes ainda mais sofisticados para lidar com coisas como realce de sintaxe em texto, vários auxiliares de seleção, gráficos e tabelas e entradas compostas, como seletores de data ou hora.

6“Duke” é o mascote oficial de Java. Você pode descobrir mais e[ma wiki do OpenJDK.]

Capítulo 13. Programação de rede em Java

Quando você pensa na web, provavelmente pensa em aplicativos e serviços baseados na web. Se for solicitado que você se aprofunde, considere ferramentas como navegadores e servidores da Web que oferecem suporte a esses aplicativos e movem dados pela rede. Neste capítulo, veremos como Java interage com serviços web.

Também daremos uma olhada nos bastidores e discutiremos algumas das classes de rede de nível inferior do pacote java.net.

Localizadores Uniformes de Recursos

Um Uniform Resource Locator (URL) aponta para um objeto na Internet. É uma string de texto que identifica um item, informa onde encontrá-lo e especifica um método para se comunicar com ele ou recuperá-lo de sua fonte. Uma URL pode apontar para qualquer tipo de fonte de informação: dados estáticos, como um arquivo em um sistema de arquivos local, um servidor web ou um site FTP. Pode apontar para um objeto mais dinâmico, como um feed de notícias RSS ou um registro em um banco de dados. URLs também podem referir-se a outros recursos, como endereços de e-mail.

Como existem muitas maneiras diferentes de localizar um item na Internet e diferentes meios e transportes exigem diferentes tipos de informações, os URLs podem ter vários formatos. A forma mais comum tem quatro componentes, conforme mostrado emFigura 13-1: um host ou servidor de rede, o nome do item, sua localização nesse host e um protocolo pelo qual o host deve se comunicar.

Figura 13-1. Elementos comuns de um URL

protocolo(também chamado de “esquema”) é um identificador como http, https ou ftp; hostname geralmente é um host da Internet e um nome de domínio; e os componentes de caminho e recurso formam um caminho exclusivo que identifica o objeto nesse host.

Variantes deste formulário incluem informações extras no URL. Por exemplo, você pode especificar identificadores de fragmentos (aqueles sufixos que começam com um caractere “#”) que fazem referência a seções dentro de documentos. Também existem outros tipos de URLs mais especializados, como URLs “mailto” para endereços de email ou URLs para endereçar coisas como componentes de banco de dados. Esses tipos de localizadores podem não seguir esse formato com precisão, mas geralmente contêm um protocolo, um host e um caminho. Alguns são mais apropriadamente chamados de Identificadores Uniformes de Recursos, ou URIs, que podem especificar mais informações sobre o nome ou a localização de um recurso. URLs são um subconjunto de URIs.

Como a maioria dos URLs tem a noção de hierarquia ou caminho, às vezes falamos de um URL relativo a outro URL, chamado URL base. Nesse caso, estamos usando o URL

base como ponto de partida e fornecendo informações adicionais para direcionar um objeto relativo a esse URL. Por exemplo, o URL base pode apontar para um diretório em um servidor web, e um URL relativo pode nomear um arquivo específico nesse diretório ou em um subdiretório.

A classe URL

A classe Java java.net.URL representa um endereço URL e fornece uma API simples para acessar recursos da web, como documentos e aplicativos em servidores. Ele pode usar um conjunto extensível de protocolos e manipuladores de conteúdo para realizar a comunicação necessária e, em teoria, até mesmo a conversão de dados. Com a classe URL, um aplicativo pode abrir uma

conexão com um servidor e recuperar conteúdo com apenas algumas linhas de código.

Uma instância da classe URL gerencia todas as informações do componente dentro de uma string de URL e fornece métodos para recuperar o objeto que identifica. Podemos construir um objeto URL a partir de uma string completa ou de partes componentes: tentar {

URL aDoc =

nova
URL("http://foo.bar.com/documents/homepage.html");

URL mesmoDoc =

nova URL("http","foo.bar.com","
/documents/homepage.html");

} catch (MalformedURLException e) {

//Algo errado com nossa URL

}

Esses dois objetos URL apontam para o mesmo recurso de rede, o documento homepage.html no servidor foo.bar.com. Não podemos saber se o recurso realmente existe e está disponível até tentarmos acessá-lo. Um novo objeto URL contém apenas dados sobre a localização do objeto e como acessá-lo. A criação de um objeto URL não estabelece conexões de rede.

Observação

A Oracle descontinuou o construtor de URL no Java 20. A descontinuação não remove métodos ou classes, mas

significa que você deve considerar outros meios de atingir seu objetivo. O Javadoc para itens obsoletos geralmente inclui alternativas sugeridas.

Nesse caso, a classe URI possui um código de validação melhor, então a Oracle recomenda o novo URI("http://your.url/").toURL() como um substituto.

Se você estiver usando Java 20 ou posterior, sinta-se à vontade para atualizar os exemplos de código para usar URI se quiser se livrar dos avisos de descontinuação do compilador. Como esta é uma descontinuação recente, você ainda verá o construtor de URL amplamente usado em exemplos online.

Podemos examinar as partes da URL com os métodos getProtocol(), getHost() e getFile(). Também podemos compará-lo a outra URL com o método sameFile() (um nome infeliz para algo que pode não apontar para um arquivo), que determina se duas URLs apontam para o mesmo recurso. Não é infalível, mas sameFile() faz mais do que comparar as strings de URL quanto à igualdade; leva em consideração a possibilidade de um servidor ter vários nomes, além de outros fatores.

Ao criar uma URL, Java analisa a especificação da URL para identificar o componente do protocolo. Em seguida, ele tenta combinar o que analisa do seu URL com um manipulador de protocolo. Um manipulador de protocolo é essencialmente um auxiliar que pode falar o protocolo fornecido e recuperar um recurso seguindo as regras do protocolo. Se o protocolo da URL não fizer sentido ou se o Java não conseguir encontrar um manipulador de protocolo compatível, o construtor da URL

lançará uma MalformedURLException.

Java fornece manipuladores de protocolo de URL para http, https (HTTP seguro) e ftp, bem como URLs de arquivos locais e URLs jar que se referem a arquivos dentro de arquivos JAR. Java também fornece a estrutura de baixo nível necessária para bibliotecas de terceiros adicionarem suporte para outros tipos de URLs.

Transmitir dados

A maneira mais geral e de nível mais baixo de recuperar dados de uma URL é solicitar um InputStream da URL chamando openStream(). Obter dados como um fluxo também pode ser útil se você quiser receber atualizações contínuas de uma fonte de informações dinâmica. Infelizmente, você mesmo terá que analisar o conteúdo deste fluxo. Nem todos os tipos de URLs suportam o método openStream() porque nem todos os tipos de URLs se referem a dados concretos; você receberá uma UnknownServiceException se o URL não.

O código a seguir (uma simplificação do arquivo ch13/examples/Read.java) imprime o conteúdo de um arquivo HTML de um servidor web imaginário:

```
tentar {

URL url = nova URL("http://some.server/index.html");

BufferedReader bin = novo BufferedReader(

novo InputStreamReader(url.openStream()));

Linha de corda;

while ((linha = bin.readLine()) != nulo) {

System.out.println(linha);
```

```
}

bin.close();

} catch (Exceção e) {

e.printStackTrace();

}
```

Neste trecho, solicitamos um InputStream de nossa url com openStream() e o envolvemos em um BufferedReader para ler as linhas de texto. Como especificamos o protocolo http na URL, recrutamos os serviços de um manipulador de protocolo HTTP.

Ainda não falamos sobre manipuladores de conteúdo. Como estamos lendo diretamente do fluxo de entrada, não precisamos de um manipulador de conteúdo para transformar o conteúdo.

Obtendo o conteúdo como um objeto

Como dissemos anteriormente, openStream() é a forma mais geral de acessar conteúdo da web, mas deixa a análise dos dados para o programador. A classe URL

suporta um mecanismo de manipulação de conteúdo mais sofisticado e conectável, mas a comunidade Java nunca padronizou realmente os manipuladores reais, portanto sua utilidade é limitada.

Muitos desenvolvedores têm curiosidade em carregar objetos pela rede porque precisam carregar imagens de URLs. Java fornece algumas abordagens alternativas para realizar essa tarefa. A abordagem mais simples é usar a classe

javax.swing.ImageIcon que possui um construtor que aceita uma URL:

//arquivo: ch13/examples/IconLabel.java

URL favorito = novo URL("https://www.oracle.com/.../favicon-192.png");
ImageIcon imagem1 = novo ImageIcon(favorito);

JLabel iconLabel = new JLabel(image1);

// iconLabel pode ser colocado em qualquer painel, assim como outros rótu los

Se precisar transformar um fluxo de rede em algum outro tipo de objeto, você pode consultar o método getContent() da classe URL. Você pode precisar escrever seu próprio manipulador. Para esse tópico avançado, recomendamos *Programação de Rede*

Java por Elliotte Rusty-Harold.

Gerenciando conexões

Ao chamar openStream() em uma URL, Java consulta o manipulador de protocolo e uma conexão é feita com o servidor ou local remoto. As conexões são representadas por um objeto URLConnection, cujos subtipos gerenciam diferentes comunicações específicas de protocolo e oferecem metadados adicionais sobre a origem. A classe HttpURLConnection, por exemplo, lida com solicitações web básicas e também adiciona alguns recursos específicos de HTTP, como a interpretação de mensagens

“404 Not Found” e outros erros de servidor web. Falaremos mais sobre

HttpURLConnection posteriormente neste capítulo.

Podemos obter uma URLConnection de nossa URL diretamente com o método openConnection(). Uma das coisas que podemos fazer com URLConnection é perguntar o tipo de conteúdo do objeto antes de ler os dados. Por exemplo:

```
Conexão URLConnection = myURL.openConnection();

String mimeType = connection.getContentType();

InputStream in = connection.getInputStream();
```

Apesar do nome, um objeto URLConnection é inicialmente criado em um estado bruto e desconectado. Neste exemplo, a conexão de rede não foi realmente iniciada até chamarmos o método getContentType(). O URLConnection não se comunica com a fonte até que os dados sejam solicitados ou seu método connect() seja invocado explicitamente. Antes da conexão, podemos configurar parâmetros de rede e fornecer detalhes específicos do protocolo. Por exemplo, podemos definir tempos limite na conexão inicial com o servidor e nas tentativas de leitura:

```
Conexão URLConnection = myURL.openConnection();

conexão.setConnectTimeout(10000); // milissegundos

conexão.setReadTimeout(10000); // milissegundos

InputStream in = connection.getInputStream();
```

Como veremos em“Usando o método POST”, podemos obter informações específicas do protocolo convertendo URLConnection em seu subtipo específico.

Conversando com aplicativos da Web

Os navegadores da Web são clientes universais para aplicativos da Web. Eles recuperam documentos para exibição e servem como interface de usuário, principalmente por meio do uso de HTML, JavaScript e documentos vinculados, como imagens. Nesta seção, escreveremos código Java do lado do cliente que usa HTTP por meio da classe URL. Essa combinação nos permite trabalhar diretamente com aplicações web usando operações GET e POST para recuperar e enviar dados.

A principal tarefa que discutimos aqui é o envio de dados ao servidor, especificamente dados codificados em formulário HTML. Os navegadores codificam os pares nome/valor dos campos de formulário HTML em um formato especial e os enviam ao servidor (normalmente) usando um de dois métodos. O primeiro método, usando o comando HTTP GET, codifica a entrada do usuário na própria URL e solicita o documento correspondente. O servidor reconhece que a primeira parte da URL se refere a um programa e o invoca, passando a informação codificada na outra parte da URL como parâmetro. O segundo método usa o comando HTTP POST para solicitar ao servidor que aceite os dados codificados e os transmita para um aplicativo da web como um fluxo.

Usando o método GET

Você pode usar os recursos da rede rapidamente usando o método GET de codificação de dados em uma URL. Basta criar uma URL apontando para um programa de servidor e usar uma convenção simples para fixar os pares nome/valor codificados que compõem nossos

dados. Por exemplo, o trecho de código a seguir abre uma URL para

um programa CGI antigo chamado login.cgi no servidor myhost e passa dois pares nome/valor. Em seguida, ele imprime qualquer texto que o CGI envia de volta: URL url = novo URL(

// esta string deve ser codificada em URL

"http://myhost/cgi-bin/login.cgi?
Name=Pat&Password=foobar"); BufferedReader bin = novo BufferedReader(

novo InputStreamReader(url.openStream()));

Linha de corda;

while ((linha = bin.readLine()) != nulo) {

System.out.println(linha);

}

Para formar a URL com parâmetros, começamos com a URL base de login.cgi.

Adicionamos um ponto de interrogação (?), que marca o início dos dados do parâmetro, seguido pelo primeiro par “nome=valor”. Podemos adicionar quantos pares quisermos, separados por caracteres de E comercial (&). O resto do nosso código simplesmente abre o fluxo e lê a resposta do servidor. Lembre-se de que criar uma URL não abre realmente a conexão. Neste caso, a conexão URL foi feita implicitamente quando chamamos openStream(). Embora estejamos assumindo aqui que

nosso servidor envia texto de volta, ele pode enviar qualquer coisa, incluindo imagens, áudio ou PDFs.

Pulamos uma etapa aqui. Este exemplo funciona porque nossos pares nome/valor são texto simples. Se algum caractere “não imprimível” ou especial (incluindo ? ou &) estiver nos pares, ele deverá ser codificado primeiro. A classe java.net.URLEncoder fornece um utilitário para codificar os dados. Mostraremos como usá-lo no próximo exemplo em“Usando o método POST”.

Embora este pequeno exemplo envie um campo de senha, você nunca deve enviar dados confidenciais usando esta abordagem simplista. Os dados neste exemplo são enviados em texto não criptografado pela rede (não são criptografados). Mesmo o uso de HTTPS (HTTP seguro) não ocultará o URL. E, nesse caso, o campo de senha também apareceria em qualquer lugar onde o URL fosse impresso, incluindo logs do servidor, histórico do navegador e favoritos.

Usando o método POST

Para quantidades maiores de dados de entrada ou conteúdo confidencial, você provavelmente usará a opção POST. Aqui está um pequeno aplicativo que funciona como um formulário HTML. Ele coleta dados de dois campos de texto – nome e senha

– e envia os dados para o serviço Postman Echo1URL usando o método HTTP POST.

Este aplicativo cliente baseado em Swing funciona como um navegador da web e se conecta a um aplicativo da web.

Aqui está o principal método de rede que executa a solicitação e trata a resposta:

//arquivo: ch13/examples/Post.java

postData void protegido() {

StringBuilder sb = new StringBuilder();

String pw = new String(passwordField.getPassword());

tentar {

sb.append(URLEncoder.encode("Nome", "UTF-8") + "="); sb.append(URLEncoder.encode(nameField.getText(), "UTF-8")); sb.append("&" + URLEncoder.encode("Senha", "UTF-8") + "="); sb.append(URLEncoder.encode(pw, "UTF-8"));

} catch (UnsupportedEncodingException uee) {

System.out.println(uee);

}

String formData = sb.toString();

tentar {

URL url = nova URL(postURL);

URLConexão HttpURL =

(HttpURLConnection) url.openConnection();

urlcon.setRequestMethod("POST");

urlcon.setRequestProperty("Tipo de conteúdo",

"aplicativo/x-www-form-urlencoded");

```
urlcon.setDoOutput(true);

urlcon.setDoInput(true);

PrintWriter beicinho = new PrintWriter(new
OutputStreamWriter(

urlcon.getOutputStream(), "8859_1"), verdadeiro);

pout.print(formData);

beicinho.flush();

// A postagem foi bem-sucedida?

if (urlcon.getResponseCode() ==
HttpURLConnection.HTTP_OK)

System.out.println("Postado ok!");

outro {

System.out.println("Postagem incorreta...");

retornar;

}

// Viva! Vá em frente e leia os resultados

InputStream é =urlcon.getInputStream();

InputStreamReader isr = novo InputStreamReader(é);

BufferedReader br = novo BufferedReader(isr);

Linha de corda;
```

```
while ((linha = br.readLine()) != null) {

System.out.println(linha);

}

close();

} catch (MalformedURLException e) {

System.out.println(e); //postURL incorreto

} catch (IOException e2) {

System.out.println(e2); //erro de E/S

}

}
```

O início da aplicação cria o formulário utilizando elementos Swing, como fizemos

emCapítulo 12. Toda a mágica acontece no método postData() protegido. Primeiro, criamos um StringBuilder e o carregamos com pares nome/valor, separados por e comercial. (Não precisamos do ponto de interrogação inicial quando usamos o método POST porque não estamos anexando ao URL.) Cada par é primeiro codificado usando o método estático URLEncoder.encode(). Também executamos os campos de nome através do codificador, embora eles não contenham caracteres especiais neste exemplo. Esta etapa extra é uma prática recomendada e simplesmente um bom hábito.

Os nomes dos campos nem sempre são tão claros.

A seguir, configuramos a conexão com o servidor. No nosso exemplo anterior, não fomos obrigados a fazer nada de especial para enviar os dados porque a solicitação foi feita pelo simples ato de abrir a URL no servidor. Aqui, temos que carregar parte do peso de conversar com o servidor web remoto. Felizmente, o objeto

HttpURLConnection faz a maior parte do trabalho para nós; só temos que dizer que tipo de dados estamos enviando e como queremos enviá-los. Obtemos um objeto URLConnection por meio do método openConnection(). Sabemos que estamos usando o protocolo HTTP, então devemos ser capazes de convertê-lo em um tipo HttpURLConnection, que tem o suporte que precisamos. Como o HTTP é um dos protocolos garantidos, podemos fazer essa suposição com segurança. (Falando em segurança, usamos HTTP aqui apenas para fins de demonstração. Atualmente, muitos dados são considerados confidenciais. As diretrizes do setor estabeleceram o padrão para HTTPS; mais sobre isso em breve em“SSL e comunicações seguras na Web”.)

Usamos setRequestMethod() para informar à conexão que queremos fazer uma operação POST. Também usamos setRequestProperty() para definir o campo ContentType de nossa solicitação HTTP para o tipo apropriado – neste caso, o tipo de mídia adequado2para dados de formulário codificados. (Isso é necessário para informar ao servidor que tipo de dados estamos enviando, "application/x-www-form-urlencoded"

no nosso caso.)

Para a etapa final de configuração, usamos os métodos setDoOutput() e setDoInput() para informar à conexão que queremos enviar e receber dados de fluxo. A conexão URL infere dessa combinação que vamos fazer uma operação POST e espera uma resposta.

Para enviar dados, obtemos um fluxo de saída da conexão com getOutputStream() e criamos um PrintWriter para que possamos escrever facilmente o conteúdo codificado do nosso formulário. Depois de postarmos os dados, nosso aplicativo chama getResponseCode() para ver se o código de resposta HTTP do servidor indica que o POST foi bem-sucedido. Outros códigos de resposta (definidos como constantes em HttpURLConnection) indicam diversas falhas.

Embora os dados codificados em formulário (conforme indicado pelo tipo de mídia especificado para o campo Content-Type) sejam comuns, outros tipos de comunicação são possíveis. Poderíamos usar os fluxos de entrada e saída para trocar tipos de dados arbitrários com o programa servidor. A operação POST poderia enviar qualquer tipo de dados; o aplicativo do servidor simplesmente precisa saber como lidar com isso.

Uma observação final: se você estiver escrevendo um aplicativo que precisa decodificar dados de formulário, poderá usar java.net.URLDecoder para desfazer a operação do URLEncoder. Certifique-se de especificar UTF-8 ao chamar decode().

A conexão HttpURL

Outras informações da solicitação também estão disponíveis em HttpURLConnection.

Poderíamos usar getContentType() e getContentEncoding() para determinar o tipo MIME e a codificação da resposta. Também poderíamos interrogar os cabeçalhos de resposta HTTP usando getHeaderField(). (Cabeçalhos de resposta HTTP são pares nome/valor de metadados transportados com a resposta.) Os métodos de

conveniência podem buscar campos de cabeçalho formatados como número inteiro e data, getHeaderFieldInt() e getHeaderFieldDate(), que retornam um tipo int e um tipo longo, respectivamente. O comprimento do conteúdo e a data da última modificação são fornecidos por meio de getContentLength() e getLastModified().

SSL e comunicações seguras na Web

Alguns dos exemplos anteriores enviaram dados confidenciais ao servidor. O HTTP

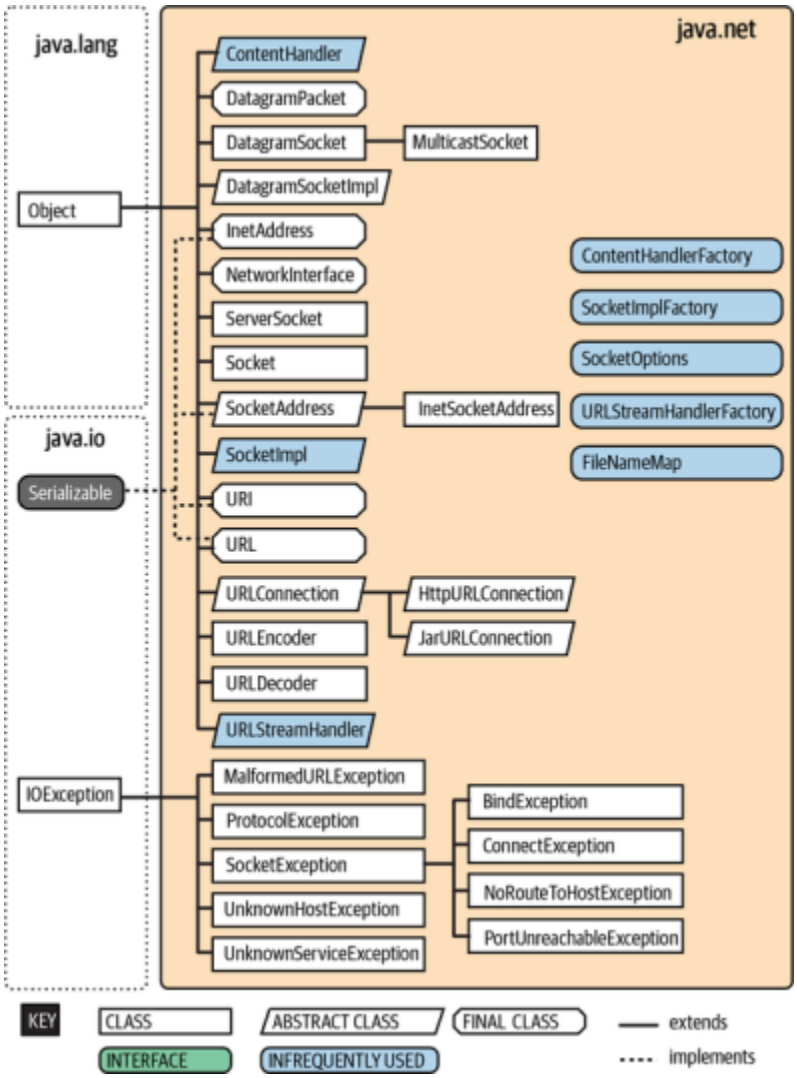
padrão não fornece criptografia para ocultar nossos dados. Felizmente, adicionar segurança para operações GET e POST como essa é fácil (trivial, na verdade, para o desenvolvedor do lado do cliente). Quando disponível, você simplesmente precisa usar uma forma segura do protocolo HTTP – HTTPS. Considere o URL de teste do exemplo Post:

https://postman-echo.com/post

HTTPS é uma versão do protocolo HTTP padrão executado em Secure Sockets Layer (SSL), que usa técnicas de criptografia de chave pública para criptografar as comunicações do navegador para o servidor. A maioria dos navegadores e servidores da web

atualmente vem com suporte integrado para HTTPS (ou soquetes SSL brutos).

Portanto, se o seu servidor web suportar HTTPS e estiver configurado, você poderá usar um navegador para enviar e receber dados seguros simplesmente especificando o protocolo https em suas URLs. Há muito mais para aprender sobre SSL e aspectos relacionados à segurança, como autenticar com quem você está realmente falando, mas no que diz respeito à criptografia básica de dados, isso é tudo que você precisa fazer. Não é algo com o qual seu código precise lidar diretamente. Java é fornecido com suporte a SSL e HTTPS.



Programação de Rede

A web domina as discussões dos desenvolvedores sobre redes, mas há mais por aí do que apenas páginas HTML! À medida que as APIs de rede Java amadureceram, Java também se tornou a linguagem preferida para a implementação de aplicativos e serviços cliente/servidor tradicionais. Nesta seção, veremos o pacote java.net, que contém as classes fundamentais para comunicação e trabalho com recursos de rede.

As classes de java.net se enquadram em duas categorias gerais: a API Sockets, para trabalhar com protocolos de

rede de baixo nível, e APIs de nível superior, orientadas para a Web, que funcionam com URLs, como vimos na seção anterior.Figura 13-

2mostra a maior parte da hierarquia de pacotes java.net.

Figura 13-2. Principais classes e interfaces do pacote java.net

A API Sockets do Java fornece acesso aos protocolos padrão usados para comunicações entre hosts. Os soquetes são o mecanismo subjacente a todos os outros tipos de comunicações portáteis em rede. Os soquetes são a ferramenta de nível mais baixo na caixa de ferramentas de rede geral – você pode usar soquetes para qualquer tipo de comunicação entre cliente e servidor ou aplicativos peer, mas precisa implementar seus próprios protocolos em nível de aplicativo para manipular e interpretar os dados. Ferramentas de rede de nível superior, como invocação de método remoto, HTTP e serviços da Web, são implementadas sobre soquetes.

tomadas

Sockets são uma interface de programação de baixo nível para comunicações em rede.

Eles enviam fluxos de dados entre aplicativos que podem ou não estar no mesmo host.

Os soquetes se originaram no BSD Unix e são, em algumas linguagens de programação, coisas confusas e complicadas, com muitas peças pequenas que podem quebrar e causar estragos. A razão para isso é que a maioria das APIs de soquete podem ser

usadas com quase qualquer tipo de protocolo de rede subjacente. Como os protocolos que transportam dados pela rede podem ter recursos radicalmente diferentes, a interface do soquete pode ser bastante complexa.3

O pacote java.net suporta uma interface de soquete simplificada e orientada a objetos que torna as comunicações de rede consideravelmente mais fáceis. Se você já fez programação de rede usando soquetes em outras linguagens, ficará agradavelmente surpreso com o quão simples as coisas podem ser quando os objetos encapsulam os detalhes sangrentos. Se esta é a primeira vez que você encontra soquetes, você descobrirá que conversar com outro aplicativo pela rede pode ser tão simples quanto ler um arquivo ou obter informações do usuário. A maioria das formas de E/S em Java, incluindo a maioria das E/S de rede, usam as classes de fluxo descritas em“Córregos”.

Os streams fornecem uma interface de E/S unificada para que a leitura ou gravação na Internet seja semelhante à leitura ou gravação no sistema local. Além das interfaces orientadas a fluxo, as APIs de rede Java podem funcionar com a API orientada a buffer Java NIO para aplicativos altamente escaláveis.

Java fornece soquetes para suportar três classes distintas de protocolos subjacentes: Socket, DatagramSocket e MulticastSocket. Nesta seção, veremos a classe Socket básica do Java, que usa um protocolo confiável e orientado à conexão. Um protocolo orientado a conexão fornece o equivalente a uma conversa telefônica. Depois de estabelecer uma conexão, dois aplicativos podem enviar fluxos de dados de um lado para outro, e a conexão permanece ativa mesmo quando ninguém está falando. Por ser confiável, o protocolo também garante

que nenhum dado seja perdido (reenviando os dados, conforme necessário) e que tudo o que você enviar sempre chegará na ordem em que foi enviado.

Teremos que deixar as outras duas classes, que usam um protocolo não confiável e sem conexão, para você explorar por conta própria. (Novamente, veja *Programação de Rede Java* por Elliotte Rusty-Harold para uma discussão detalhada.) Um protocolo sem conexão é como o serviço postal. Os aplicativos podem enviar mensagens curtas entre si, mas nenhuma conexão ponta a ponta é configurada antecipadamente e nenhuma tentativa é feita para manter as mensagens em ordem. Nem é garantido que as mensagens chegarão. Um MulticastSocket é uma variação de um DatagramSocket que realiza multicasting – enviando dados simultaneamente para vários destinatários.

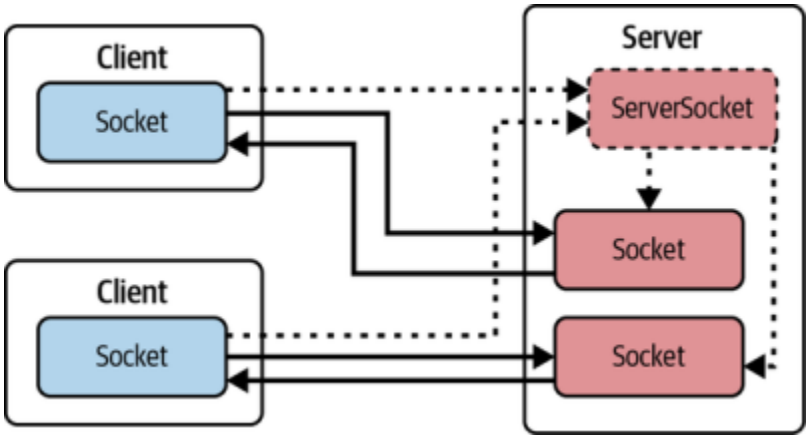
Pense no serviço postal entregando panfletos do mercado local para endereços de

“residentes” em um grande bairro. Trabalhar com soquetes multicast é muito parecido com trabalhar com soquetes de datagrama; há simplesmente mais destinatários.

Em teoria, praticamente qualquer protocolo pode ser usado abaixo da camada de soquete. Na prática, existe apenas uma família de protocolos importante usada na Internet e apenas uma família de protocolos suportada por Java: o Protocolo de Internet (IP). A classe Socket fala TCP, protocolo de controle de transmissão, sobre IP

(geralmente agrupado como TCP/IP); e a classe DatagramSocket sem conexão fala UDP, User Datagram

Protocol, sobre IP.



Clientes e Servidores

Ao escrever aplicações de rede, é comum falar sobre clientes e servidores. A distinção é cada vez mais vaga, mas geralmente o cliente inicia a conversa. O servidor geralmente aceita solicitações recebidas. Há muitas sutilezas nessas funções,4mas para simplificar usaremos esta definição.

Uma diferença importante entre um cliente e um servidor é que um cliente pode criar um soquete para iniciar uma conversa com um aplicativo de servidor a qualquer momento, enquanto um servidor deve estar preparado com antecedência para escutar solicitações de conversação recebidas. A classe java.net.Socket representa um lado de uma conexão de soquete individual no cliente e no servidor. Além disso, o servidor usa a classe java.net.ServerSocket para escutar novas conexões de clientes. Na maioria dos casos, um aplicativo que atua como servidor cria um objeto ServerSocket e espera, bloqueado em uma chamada ao seu método accept(), até que uma solicitação chegue.

Quando um cliente tenta se conectar, o método accept() cria um novo objeto Socket que o servidor usa para se comunicar com o cliente. A instância ServerSocket transfere detalhes do cliente para o novo Socket, conforme mostrado em Figura 13-3.

Figura 13-3. Clientes e servidores usando Socket e ServerSocket

Esse soquete continua a conversa com o cliente, permitindo que o ServerSocket retome sua tarefa de escuta. Desta forma, um servidor pode manter conversas com vários clientes ao mesmo tempo. Ainda existe apenas um ServerSocket, mas o servidor possui vários objetos Socket – um associado a cada cliente.

Clientes

No nível do soquete, um cliente precisa de duas informações para localizar e conectar-se a um servidor na Internet: um nome de host (usado para localizar o endereço de rede do computador host) e um número de porta. O número da porta é um identificador que diferencia vários serviços de rede ou conexões no mesmo host.

Um aplicativo de servidor escuta em uma porta pré-combinada enquanto aguarda conexões. Os clientes fazem uma solicitação para esse número de porta pré-combinado. Se você pensar no computador host como um hotel e nos vários serviços disponíveis para os hóspedes, as portas são como os números dos quartos dos hóspedes. Para se conectar a um serviço, você deve saber o nome do hotel e o número correto do quarto.

Um aplicativo cliente abre uma conexão com um servidor construindo um Socket que especifica esses dois bits de

informação:

```
tentar {

Soquete meia = new Socket("wupost.wustl.edu", 25);

} catch (UnknownHostException e) {

System.out.println("Não foi possível encontrar o host.");

} catch (IOException e) {

System.out.println("Erro ao conectar ao host.");

}
```

Este fragmento de código do lado do cliente tenta conectar um Socket à porta 25 (o serviço de correio SMTP) do host wupost.wustl.edu. O cliente deve lidar com a possibilidade de o nome do host não poder ser resolvido (UnknownHostException) e de o servidor não aceitar uma nova conexão (IOException). Java usa DNS, o serviço de nomes de domínio (DNS) padrão, para resolver o nome do host em um endereço IP

para nós.

Os endereços IP (do protocolo da Internet) são os números de telefone da Internet e o DNS é a lista telefônica global. Cada máquina conectada à internet possui um endereço IP. Se você não conhece esse endereço, procure-o no DNS. Mas se você conhece o endereço de um servidor, o construtor Socket também pode aceitar uma string contendo um endereço IP bruto:

```
Soquete meia = novo Soquete("22.66.89.167", 25);
```

Independentemente de como você inicia, após a conexão do sock, você pode recuperar fluxos de entrada e saída com os métodos getInputStream() e getOutputStream(). O

seguinte código (bastante arbitrário) envia e recebe alguns dados com os streams: tentar {

Servidor de soquete = new Socket("foo.bar.com", 1234);

InputStream in = server.getInputStream();

SaídaStream out = server.getOutputStream();

// escreve um byte

out.write(42);

// escreve uma string delimitada por nova linha ou retorno de carro

PrintWriter pout = new PrintWriter(out, true);

beicinho.println("Olá!");

//lê um byte

byte de volta = (byte)in.read();

// lê uma string delimitada por nova linha ou retorno de carro

Compartimento BufferedReader =

novo BufferedReader(novo InputStreamReader(in));

Resposta de string = bin.readLine();

```
servidor.close();

} catch (IOException e) {

System.err.println(e);

}
```

Nessa troca, o cliente primeiro cria um Socket para comunicação com o servidor. O

construtor Socket especifica o nome do host do servidor (foo.bar.com) e um número de porta pré-combinado (1234). Depois que o cliente se conecta, ele grava um único byte no servidor usando o método write() do OutputStream. Para enviar uma sequência de texto com mais facilidade, ele envolve um PrintWriter em torno do OutputStream. Em seguida, ele executa as operações complementares: lendo um byte do servidor usando o método read() do InputStream e, em seguida, criando um BufferedReader do qual obterá uma string completa de texto. O cliente então encerra a conexão com o método close(). Todas essas operações têm potencial para gerar IOExceptions; nosso snippet lida com essas exceções verificadas envolvendo toda a conversa em um bloco try/catch.

Servidores

Do outro lado da conversa, após o estabelecimento de uma conexão, um aplicativo de servidor utiliza o mesmo tipo de objeto Socket para sua comunicação com um cliente.

Porém, para aceitar uma conexão de um cliente, ele deve primeiro criar um ServerSocket, vinculado à porta

correta. Vamos recriar a conversa anterior do ponto de vista do servidor:

```
// Enquanto isso, em foo.bar.com...

tentar {

Ouvinte ServerSocket = novo ServerSocket(1234);

enquanto (!concluído) {

Cliente de soquete = listener.accept(); //espera pela conexão

InputStream in = client.getInputStream();

OutputStream out = client.getOutputStream();

//lê um byte

byte someByte = (byte)in.read();

// lê uma nova linha ou string delimitada por retorno de carro

Compartimento BufferedReader =

novo BufferedReader(novo InputStreamReader(in) );

String algumaString = bin.readLine();

// escreve um byte

out.write(43);

// diga adeus

PrintWriter pout = new PrintWriter(out, true);
```

```
beicinho.println("Adeus!");

cliente.close();

}

ouvinte.close();

} catch (IOException e) {

System.err.println(e);

}
```

Primeiro, nosso servidor cria um ServerSocket conectado à porta 1234. Na maioria dos sistemas, existem regras sobre quais portas um aplicativo pode usar. Os números de porta são inteiros não assinados de 16 bits, o que significa que podem variar de 0 a 65.535. Os números de porta abaixo de 1.024 são geralmente reservados para processos do sistema e serviços padrão “bem conhecidos”, portanto escolhemos um número de porta fora desse intervalo reservado.5Precisamos criar o ServerSocket apenas uma vez; depois disso, ele pode aceitar quantas conexões chegarem.

Em seguida, entramos em um loop, aguardando que o método accept() do

ServerSocket retorne uma conexão Socket ativa de um cliente. Quando uma conexão for estabelecida, executamos o lado do servidor da nossa caixa de diálogo, fechamos a conexão e retornamos ao topo do loop para aguardar outra conexão. Finalmente, quando o aplicativo do servidor deseja parar de escutar conexões, ele chama o método close() do ServerSocket.

Este servidor é de thread único; ele lida com uma conexão por vez, finalizando uma conversa completa com um cliente antes de retornar ao topo do loop e chamar accept() para escutar outra conexão. Um servidor mais realista teria um loop que aceita conexões simultaneamente e as repassa para seus próprios threads para processamento. Mesmo que não planejemos criar um MMORPG,6mostramos como conduzir uma conversa usando a abordagem thread por cliente em“Um jogo

distribuído”. Se quiser fazer uma pequena leitura independente, você também pode procurar o ServerSocketChannel equivalente ao NIO e sem bloqueio.

O cliente DateAtHost

No passado, muitos computadores em rede executavam um serviço de horário simples que distribuía a hora local do relógio em uma porta bem conhecida. O protocolo de tempo é um precursor do NTP, o Network Time Protocol mais geral. Manteremos o protocolo de tempo por sua simplicidade, mas se você quiser sincronizar os relógios do sistema em rede, o NTP é a melhor opção.7

O próximo exemplo, DateAtHost, inclui uma subclasse de java.util.Date que busca a hora de um host remoto em vez de se inicializar no relógio local. (VerCapítulo 8para uma discussão sobre a classe Date, que ainda é boa para alguns usos, mas foi amplamente substituída por suas primas mais novas e flexíveis, LocalDate e LocalTime.)

DataNoHostconecta-se ao serviço de horário (porta 37) e lê quatro bytes que representam o horário no host remoto. Esses quatro bytes possuem uma especificação

peculiar que decodificamos para obter a hora. Aqui está o código:

```
//arquivo: ch13.examples.DateAtHost.java

pacote ch13.examples;

importar java.net.Socket;

importar java.io.*;

classe pública DateAtHost estende java.util.Date {

static int timePort = 37;

// segundos do início do século 20 até 1º de janeiro de 1970 00:00 GMT

deslocamento longo final estático = 2208988800L;

public DateAtHost(String host) lança IOException {

this(host, timePort);

}

public DateAtHost(String host, porta int) lança IOException {

Servidor de soquete = novo soquete (host, porta);

DataInputStream din =

novo DataInputStream(server.getInputStream());

tempo interno = din.readInt();

servidor.close();
```

```
setTime((((1L << 32) + hora) - deslocamento) * 1000);

}

}
```

Isso é tudo que há para fazer. Não é muito longo, mesmo com alguns babados.

Fornecemos dois construtores possíveis para DateAtHost. Normalmente esperaríamos usar o primeiro, que simplesmente usa o nome do host remoto como argumento. O

segundo construtor especifica o nome do host e o número da porta do serviço de horário remoto. (Se o serviço de horário estivesse sendo executado em uma porta fora do padrão, usaríamos o segundo construtor para especificar o número da porta alternativa.) Esse segundo construtor faz o trabalho de estabelecer a conexão e definir o horário. O primeiro construtor simplesmente invoca o segundo (usando a construção this()) com a porta padrão como argumento. Fornecer construtores simplificados que invocam seus irmãos com argumentos padrão é um padrão comum e útil em Java; essa é a principal razão pela qual mostramos isso aqui.

O segundo construtor abre um soquete para a porta especificada no host remoto. Ele cria um DataInputStream para agrupar o fluxo de entrada e então lê um número inteiro de quatro bytes usando o método readInt(). Não é por acaso que os bytes estão na ordem certa. As classes DataInputStream e DataOutputStream do Java funcionam com bytes de tipos inteiros na ordem de bytes da rede (do mais significativo para o menos significativo). O protocolo de tempo (e outros protocolos de rede padrão que lidam com dados binários) também

usa a ordem de bytes da rede, portanto não precisamos chamar nenhuma rotina de conversão. Provavelmente seriam necessárias conversões explícitas de dados se estivéssemos usando um protocolo fora do padrão, especialmente ao conversar com um cliente ou servidor não-Java. Nesse caso,

teríamos que ler byte por byte e fazer algumas reorganizações para obter nosso valor de quatro bytes. Após a leitura dos dados, finalizamos o soquete, então o fechamos, encerrando a conexão com o servidor. Finalmente, o construtor inicializa o resto do objeto chamando o método setTime() de Date com o valor de tempo calculado.

Os quatro bytes do valor do tempo são interpretados como um número inteiro que representa o número de segundos desde o início do século XX. DateAtHost converte isso para a noção de tempo absoluto do Java – a contagem de milissegundos desde 1º

de janeiro de 1970 (uma data arbitrária padronizada por C e Unix). A conversão primeiro cria um valor longo, que é o equivalente sem sinal do tempo inteiro. Ele subtrai um deslocamento para tornar o tempo relativo à época (1º de janeiro de 1970) em vez do século, e multiplica por 1.000 para converter em milissegundos. O tempo convertido é usado para inicializar o objeto.

A classe DateAtHost pode trabalhar com uma hora recuperada de um host remoto quase tão facilmente quanto Date é usada com a hora no host local. A única sobrecarga adicional é lidar com a possível IOException que o construtor DateAtHost pode lançar: tentar {

Data d = new DateAtHost("time.nist.gov");

```
System.out.println("A hora ali é: " + d);

}

pegar (IOException e) {

System.err.println("Falha ao obter a hora: " + e);

}
```

Este exemplo busca a hora no host time.nist.gov e imprime seu valor.

Um jogo distribuído

Podemos usar nossas novas habilidades de networking para estender nosso jogo de lançamento de maçãs e passar para dois jogadores. Teremos que manter esta incursão simples, mas você pode se surpreender com a rapidez com que podemos lançar uma prova de conceito. Embora existam vários mecanismos que dois jogadores poderiam usar para se conectarem e compartilharem uma experiência, nosso exemplo usa o modelo cliente/servidor básico que discutimos neste capítulo. Um usuário iniciará o servidor e o segundo usuário poderá entrar em contato com esse servidor como cliente. Assim que os dois jogadores estiverem conectados, eles correrão para ver quem consegue limpar as árvores e sebes mais rápido!

Configurando a IU

Vamos começar adicionando um menu ao nosso jogo. Lembrar de“Menus”que os menus ficam em uma barra de menus e funcionam com objetos ActionEvent, como botões. Precisamos de uma opção para iniciar um servidor e outra para entrar em um jogo em um servidor

que alguém já iniciou. O código principal desses itens de menu é simples; podemos usar outro método auxiliar na classe AppleToss:

```
private void setupNetworkMenu() {

JMenu netMenu = new JMenu("Multijogador");

multiplayerHelper = novo Multijogador();

JMenuItem startItem = new JMenuItem("Iniciar Servidor");

startItem.addActionListener(

e -> multiplayerHelper.startServer());

netMenu.add(startItem);

JMenuItem joinItem = new JMenuItem("Entrar no jogo...");

joinItem.addActionListener(e -> {

String outroServidor = JOptionPane.showInputDialog(

AppleToss.this, "Digite o nome ou endereço do servidor:");

multiplayerHelper.joinGame(outroServidor);

});

netMenu.add(joinItem);

JMenuItem quitItem = new JMenuItem("Desconectar");

quitItem.addActionListener(

e -> multiplayerHelper.disconnect());
```

```
netMenu.add(quitItem);

//construa um JMenuBar para a aplicação

JMenuBar mainBar = new JMenuBar();

mainBar.add(netMenu);

setJMenuBar(mainBar);

}
```

O uso de lambdas para ActionListener de cada menu deve parecer familiar. Também usamos o JOptionPane discutido em "Modais e Pop-Ups" para pedir ao segundo jogador o nome ou endereço IP do servidor onde o primeiro jogador está esperando. A lógica de rede é tratada em uma classe separada.

Veremos a classe Multiplayer com mais detalhes nas próximas seções, mas você poderá ver os métodos que implementaremos. O código para esta versão do jogo (na pasta ch13/examples/game) contém o método setupNetworkMenu(), mas os ouvintes lambda apenas abrem uma caixa de diálogo de informações para indicar qual item de menu foi selecionado. Você pode construir a classe Multiplayer e chamar os métodos multiplayer reais nos exercícios no final do capítulo. Mas fique à vontade para conferir o jogo completo – incluindo as partes de rede – na pasta ch13/solutions/game.

O servidor do jogo

Como fizemos em "Servidores", precisamos escolher uma porta e configurar um soquete que esteja escutando uma conexão de entrada. Usaremos a porta 8677 –

“TOSS” em um teclado numérico de telefone. Podemos criar uma classe interna Server em nossa classe Multiplayer para conduzir um thread pronto para comunicações de rede. As variáveis de leitor e gravador serão usadas para enviar e receber os dados reais do jogo. Mais sobre isso em“O protocolo do jogo”:

```
class Server implementa Runnable {

Ouvinte ServerSocket;

execução nula pública() {

Soquete soquete = nulo;

tentar {

ouvinte = novo ServerSocket(gamePort);

while (continue ouvindo) {

soquete = ouvinte.accept(); //espera pela conexão

InputStream in = socket.getInputStream();

Leitor BufferedReader =

novo BufferedReader(novo InputStreamReader(in));

SaídaStream out = socket.getOutputStream();

Gravador PrintWriter = new PrintWriter(out, true);

// ... a lógica do protocolo do jogo começa aqui

}

} catch (IOException ioe) {
```

```
System.err.println(ioe);

}

}

}
```

Configuramos nosso ServerSocket e aguardamos um novo cliente dentro de um loop.

Embora planejemos jogar apenas com um oponente por vez, isso nos permite aceitar clientes subsequentes sem passar por toda a configuração da rede novamente.

Para realmente iniciar o servidor ouvindo pela primeira vez, precisamos apenas de um novo thread que use nossa classe Server:

```
//do multijogador

Servidor servidor;

// ...

public void startServer() {

keepListening = verdadeiro;

// ... outro estado do jogo pode ir aqui

servidor = novo Servidor();

serverThread = new Thread(servidor);

serverThread.start();

}
```

Mantemos uma referência à instância do Servidor em nossa classe Multiplayer para que tenhamos pronto acesso para desligar as conexões caso o usuário selecione a opção “desconectar” no menu, assim:

```
//do multijogador

public void desconectar() {

desconectando = verdadeiro;

keepListening = falso;

// Estamos no meio de um jogo e verificando regularmente essas bandeiras?

// Caso contrário, basta fechar o soquete do servidor para interromper o bloqueio

//aceitar() método.

if (servidor! = nulo && keepPlaying == falso) {

servidor.stopListening();

}

// ... limpa outro estado do jogo aqui

}
```

Usamos principalmente o sinalizador keepPlaying quando estamos dentro do loop do jogo, mas também é útil acima. Se tivermos uma referência de servidor válida, mas não estivermos jogando no momento (keepPlaying é falso), sabemos que devemos desligar o soquete do ouvinte.

O método stopListening() na classe interna Server é simples:

public void stopListening() {

if (ouvinte! = null &&! ouvinte.isClosed()) {

tentar {

ouvinte.close();

} catch (IOException ioe) {

System.err.println("Erro ao desconectar o ouvinte: " +

ioe.getMessage());

}

}

}

Fazemos uma verificação rápida em nosso servidor e tentamos fechar o listener somente se ele existir e ainda estiver aberto.

O cliente do jogo

A configuração e desmontagem do lado do cliente são semelhantes – sem o ServerSocket de escuta, é claro. Espelharemos a classe interna Server com uma classe interna Client e construiremos um método run() inteligente para implementar nossa lógica de cliente:

classe Cliente implementa Executável {

String gameHost;

booleano startNewGame;

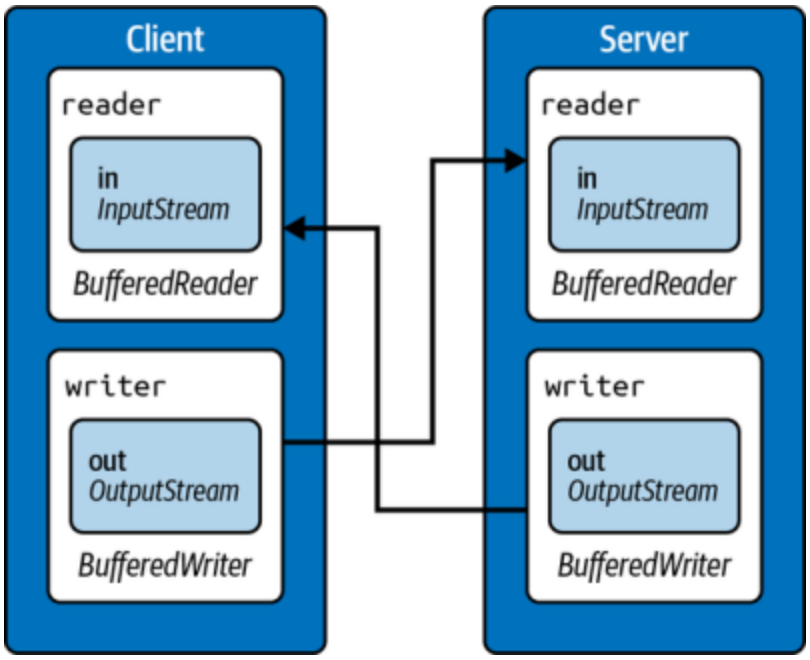
Cliente público(String host) {

gameHost=host;

continue jogando = falso;

iniciarNovoJogo = falso;

}



execução nula pública() {

tente (Socket socket = new Socket(gameHost, gamePort)) {

InputStream in = socket.getInputStream();

Leitor BufferedReader =

novo BufferedReader(novo InputStreamReader(in));

```
SaídaStream out = socket.getOutputStream();

Gravador PrintWriter = new PrintWriter(out, true);

// ... a lógica do protocolo do jogo começa aqui

} catch (IOException ioe) {

System.err.println(ioe);

}

}

}
```

Passamos o nome do servidor para o construtor Client e contamos com a variável gamePort comum usada pelo Server para configurar o soquete. Usamos a técnica

“tentar com recurso” discutida em“tente com recursos”para criar nosso soquete e garantir que ele seja limpo quando terminarmos. Dentro desse bloco try do recurso, criamos nossas instâncias de leitor e gravador para a metade da conversa do cliente, conforme mostrado emFigura 13-4.

Figura 13-4. Conexões de cliente e servidor de jogo

Para fazer isso, adicionaremos outro método auxiliar à nossa classe auxiliar Multiplayer:

```
//do multijogador

public void joinGame(String outroServidor) {

clientThread = new Thread(novo Cliente(outroServidor));
```

```
clienteThread.start();

}
```

Não precisamos de um método desconectado() separado — podemos usar as mesmas variáveis de estado usadas pelo servidor. Para o cliente, a referência do servidor será nula, portanto não tentaremos desligar um ouvinte inexistente.

O protocolo do jogo

Você provavelmente notou que deixamos de fora a maior parte do método run() para as classes Server e Client. Depois de construirmos e conectarmos nossos fluxos de dados, o trabalho restante envolve o envio e recebimento colaborativo de informações sobre o estado do nosso jogo. Esta comunicação estruturada é o protocolo do jogo.

Todo serviço de rede possui um protocolo. Pense no “P” em HTTP. Até mesmo nosso exemplo DateAtHost usa um protocolo (muito simples) para que clientes e servidores saibam quem deve falar e quem deve ouvir a qualquer momento. Se ambos os lados tentarem conversar ao mesmo tempo, a informação provavelmente será perdida. Se ambos os lados acabarem esperando que o outro lado diga algo (por exemplo, tanto o servidor quanto o cliente estão bloqueando uma chamada reader.readLine()), então a conexão parecerá travada.

Gerenciar essas expectativas de comunicação é o núcleo de qualquer protocolo, mas o que dizer e como responder também são importantes. Esta parte de um protocolo geralmente exige mais esforço do desenvolvedor. Parte da dificuldade é que você

realmente precisa de ambos os lados para testar seu trabalho à medida que avança.

Você não pode testar um servidor sem cliente e vice-versa. Construir os dois lados à medida que avança pode parecer entediante, mas vale a pena o esforço extra. Tal como acontece com outros tipos de depuração, corrigir uma pequena alteração incremental é muito mais simples do que descobrir o que pode estar errado em um grande bloco de código.

Em nosso jogo, faremos com que o servidor conduza a conversa. Essa escolha é arbitrária – poderíamos ter usado o cliente ou poderíamos ter construído uma base mais sofisticada e permitido que tanto o cliente quanto o servidor ficassem encarregados de certas coisas simultaneamente. Porém, com a decisão do “servidor responsável” tomada, podemos tentar um primeiro passo muito simples em nosso protocolo. Faremos com que o servidor envie um comando “NEW_GAME” e então espere que o cliente responda com uma resposta “OK”. O código do lado do servidor (após o estabelecimento da conexão com o cliente) fica assim:

```
//Cria um novo jogo com o cliente

escritor.println("NEW_GAME");

// Se o cliente concordar, envia a localização das árvores

Resposta de string = leitor.readLine();

if (resposta! = null && resposta.equals("OK")) {

System.out.println("Iniciando um novo jogo!")

// ... escreva os dados do jogo aqui
```

} outro {

System.err.println("Resposta de início inesperada: " + resposta); System.err.println("Pulando o jogo e esperando novamente."); continue jogando = falso;

}

Se obtivermos a resposta "OK" esperada, podemos prosseguir com a criação de um novo jogo e compartilhar a localização das árvores e sebes com nosso oponente – mais sobre isso em um minuto. (Se não obtivermos um "OK", mostramos um erro e redefinimos para aguardar alguma outra tentativa.) O código do lado do cliente correspondente para esta primeira etapa flui de forma semelhante:

// Esperamos ver o comando NEW_GAME primeiro

Resposta de string = leitor.readLine();

// Se não virmos esse comando, desconectamos e retornamos

if (resposta == nulo || !response.equals("NEW_GAME")) {

System.err.println("Comando inicial inesperado: " + resposta); System.err.println("Desconectando");

escritor.println("DESCONECTAR");

retornar;

}

// Yay! Nós vamos jogar um jogo. Envie uma confirmação

escritor.println("OK");

Se quiser experimentar as coisas como estão, você pode iniciar seu servidor a partir de um sistema e depois entrar no jogo a partir de um segundo sistema. (Você também pode simplesmente iniciar uma segunda cópia do jogo a partir de uma janela de terminal separada. Nesse caso, o nome do “outro host” seria a palavra-chave de rede localhost.) Quase imediatamente após ingressar na segunda instância do jogo, você deve consulte a seção “Iniciando um novo jogo!” confirmação impressa no terminal do primeiro jogo. Parabéns! Você está no caminho certo para projetar um protocolo de jogo. Vamos continuar.

Precisamos equilibrar o campo de jogo – literalmente. O servidor instruirá seu jogo a construir um novo campo e então poderá enviar as coordenadas de todos os novos obstáculos ao cliente. O cliente, por sua vez, pode aceitar todas as árvores e sebes que chegam e colocá-las em campo limpo. Depois que o servidor enviar todas as árvores, ele poderá enviar um comando "START" e o jogo poderá começar. Continuaremos usando strings para comunicar nossas mensagens. Esta é uma maneira de passar os detalhes da nossa árvore para o cliente:

```
gameField.setupNewGame();

for (Árvore: gameField.trees) {

escritor.println("ÁRVORE " + tree.getPositionX() + " " +

tree.getPositionY());

}

// faça o mesmo para hedges ou quaisquer outros elementos compartilhados.
```

..

//Tente iniciar o jogo, mas certifique-se de que o cliente esteja pronto escritor.println("INICIAR");

resposta = leitor.readLine();

keepPlaying = resposta.equals("OK");

No lado do cliente, podemos chamar readLine() em um loop para linhas "TREE" até vermos a linha “START”, assim (com um pouco de tratamento de erros):

// E agora junte as árvores e monte nosso campo

gameField.trees.clear();

resposta = leitor.readLine();

while (response.startsWith("TREE")) {

String[] partes = resposta.split(" ");

int x = Integer.parseInt(partes[1]);

int y = Integer.parseInt(partes[2]);

Árvore árvore = nova Árvore();

árvore.setPosition(x, y);

gameField.trees.add(árvore);

resposta = leitor.readLine();

}

// Faça o mesmo para hedges ou outros elementos compartilhados

// Após o envio de todas as listas de obstáculos, o servidor emitirá

// um comando INICIAR. Certifique-se de que entendemos isso antes de joga r

if (!response.equals("INICIAR")) {

// Hmm, deveríamos ter encerrado a lista de obstáculos com um START,

// mas não o fez. Resgate.

System.err.println("Início inesperado do jogo: " + resposta); System.err.println("Desconectando");

escritor.println("DESCONECTAR");

retornar;

} outro {

// Oba de novo! Estamos começando um jogo. Reconheça este comando

escritor.println("OK");

continue jogando = verdadeiro;

gameField.repaint();

}

Neste ponto, ambos os jogos devem ter os mesmos obstáculos e os jogadores podem começar a superá-los.

O servidor entrará em um loop de votação e enviará a pontuação atual duas vezes por segundo. O cliente responderá com sua pontuação atual. Observe que certamente existem outras opções de como compartilhar alterações na pontuação. Embora a votação seja simples, jogos mais avançados ou que exigem feedback mais imediato sobre jogadores remotos provavelmente usarão opções de comunicação mais diretas. Por enquanto, queremos nos concentrar principalmente em uma boa rede de ida e volta, para que a pesquisa mantenha nosso código mais simples.

O servidor deve continuar enviando a pontuação atual até que o jogador local limpe tudo ou vejamos uma resposta de final de jogo do cliente. Precisaremos analisar a resposta do cliente para atualizar a pontuação do outro jogador e observar sua solicitação para encerrar o jogo. Também temos que estar preparados para que o cliente simplesmente se desconecte. Esse loop se parece com isto:

```
while (continue jogando) {

tentar {

if (gameField.trees.size() > 0) {

escritor.print("PONTUAÇÃO ");

} outro {

escritor.print("END ");

continue jogando = falso;

}
```

```
escritor.println(gameField.getScore(1));
resposta = leitor.readLine();
if (resposta == nulo) {
continue jogando = falso;
desconectando = verdadeiro;
} outro {
String partes[] = resposta.split(" ");
mudar (partes[0]) {
caso "FIM":
continue jogando = falso;
caso "PONTUAÇÃO":
gameField.setScore(2, partes[1]);
quebrar;
caso "DESCONECTAR":
desconectando = verdadeiro;
continue jogando = falso;
quebrar;
padrão:
System.err.println("Aviso. Comando inesperado: " +
partes[0] + ". Ignorando.");
```

```
}

}

Thread.sleep(500);

} catch(InterruptedException e) {

System.err.println("Interrompido durante a pesquisa.
Ignorando.");

}

}
```

O cliente irá espelhar essas ações. Felizmente para o cliente, ele está apenas reagindo aos comandos vindos do servidor. Não precisamos de um mecanismo de votação separado aqui. Bloqueamos a espera para ler uma linha, analisamos e então construímos nossa resposta:

```
while (continue jogando) {

resposta = leitor.readLine();

String[] partes = resposta.split(" ");

mudar (partes[0]) {

caso "FIM":

continue jogando = falso;

caso "PONTUAÇÃO":

gameField.setScore(2, partes[1]);
```

```
quebrar;

caso "DESCONECTAR":

desconectando = verdadeiro;

continue jogando = falso;

quebrar;

padrão:

System.err.println("Comando de jogo inesperado: " +

resposta + ". Ignorando.");

}

if (desconectando) {

// Estamos nos desconectando ou eles estão. Reconheça
e desista.

escritor.println("DESCONECTAR");

retornar;

} outro {

// Se não estivermos desconectando, responda com
nossa pontuação atual if (gameField.trees.size() > 0) {

escritor.print("PONTUAÇÃO ");

} outro {

continue jogando = falso;

escritor.print("END ");
```

```
}

escritor.println(gameField.getScore(1));

}

}
```

Quando um jogador limpa todas as suas árvores e sebes, ele envia (ou responde com) um comando "END" que inclui sua pontuação final. Nesse ponto, perguntamos se os mesmos dois jogadores querem jogar novamente. Nesse caso, podemos continuar usando as mesmas instâncias de leitor e gravador tanto para o servidor quanto para o cliente. Caso contrário, deixaremos o cliente se desconectar e o servidor voltará a escutar a entrada de outro jogador:

```
// Se não estivermos desconectando, pergunte sobre jogar novamente

if (!desconectando) {

String mensagem = gameField.getWinner() +

"Você gostaria de pedir para eles jogarem de novo?";

int meuPlayAgain = JOptionPane.showConfirmDialog(gameField,

mensagem, "Jogar novamente?", JOptionPane.YES_NO_OPTION);

if (myPlayAgain == JOptionPane.YES_OPTION) {

// Se eles não desconectaram, peça para jogar novamente
```

```
escritor.println("PLAY_AGAIN");
String playAgain = leitor.readLine();
if (jogar novamente! = nulo) {
mudar (reproduzir novamente) {
caso "SIM":
iniciarNovoJogo = verdadeiro;
quebrar;
caso "DESCONECTAR":
continue jogando = falso;
iniciarNovoJogo = falso;
desconectando = verdadeiro;
quebrar;
padrão:
System.err.println("Aviso. Resposta inesperada: "
+ playAgain + ". Não estou jogando de novo.");
}
}
}
}
```

E um último trecho de código recíproco para o cliente:

```
if (!desconectando) {

// Verifica se eles querem jogar novamente

resposta = leitor.readLine();

if (resposta! = null && resposta.equals("PLAY_AGAIN")) {

// Queremos jogar de novo?

String mensagem = gameField.getWinner() +

" Gostaria de jogar novamente?";

int meuPlayAgain =
JOptionPane.showConfirmDialog(gameField,

mensagem, "Jogar novamente?",
JOptionPane.YES_NO_OPTION);

if (myPlayAgain == JOptionPane.YES_OPTION) {

escritor.println("SIM");

iniciarNovoJogo = verdadeiro;

} outro {

// Não estou jogando novamente então desconecte.

desconectando = verdadeiro;

escritor.println("DESCONECTAR");

}

}

}
```

Tabela 13-1resume nosso protocolo simples.

Tabela 13-1. Protocolo de jogo de lançamento da Apple

Comando do

Argumentos

Argumentos

servidor

(opcional)

Resposta do cliente (opcional)

NOVO JOGO

OK

ÁRVORE

xy

COMEÇAR

OK

PONTUAÇÃO

pontuação

SCOREENDDISCON pontuação

NECT

FIM

pontuação

PONTUAÇÃO

pontuação

DESCONECTAR

Comando do

Argumentos

Argumentos

servidor

(opcional)

Resposta do cliente (opcional)

JOGAR DE NOVO

SIM DESLIGAR

DESCONECTAR

Muito mais para explorar

Poderíamos passar muito mais tempo em nosso jogo. Poderíamos expandir o protocolo para permitir múltiplos oponentes. Poderíamos mudar o objetivo para eliminar os obstáculos e destruir seu oponente. Poderíamos tornar o protocolo mais bidirecional, permitindo ao cliente iniciar algumas das atualizações. Poderíamos usar protocolos alternativos de nível inferior suportados por Java, como UDP em vez de TCP. Na verdade, existem livros inteiros dedicados a jogos, programação em rede e programação de jogos em rede!

Mas ufa! Você conseguiu! Dizer que cobrimos muito terreno é um eufemismo.

Esperamos que você tenha um conhecimento sólido da sintaxe e das classes principais do Java. Você pode usar esse entendimento à medida que avança e aprende outros detalhes interessantes e técnicas avançadas. Escolha uma área que lhe interesse e vá um pouco mais fundo. Se você ainda estiver curioso sobre Java em geral, tente conectar partes deste livro. Por exemplo, você pode tentar usar expressões regulares para analisar nosso protocolo de jogo de lançamento de maçã. Ou você pode criar um protocolo mais sofisticado e passar pequenos pedaços de dados binários pela rede, em vez de simples strings. Para praticar a escrita de programas mais complexos, você pode reescrever algumas das classes internas e anônimas do jogo para serem classes separadas e independentes, ou até mesmo substituí-las por expressões lambda.

Se você quiser explorar outras bibliotecas e pacotes Java enquanto segue alguns dos exemplos nos quais já trabalhou, você pode se aprofundar na API Java2D e criar maçãs e árvores mais bonitas. Você poderia experimentar alguns dos outros objetos de coleção, como TreeMap ou Deque. Você poderia pesquisar o popularFormato JSONe

tente reescrever o código de comunicação multijogador. Usar JSON para o protocolo provavelmente também lhe dará a oportunidade de trabalhar com uma biblioteca.

Quando estiver pronto para diversificar ainda mais, você poderá ver como o Java funciona fora do desktop, tentando algum desenvolvimento Android. Ou veja grandes ambientes de rede e o Jakarta Enterprise Edition

da Eclipse Foundation. Talvez big data esteja no seu radar? A Fundação Apache possui vários projetos, como Hadoop ou Spark. Java tem seus detratores, mas continua sendo uma parte vital e vibrante do mundo do desenvolvedor profissional.

Agora que definimos alguns caminhos para pesquisas futuras, estamos prontos para encerrar a parte principal do nosso livro. OGlossárioconté m uma referência rápida de muitos termos e tópicos úteis que abordamos.Apêndice Atem alguns detalhes sobre como importar exemplos de código para o IntelliJ IDEA.Apêndice Binclui respostas

para todas as perguntas de revisão, bem como algumas dicas e orientações sobre os exercícios de código.

Esperamos que você tenha gostado desta sexta edição do Learning Java. Esta é realmente a oitava edição da série que começou há mais de duas décadas com Explorando Java. Foi uma viagem longa e incrível observar o desenvolvimento do Java naquela época, e agradecemos a todos vocês que nos acompanharam ao longo dos anos. Como sempre, agradecemos seus comentários para nos ajudar a continuar melhorando este livro no futuro. Pronto para mais uma década de Java? Nós somos!

Perguntas de revisão

1. Quais protocolos de rede a classe URL suporta por padrão?

2. Você pode usar Java para baixar dados binários de uma fonte online?

3. Quais são as etapas de alto nível para enviar dados de formulário para um servidor web usando Java? Quais classes estão envolvidas?

4. Que classe você usa para escutar conexões de rede de entrada?

5. Ao criar seu próprio servidor, como fez para o jogo, há alguma regra para escolher um número de porta?

6. Um aplicativo de servidor escrito em Java pode suportar vários clientes simultâneos?

7. A quantos servidores simultâneos uma determinada instância de cliente Socket pode se conectar?

Exercícios de código

1. Crie seu próprio cliente DateAtHost amigável (FDClient, Friendly Date Client, em nossas soluções) e servidor (FDServer). Use as classes e formatadores

de“Datas e horários”para produzir um servidor que envie uma linha de texto bem formatado contendo a data e hora atuais. Seu cliente deve ler essa linha após conectar-se e imprimi-la. (Seu cliente não precisa estender o Instant ou mesmo armazenar a resposta além de imprimi-la.)

2. Nosso protocolo de jogo ainda não inclui suporte para obstáculos de hedge. (Os hedges ainda estão em jogo, mas ainda não fazem parte das comunicações da rede.) RevisãoTabela 13-1e adicionar suporte para uma entrada HEDGE

semelhante às nossas linhas TREE. Pode ser mais fácil atualizar primeiro o lado do cliente, embora você precise

atualizar ambos os lados para que as sebes funcionem como as árvores para ambos os jogadores.

Exercícios Avançados

1. Atualize sua classe FDServer para lidar com vários clientes simultâneos com threads ou threads virtuais. Você pode colocar o código de manipulação do cliente em um lambda, uma classe interna (anônima) ou uma classe auxiliar

separada. Você poderá usar a mesma classe FDClient do primeiro exercício sem recompilar. Se você usar threads virtuais, lembre-se de que eles ainda podem ser um recurso de visualização na sua versão do Java. Use os sinalizadores apropriados ao compilar e executar. (Nossa solução para este exercício está no FDServer2.)

2. Este exercício é mais um incentivo para explorar o mundo dos serviços da web, agora que você viu alguns exemplos de uso de Java para interagir com APIs online. Pesquise online um serviço com uma opção de conta de desenvolvedor gratuita (ainda pode ser necessária a inscrição) e escreva um cliente Java para esse serviço. Muitos dos serviços disponíveis online exigem algum tipo de autenticação, como um token de API ou uma chave de API (geralmente strings longas que funcionam como nomes de usuário exclusivos). Sites

comoaleatório.orgouopenweathermap.orgpodem ser lugares divertidos para começar. (Nós fornecemos um cliente completo, NetworkInt, para obter

números inteiros aleatórios de random.org nas soluções. Você precisará fornecer sua própria chave de API na fonte para que o cliente funcione.) 1Postman é uma ferramenta fantástica para desenvolvedores web. Você

pode aprender mais noSite do carteiro. O serviço de teste que eles hospedampostman-

echo.comaceita solicitações GET e POST e retorna a solicitação e qualquer formulário ou parâmetro de URL.

2Você já deve ter ouvido a frase “tipo MIME” antes. O MIME tem suas raízes no e-mail, e o termo “mídia” pretende ser mais genérico.

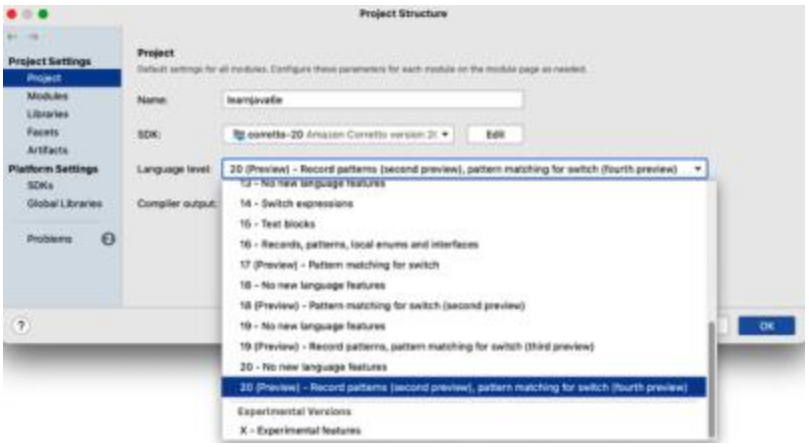
3Para uma discussão detalhada desses soquetes de baixo nível, consulte Unix Network Programming de W. Richard Stevens et al. (Prentice-Hall).

4Um ambiente ponto a ponto, por exemplo, possui máquinas que desempenham simultaneamente as duas funções.

5Para obter mais informações sobre serviços conhecidos e seus números de porta padrão, consulte olista oficialhospedado pela Autoridade para Atribuição de Números da Internet.

6Jogo de RPG on-line massivamente multijogador, se, como os autores, você for velho demais para reconhecer a sigla. Suspirar.

7Na verdade, o site publicamente disponível que usamos do NIST incentiva fortemente os usuários a atualizarem. Veja onotas introdutóriaspara os servidores de horário da Internet do NIST para obter mais informações.



Apêndice A. Exemplos de código e IntelliJ IDEA

Este apêndice o ajudará a começar a usar os exemplos de código encontrados ao longo do livro. Algumas das etapas aqui foram mencionadas emCapítulo 2, mas queremos analisá-los um pouco mais devagar aqui com detalhes específicos sobre como usar os exemplos de livros dentro da Community Edition gratuita do IntelliJ IDEA da JetBrains. Como observado em“Instalando o IntelliJ IDEA e criando um projeto”, você pode obter o IntelliJ nojetbrains. comsite. E eles têm um excelenteGuia de instalaçãose

precisar de mais ajuda para configurar as coisas.

Depois de instalar o IDEA, você vai querer ter certeza de ter uma versão recente do JDK selecionada. (Se você ainda precisar instalar o próprio Java,“Instalando o JDK”tem detalhes para cada uma das principais plataformas.) A caixa de diálogo Arquivo →

Estrutura do Projeto mostrada emFigura A-1permite que você selecione qualquer JDK

que você tenha instalado. Para os propósitos deste livro, você precisará de pelo menos Java 19. Você também

desejará definir a opção “Nível de idioma” para a versão de visualização do JDK escolhido.

Figura A-1. Habilitando os recursos de visualização do Java no IntelliJ IDEA Neste exemplo, usamos Corretto 20 e selecionamos o nível de idioma 20

(Visualização). Se você quiser mais informações sobre como habilitar os recursos de visualização do Java no IntelliJ IDEA, confira seututorial do recurso de visualizaçãoon-

line.

Também queremos reiterar que o IntelliJ IDEA não é o único ambiente de desenvolvimento integrado compatível com Java que existe. Não é nem o único grátis!

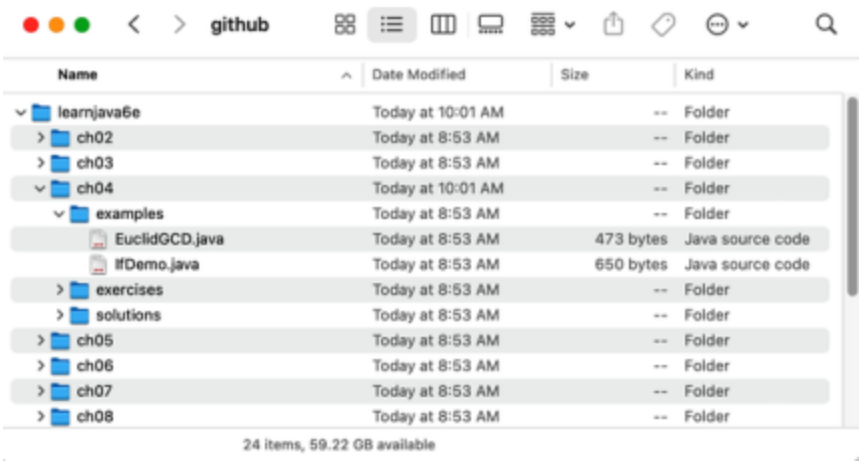
da MicrosoftCódigo VSpode ser rapidamente configurado para suportar Java. EEclipse,

mantido pela IBM, permanece disponível. Iniciantes que procuram uma ferramenta projetada para facilitar a programação Java e o mundo dos IDEs Java podem conferirAzulJ, criado pelo King's College London.

Pegando os principais exemplos de código

Não importa qual IDE ou editor você usa, você vai querer pegar os exemplos de código do livro no GitHub. Embora muitas vezes incluamos listagens completas de fontes ao discutir tópicos específicos, muitas vezes optamos por deixar de fora coisas como instruções de importação ou pacote, ou a estrutura de classes anexa por questões de brevidade e legibilidade. Os exemplos de código para

download pretendem ser completos para ajudar a reforçar as discussões do livro.



Você pode visitar o GitHub em um navegador para percorrer os exemplos individuais sem baixar nada. Basta ir para o *aprenderjava6e* repositório. (Se esse link não funcionar, basta ir paragithub.come pesquise o termo "learnjava6e".) Pode valer a pena dar uma olhada no GitHub em geral, pois ele se tornou o principal ponto de encontro para desenvolvedores de código aberto e até mesmo para equipes corporativas. Você pode consultar o histórico de um repositório, bem como relatar bugs e discutir questões relacionadas ao código.

O nome do site refere-se à ferramenta git, um sistema de controle de código-fonte ou gerenciador de código-fonte, que os desenvolvedores usam para gerenciar revisões entre equipes para projetos de código. Mas lembre-se deFigura 2-14que você não precisa usar o git. Você pode simplesmente pegar todo o lote de exemplos baixando o branch principal do projeto como umArquivo ZIP. Depois de baixado, basta descompactar o arquivo em uma pasta onde você possa encontrar facilmente os exemplos. Você deverá ver uma estrutura de pastas semelhante aFigura A-2.

Figura A-2. Estrutura de pastas para os exemplos de código

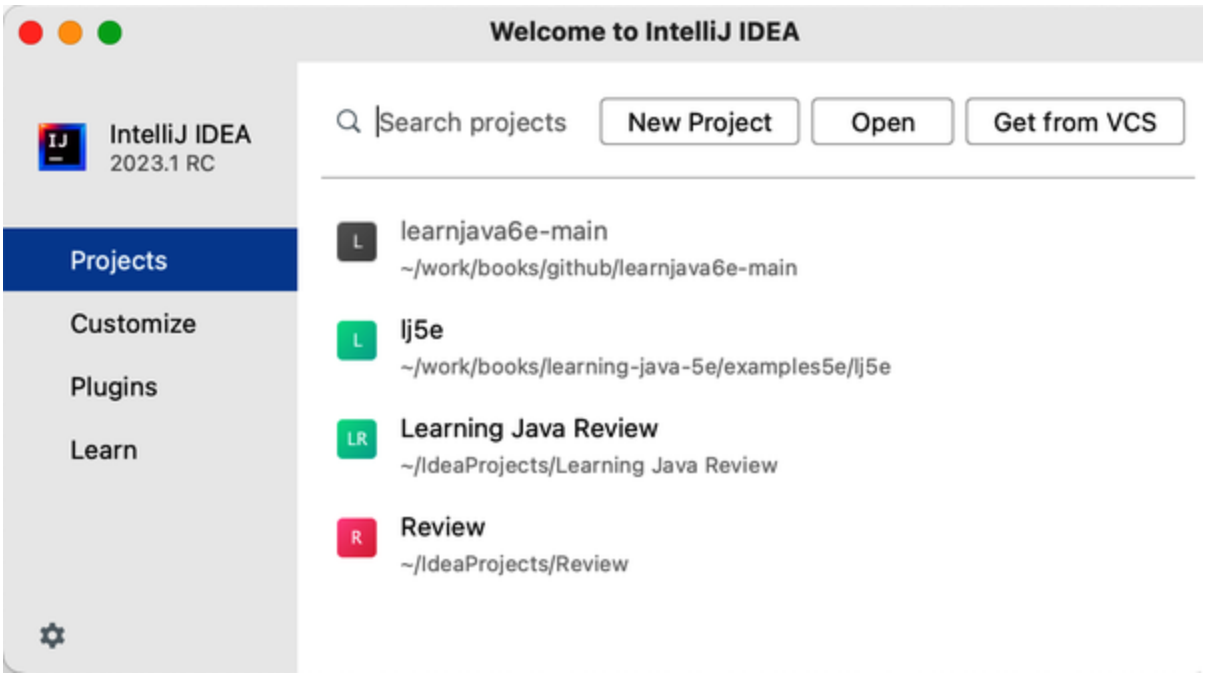
Se você está curioso sobre o Git, mas seu sistema ainda não possui o comando git disponível, você pode baixá-lo noSite Git. O GitHub tem seu próprio site para ajudá-lo a aprender sobre o git emtente.github.io. Depois que o git estiver instalado, você pode clonar o projeto em uma pasta no seu computador. Você pode trabalhar a partir desse clone ou mantê-lo como uma cópia limpa dos exemplos de código. Como um pequeno bônus, se publicarmos alguma correção ou atualização no futuro, você também poderá sincronizar facilmente sua pasta clonada.

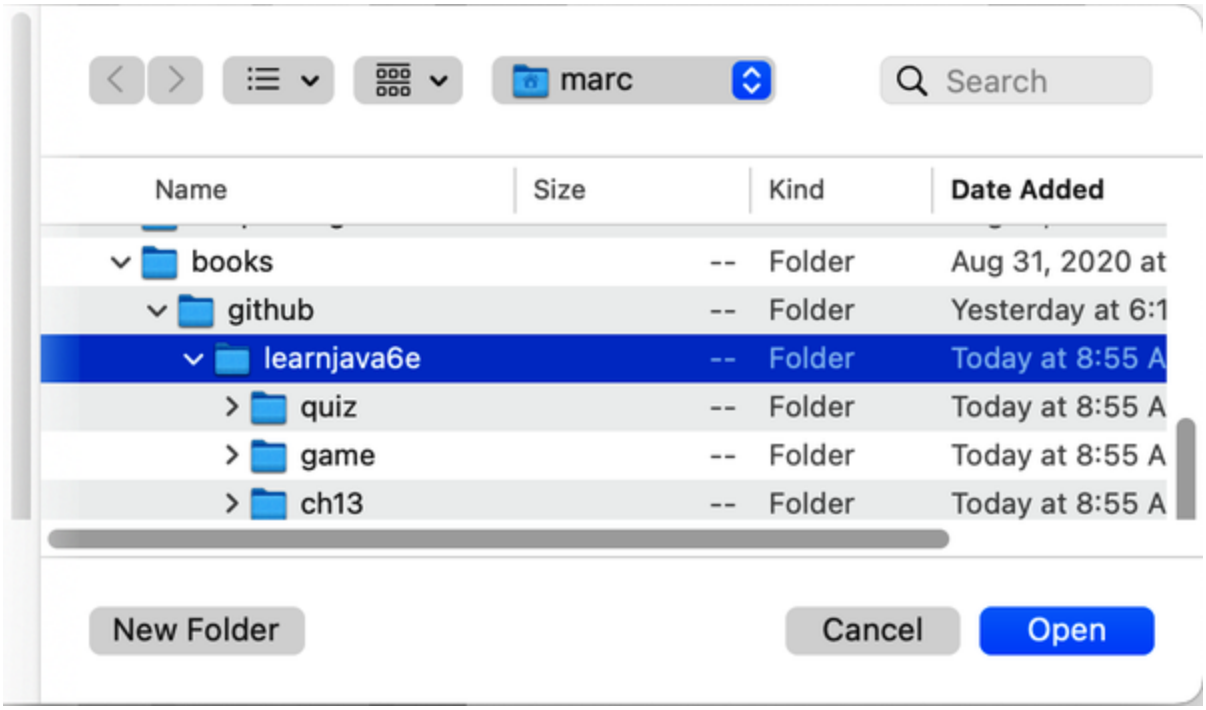
Importando os exemplos

Antes de importarmos qualquer coisa para o IntelliJ IDEA, você pode renomear a pasta onde baixou os exemplos de código do GitHub. Após clonar ou descompactar, você provavelmente terá uma pasta chamada learnjava6e-main. Esse é um nome perfeitamente adequado, mas se você quiser algo mais amigável (ou mais curto), vá em frente e renomeie a pasta agora. Optamos por renomear a pasta learnjava6e (sem o sufixo -main).

Inicie o IntelliJ IDEA e selecione a opção Abrir na tela de boas-vindas mostrada

emFigura A-3. Se você já usou o IntelliJ IDEA e não vê a tela de boas-vindas, também pode selecionar Arquivo → Abrir na barra de menu.


Welcome to IntelliJ IDEA
IntelliJ IDEA
2023.1 RC
Search projects
New Project
Open
Get from VCS
Projects
Customize
Plugins
Learn
learnjava6e-main
~/work/books/github/learnjava6e-main
lj5e
~/work/books/learning-java-5e/examples5e/lj5e
Learning Java Review
~/IdeaProjects/Learning Java Review
Review
~/IdeaProjects/Review


marc
Search
Name
Size
Kind
Date Added
books -- Folder Aug 31, 2020 at
github -- Folder Yesterday at 6:1
learnjava6e -- Folder Today at 8:55 A
quiz -- Folder Today at 8:55 A
game -- Folder Today at 8:55 A
ch13 -- Folder Today at 8:55 A
New Folder
Cancel
Open

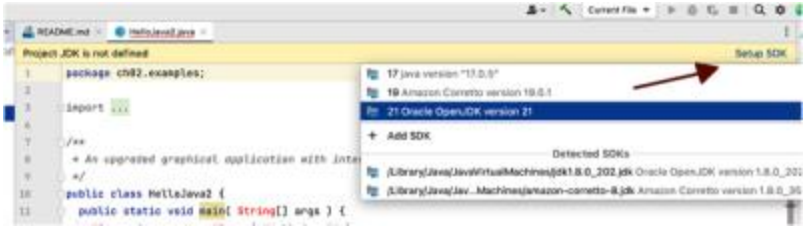


Figura A-3. A tela de boas-vindas do IntelliJ IDEA

Navegue até a pasta de exemplo de código, conforme mostrado emFigura A-4.

Certifique-se de selecionar a pasta superior que contém todos os capítulos, e não uma das pastas de capítulos individuais.

Figura A-4. Importando a pasta de exemplos de código

Depois de abrir a pasta de exemplos, você pode ser solicitado a “confiar” na pasta que contém os exemplos. IDEA faz esta pergunta para confirmar se as classes potencialmente executáveis na pasta são seguras para execução.

Também pode ser necessário especificar a versão do Java que deseja usar para compilar e executar os exemplos. Usando a hierarquia do projeto à esquerda, abra a pasta ch02/examples e clique na classe HelloJava. O arquivo fonte da nossa classe deve aparecer à direita. Se você vir um banner amarelo claro semelhante ao mostrado

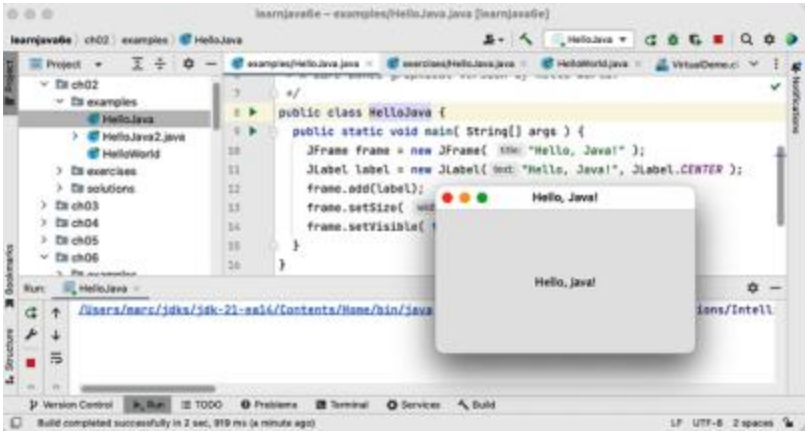
emFigura A-5, clique no texto vinculado do Setup SDK no canto superior direito. (SDK

significa Software Development Kit e, no nosso caso, é sinônimo de JDK.) Escolha o JDK que deseja usar na caixa de diálogo exibida.

Figura A-5. Selecionando um JDK

Para este exemplo, escolhemos a versão de suporte de longo prazo (21), mas você pode escolher qualquer versão 19 ou superior que possa ter instalada. (Você sempre pode alterar sua seleção ou ativar recursos de

visualização usando a caixa de diálogo Arquivo → Estrutura do Projeto mostrada emFigura A-1.)



Para verificar se tudo está funcionando, clique com o botão direito na classe HelloJava na árvore à esquerda e selecione o item Run HelloJava.main() no menu de contexto.

Figura A-6. Executando um aplicativo diretamente no IDEA

Parabéns! O IntelliJ IDEA está configurado e pronto para você começar a explorar o incrível e gratificante mundo da programação Java.

Apêndice B. Respostas dos Exercícios

Este apêndice contém respostas (e geralmente um pouco de contexto) para as questões de revisão no final de cada capítulo. As respostas dos exercícios de código estão incluídas no download do código-fonte dos programas de exemplo, na pasta de exercícios.Apêndice Apossui detalhes sobre como obter a fonte e configurá-la no IntelliJ IDEA.

Capítulo 1: Uma Linguagem Moderna

1. Qual empresa mantém Java atualmente?

Enquanto o Java foi desenvolvido na década de 1990 na Sun Microsystems, a Oracle comprou a Sun (e, portanto, o Java) em 2009. A Oracle mantém a propriedade e é um parceiro ativo no desenvolvimento e distribuição de seu próprio JDK comercial e do OpenJDK de código aberto.

2. Qual é o nome do kit de desenvolvimento de código aberto para Java?

A versão de código aberto do JDK é conhecida como OpenJDK.

3. Cite os dois componentes principais que desempenham um papel na

abordagem Java para executar bytecode com segurança.

Java possui muitos recursos relacionados à segurança, mas os principais componentes em jogo em cada aplicativo Java são o carregador de classes e o verificador de bytecode.

Capítulo 2: Uma primeira aplicação

1. Qual comando você deve usar para compilar um arquivo fonte Java?

Se você estiver trabalhando em um terminal, o comando javac compila os arquivos de origem Java. Embora os detalhes geralmente fiquem ocultos ao usar um IDE como o IntelliJ IDEA, o IDE também usa javac nos bastidores.

2. Como a JVM sabe por onde começar quando você executa um aplicativo Java?

Qualquer classe Java executável deve ter um método main() definido. A JVM

usa esse método como ponto de entrada.

3. Você pode estender mais de uma classe ao criar uma nova classe?

Não. Java não suporta herança múltipla direta de múltiplas classes separadas.

4. Você pode implementar mais de uma interface ao criar uma nova classe?

Sim. Java permite implementar quantas interfaces forem necessárias. O uso de interfaces fornece aos programadores a maioria dos recursos úteis de herança múltipla, sem muitas armadilhas.

5. Qual classe representa a janela em uma aplicação gráfica?

A classe JFrame representa a janela principal usada em aplicativos gráficos Java, embora capítulos posteriores apresentem algumas classes de nível inferior que também podem criar janelas especializadas.

Exercícios de código

Geralmente não listaremos as soluções de código neste apêndice, mas queremos facilitar a verificação da sua solução para este primeiro programa. A versão em texto simples de “Adeus, Java!” deve ser algo assim:

classe pública AdeusJava {

public static void main(String[] args) {

```
System.out.println("Adeus, Java!");

}

}
```

Para a versão gráfica, seu código deve ser semelhante a este:

```
importar javax.swing.*;

classe pública AdeusJava {

public static void main(String[] args) {

JFrame frame = new JFrame("Exercícios do Capítulo 2");

frame.setSize(300, 150);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

Rótulo JLabel = new JLabel("Adeus, Java!",
JLabel.CENTER);

frame.add(rótulo);

frame.setVisible (verdadeiro);

}

}
```

Observe que adicionamos EXIT_ON_CLOSE extra que introduzimos no HelloJava2 para que o aplicativo seja encerrado corretamente quando você fechá-lo. Se estiver usando o IDEA, você pode executar qualquer uma das classes usando o botão verde Play dentro do IDE. Se estiver usando um terminal, você pode mudar para o

diretório onde GoodbyeJava.java está localizado e digitar os seguintes comandos:

% javac AdeusJava.java

% java AdeusJava

Capítulo 3: Ferramentas do Comércio

1. Qual instrução dá acesso aos componentes Swing em seu aplicativo?

A instrução import carrega as informações que o compilador precisa da classe ou pacote especificado. Para componentes Swing, você normalmente importa o pacote inteiro com import javax.swing.*;.

2. Qual variável de ambiente determina onde o Java procurará os arquivos de classe ao compilar ou executar?

A variável de ambiente CLASSPATH contém uma lista de diretórios contendo outras classes ou arquivos JAR disponíveis para compilação e execução. Se você estiver usando um IDE, o CLASSPATH ainda estará definido, mas não é algo que você normalmente edita.

3. Quais opções você pode usar para visualizar o conteúdo de um arquivo JAR

sem descompactá-lo?

Você pode executar o seguinte comando para mostrar o conteúdo de um

arquivo JAR sem realmente descompactá-lo no diretório atual:

% jar tvf MyApp.jar

Os sinalizadores tvf representam o índice (t), detalhado (v) e arquivo (f seguido por um nome de arquivo).

4. Qual entrada é necessária no arquivo MANIFEST.MF para tornar um arquivo JAR executável?

Você deve incluir uma entrada para Main-Class que forneça o nome completo de uma classe com um método main() válido para tornar executável um

determinado arquivo JAR.

5. Qual ferramenta permite que você experimente o código Java de forma interativa?

Você pode executar o jshell a partir de um terminal para testar o código Java simples de forma interativa.

Exercícios de código

Você encontrará nossas soluções para os exercícios de código na pasta ch03/solutions. (Apêndice Atem detalhes sobre como baixar os exemplos.) Nossas soluções não são a única — nem mesmo a melhor — maneira de resolver esses problemas. Tentamos apresentar um código limpo e de fácil manutenção que siga as melhores práticas, mas sempre há outras maneiras de abordar um problema de codificação. Esperamos que você seja capaz de escrever e executar suas próprias respostas, mas aqui estão algumas dicas se você tiver dúvidas.

Para tornar o arquivo executável hello.jar, faremos todo o nosso trabalho na pasta ch03/exercises em um terminal. (Você certamente pode fazer esse tipo de trabalhodentro da IDEIAtambém.) Vá em frente e abra um terminal e mude para essa pasta.

Antes de criarmos o arquivo JAR, precisamos editar o arquivo manifest.mf. Adicione a entrada Main-Class. O arquivo final deve ficar parecido com isto:

Versão do manifesto: 1.0

Criado por: 11.0.16

Classe principal: ch03.exercises.HelloJar

Agora você pode criar e testar seu arquivo JAR com os seguintes comandos:

% jar cvmf manifesto.mf hello.jar *.class

% java -jar olá.jar

Lembre-se de que o elemento m nas flags é necessário para incluir nosso manifesto.

Também vale a pena lembrar que a ordem dos sinalizadores m e f determina a ordem dos argumentos de linha de comando manifest.mf e hello.jar a seguir. Você se lembra de como observar o conteúdo do seu JAR recém-criado para verificar se o manifesto está lá?1

Capítulo 4: A Linguagem Java

1. Qual formato de codificação de texto o JAVA usa por padrão em classes compiladas?

Por padrão, Java usa a codificação Unicode Transformation Format (UTF-8) de 8 bits. A codificação de 8 bits (ou um byte) pode acomodar caracteres únicos e multibyte.

2. Quais caracteres são usados para incluir um comentário de múltiplas linhas?

Esses comentários podem ser aninhados?

Java toma emprestado de C e C++ para sua sintaxe de comentários. Os

comentários de linha única começam com duas barras (//), enquanto os

comentários de múltiplas linhas são colocados entre pares /* e */. O estilo

multilinha também pode ser usado para incorporar pequenos comentários no meio de uma linha de código.

3. Quais construções de loop o Java suporta?

Java suporta o loop for (estilo C tradicional e uma forma aprimorada para iterar coleções), o loop while e o loop do/while.

4. Em uma cadeia de testes if/else if/else, o que acontece se múltiplas condições forem verdadeiras?

O bloco associado ao primeiro teste avaliado como verdadeiro será executado.

Após a conclusão desse bloco, o controle é retomado após toda a cadeia –

independentemente de quantos outros testes também teriam retornado

verdadeiros.

5. Se você quisesse armazenar a capitalização total do mercado de ações dos EUA (cerca de US$ 31 trilhões no fechamento do ano fiscal de 2022) como dólares inteiros, que tipo de dados primitivos você poderia usar?

Você poderia usar o tipo inteiro longo; pode armazenar números de até 9

quintilhões (positivos ou negativos). Embora você também possa usar o tipo double, à medida que os números aumentam, sua precisão diminui. E como

“dólares inteiros” não implica frações, um tipo inteiro faz mais sentido.

6. Qual é o valor da expressão 18 – 7 * 2?

Esta é uma questão de ordem de precedência para garantir que seu professor de álgebra do ensino médio finalmente receba algum crédito depois de todos aqueles “mas quando vou usar isso?” consultas. A multiplicação de 7 e 2

ocorrerá primeiro, depois a subtração. A resposta final é 4. (Você pode ter obtido 22, que resulta da execução das operações da esquerda para a direita. Se você realmente deseja esse resultado, pode colocar a parte 18 – 7 entre parênteses… assim à parte.)

7. Como você criaria um array contendo os nomes dos dias da semana?

Você pode criar e inicializar um array usando chaves. Para os dias da semana, precisamos de um array de Strings, assim:

String[] diaNomes = {

"Domingo Segunda Terça Quarta",

"Quinta sexta sábado"

};

O espaçamento entre os nomes na lista é opcional. Você pode listar tudo em uma linha, listar cada nome em sua própria linha ou alguma combinação como fizemos aqui.

Exercícios de código

1. Existem várias maneiras de imprimir o original a e b na tela junto com o maior denominador comum calculado. Você pode usar uma instrução print() (não println()) antes de começar a calcular e, em seguida, usar println() com a resposta para finalizar a saída. Ou você pode armazenar uma cópia de aeb em um segundo conjunto de variáveis antes de iniciar o cálculo. Depois de encontrar a resposta, você pode imprimir os valores copiados junto com o resultado.

2. Para gerar os dados triangulares em uma linha simples, você pode usar os mesmos loops aninhados usados para preencher o triângulo:

for (int i; i <triângulo.comprimento; i++)

for (int j; j <triângulo[i].comprimento; j++)

System.out.println(triângulo[i][j]);

Exercícios Avançados

1. Para apresentar sua saída em um triângulo visual, use uma instrução print() dentro do loop j interno. (Certifique-se de imprimir um espaço após cada número também.) Depois que o loop interno terminar, você pode usar um println() vazio para encerrar a linha e estar pronto para a próxima linha.

Capítulo 5: Objetos em Java

1. Qual é a principal unidade organizadora em Java?

A principal “unidade” em Java é uma classe. Muitas outras estruturas

desempenham papéis importantes, é claro, mas você não pode usar nenhuma dessas outras coisas sem pelo menos uma classe.

2. Qual operador você usa para criar um objeto (ou instância) de uma classe?

O operador new instancia um objeto de uma classe e chama o construtor apropriado.

3. Java não suporta herança múltipla clássica. Que mecanismos o Java oferece como alternativas?

Java usa interfaces para atingir a maioria dos objetivos de herança múltipla encontrados em muitas outras linguagens orientadas a objetos.

4. Como você pode organizar várias classes relacionadas?

Você coloca classes relacionadas em um pacote. No seu sistema de arquivos, um pacote aparece como pastas aninhadas. No seu código, os pacotes usam nomes separados por pontos.

5. Como você inclui classes de outros pacotes para uso em seu próprio código?

Você pode importar outras classes individuais ou pacotes inteiros para seu próprio uso.

6. Como você chama uma classe definida dentro do escopo de outra classe? Quais são alguns recursos que tornam essa classe útil em algumas circunstâncias?

Uma classe simples definida entre chaves de outra classe (não apenas no mesmo arquivo) é chamada de classe interna. As classes internas têm acesso a todas as variáveis e métodos da classe externa — incluindo membros privados.

Eles podem ser usados para ajudar a dividir seu código em partes gerenciáveis e reutilizáveis, ao mesmo tempo que fornecem um bom controle sobre quem mais pode usá-los.

7. Como você chama um método projetado para ser substituído que possui nome, tipo de retorno e lista de argumentos, mas não possui corpo?

Métodos definidos apenas com suas assinaturas são conhecidos como métodos abstratos. Incluir um método abstrato em uma classe também torna a classe abstrata. A classe abstrata não pode ser instanciada. Você deve criar uma subclasse e então fornecer um corpo real para o método abstrato usá-la.

8. O que é um método sobrecarregado?

Java permite que você use o mesmo nome de método com diferentes tipos ou números de argumentos. Se dois métodos compartilham o mesmo nome,

dizemos que estão sobrecarregados. A sobrecarga torna possível criar um lote de métodos que realizam o mesmo trabalho lógico em argumentos diferentes.

O exemplo clássico de método sobrecarregado em Java é System.out.println() que pode pegar vários tipos diferentes de argumentos e convertê-los todos em strings para impressão em um terminal.

9. Se você quiser ter certeza de que nenhuma outra classe poderá usar uma variável que você definiu, qual modificador de acesso você deve usar?

O modificador de acesso privado para uma variável (ou um método, ou mesmo uma classe interna inteira) restringe seu uso à classe onde está definida.

Exercícios de código

1. Para o primeiro problema em nosso Zoo, você só precisa adicionar uma instrução print() ao método speak() vazio na classe interna Gibbon. Esperamos que o exemplo do Leão seja fácil de seguir.

2. Adicionar outro animal também deve ser simples; você pode copiar toda a classe Lion. Renomeie a classe e imprima um ruído apropriado no método speak(). Você também precisará replicar algumas linhas no método listen() para que o som do seu animal seja adicionado à saída.

3. Para refatorar o método listen(), notamos que a saída para cada animal é muito semelhante, mas o nome do animal obviamente muda para cada animal. Se

movermos esse nome para as respectivas classes dos animais, podemos criar um loop cujo corpo imprime os detalhes (nome e ruído) de um animal. Em seguida, iteramos sobre nossa variedade de animais. Se você criar outro animal, basta adicionar uma instância da sua nova classe interna ao array.

4. A classe AppleToss para este exercício faz parte do pacote exerce.ch05. (A pasta do jogo contém o jogo completo, com todos os recursos que

construiremos ao longo do livro. As classes nessa pasta fazem parte do pacote do jogo. Você pode compilar e executar essa versão, mas ela possui vários recursos ainda não discutimos.) Para compilar o jogo a partir de um terminal, você pode mudar para o diretório ch05/exercises e compilar as classes Java lá, ou permanecer na pasta de nível superior onde você descompactou o código-fonte e fornecer o caminho ao compilar:

% javac ch05/exercícios/AppleToss.java

Para executar o jogo, você precisa estar na pasta de nível superior. A partir daí, você pode executar a classe exerces.ch05.AppleToss com o comando java.

Exercícios Avançados

1. Esperançosamente, adicionar um Hedge parece simples. Você pode começar com uma cópia da classe Tree. Renomeie o arquivo Hedge.java. Edite a classe para refletir nosso novo obstáculo Hedge e atualize seu método

paintComponent(). Dentro da classe Field, você precisará adicionar uma variável de membro para o Hedge. Crie um método setupHedge() semelhante a setupTree() e certifique-se de incluir seu hedge no método paintComponent() de Field.

Por último, mas não menos importante, atualize o método

setupFieldForOnePlayer() para chamar nosso método setupHedge(). Compile e execute o jogo como você fez no exercício anterior. Suas novas sebes devem aparecer!

Capítulo 6: Tratamento de Erros

1. Qual instrução você usa para gerenciar possíveis exceções em seu código?

Você pode usar um bloco try/catch em torno de qualquer instrução ou grupo de instruções que possa gerar uma exceção.

2. Quais exceções o compilador exige que você manipule ou lance?

Em Java, o termo exceção verificada refere-se a uma categoria de exceções que o compilador entende e exige que o programador reconheça. Você pode usar um bloco try/catch dentro de um método onde podem ocorrer exceções

verificadas ou pode adicionar a exceção à cláusula throws do método em sua assinatura.

3. Onde você coloca qualquer código de “limpeza” que sempre deseja executar depois de usar alguns recursos em um bloco try?

A cláusula finalmente será executada no final de um bloco try,

independentemente do que acontecer. Se não houver problemas, o código da cláusula finalmente será executado. Se houver uma exceção e um bloco catch lidar com isso, finalmente ainda será executado. Se ocorrer uma exceção que não seja tratada, a cláusula

finalmente ainda será executada antes que o controle seja transferido de volta para o método de chamada.

4. As asserções têm uma grande penalidade de desempenho quando são

desativadas?

Não. Isso ocorre intencionalmente. As asserções devem ser mais usadas durante o desenvolvimento ou depuração. Quando você os desliga, eles são ignorados. Mesmo em um aplicativo de produção, você pode deixar asserções em seu código. Se um usuário relatar um problema, as asserções poderão ser ativadas temporariamente para permitir que o usuário colete qualquer saída e ajude você a encontrar a causa.

Exercícios de código

1. Para compilar o arquivo Pause.java, você precisa adicionar um bloco try/catch ao redor da chamada para Thread.sleep(). Para este exercício simples, você só precisa encapsular a linha Thread.sleep().

2. As declarações de asserções que precisamos terão o seguinte formato: assert x > 0: "X é muito pequeno";

assert y > 0: "Y é muito pequeno";

A grande questão é: onde devemos colocá-los? Precisamos apenas verificar a posição inicial da nossa mensagem, por isso não queremos as asserções dentro do método paintComponent(). Um lugar melhor seria no construtor

HelloComponent0(), talvez logo após armazenarmos o argumento da

mensagem fornecida.

Para testar as asserções, você precisará editar o arquivo de origem para alterar os valores x ou y e recompilar.

Exercícios Avançados

1. Sua classe GCDException provavelmente será parecida com isto:

pacote ch06.exercícios;

classe pública GCDException estende exceção {

privado int ruimA;

privado int ruimB;

GCDException(int a, int b) {

super("Nenhum fator comum para " + a + ", " + b);
ruimA = a;

ruimB = b;

}

public int getA() { return badA; }

public int getB() { return badB; }

}

Você pode testar o resultado do cálculo do GCD com uma simples instrução if.

Se o resultado for 1, você pode lançar sua nova GCDException com nossos argumentos a e b originais

para seu construtor, assim:

se (uma == 1) {

lançar nova GCDException(a1, b1);

}

// ...

Capítulo 7: Coleções e Genéricos

1. Se você quiser armazenar uma lista de contatos com nomes e números de telefone, que tipo de coleção funcionaria melhor?

Um mapa resolveria o problema. As chaves podem ser strings simples

contendo o nome do contato e os valores podem ser um número longo simples (embora agrupado). Ou você poderia criar uma classe Person e uma classe PhoneNumber, e o mapa poderia usar suas classes personalizadas.

2. Qual método você usa para obter um iterador para os itens de um conjunto?

O método iterator() de nome criativo da interface Collection obterá seu iterador.

3. Como você pode transformar uma lista em um array?

Você pode usar o método toArray() para transformar uma Lista em um array do tipo Object ou em um array do tipo parametrizado da lista.

4. Como você pode transformar um array em uma lista?

A classe auxiliar Arrays inclui o prático método asList() que aceita um array e retorna uma lista parametrizada do mesmo tipo.

5. Qual interface você deve implementar para classificar uma lista usando o método Collections.sort()?

Embora existam muitas maneiras de classificar coleções, uma lista de objetos Comparable (ou seja, objetos cuja classe implementa a interface Comparable) pode usar o método sort() padrão fornecido pela classe auxiliar Collections.

Exercícios de código

1. Como mencionamos neste capítulo, você não pode ordenar diretamente um mapa simples da mesma forma que ordena uma lista ou array. Mesmo os

conjuntos normalmente não são classificáveis.2Você pode classificar uma lista, portanto, usar o método keySet() para preencher uma lista deve fornecer o que você precisa:

```
List<Integer> ids = new ArrayList<>
(employees.keySet());

ids.sort();

for (ID inteiro: ids) {

System.out.println(id + ": " + funcionários.get(id));

}
```

2. Esperançosamente, a expansão para suportar múltiplas coberturas parece simples para você. Na

maioria das vezes, apenas duplicamos qualquer código que já temos para as árvores. E o uso de Listas nos permite usar o loop for aprimorado para percorrer rapidamente todos os nossos hedges:

```
// Arquivo: ch07/exercises/Field.java

List<Hedge> hedges = new ArrayList<>();

// ...

protegido void paintComponent(Gráficos g) {

// ...

for (Hedge h : hedges) {

h.draw(g);

}

// ...

}
```

Exercícios Avançados

1. Você pode usar a saídavalues() para criar e classificar uma lista semelhante à solução do Exercício de Código 1. A parte interessante deste exercício é implementar a interface Comparable com a classe Employee. (Na verdade, na pasta ch07/solutions, a classe de funcionário classificável é Employee2.

Queríamos deixar a classe Employee original como uma solução válida para o primeiro exercício.) Aqui está um exemplo de comparação de strings, usando os nomes dos funcionários:

```
classe pública Employee2 implementa
Comparable<Employee2> {

// ...

public int compareTo(Funcionário2 outro) {

// Vamos ser um pouco sofisticados e classificar um nome
construído

String meuNome = último + ", " + primeiro;

String outroNome = outro.último + ", " + outro.primeiro;

return meuNome.compareToIgnoreCase(outroNome);

}

// ...

}
```

Claro, você poderia realizar outras comparações usando os outros atributos Employee. Experimente brincar com outras ordenações e veja se obtém os resultados esperados. E se você quiser se aprofundar ainda mais, dê uma olhada na classe java.util.TreeMap como uma forma de armazenar seus

funcionários de maneira ordenada, sem precisar do desvio de conversão de lista.

Capítulo 8: Texto e utilitários principais

1. Qual classe contém a constante π? Você precisa importar essa classe para usar π?

A classe java.lang.Math contém a constante PI. Todas as classes do pacote java.lang são importadas por padrão; nenhuma importação explícita é

necessária para usá-los.

2. Qual pacote contém substitutos melhores e mais novos para a classe java.util.Date original?

O pacote java.time contém uma ampla variedade de classes de qualidade para lidar com datas, horas, carimbos de data/hora (ou “instantes” que consistem em uma data e uma hora) e intervalos de tempo ou durações.

3. Qual classe você usa para formatar uma data para uma saída amigável?

A classe DateFormat no pacote java.text possui um mecanismo de formatação maravilhosamente flexível (embora ocasionalmente opaco) para renderizar datas e horas.

4. Que símbolo você usa em uma expressão regular para ajudar a combinar as palavras “sim” e “sim”?

Você pode usar o operador de alternância, | (pipe vertical), para criar uma expressão como yes|yup para usar como padrão.

5. Como você converteria a string “42” no número inteiro 42?

Todos os vários wrappers numéricos possuem métodos de conversão de string.

Para um número inteiro como 42, o método Integer.parseInt() funcionaria. As classes wrapper fazem parte do pacote java.lang.

6. Como você compararia duas strings para ver se elas correspondem, ignorando qualquer capitalização, como “sim” e “SIM”?

A classe String possui dois métodos principais de comparação: equals() e equalsIgnoreCase(). Este último ignoraria a capitalização, como o próprio nome sugere.

7. Qual operador permite a concatenação simples de strings?

Java geralmente não suporta sobrecarga de operador, mas o sinal de mais (+) realiza adição quando usado com tipos base numéricos e concatenação quando usado com objetos String. Se você usar + para “adicionar” uma string e um número, o resultado será uma string. (Portanto, 7 + “tem sorte” resultaria na string “7 tem sorte”. Observe que a concatenação não insere nenhum espaço em branco. Se você estiver montando uma frase típica, deverá adicionar seu próprio espaçamento entre as partes.)

Exercícios de código

Há muitas maneiras de atingir os objetivos deste exercício. Testar o número de argumentos deve ser simples. Em seguida, você pode usar alguns dos recursos da classe String para descobrir se possui a palavra-chave aleatória ou um par de coordenadas. Você pode dividir() as coordenadas ou criar uma expressão regular para separar os valores numéricos. Ao criar coordenadas aleatórias, você pode usar Math.random(), semelhante a como posicionamos as árvores para nosso jogo

em“Matemática em ação”.

Capítulo 9: Tópicos

1. O que é um tópico?

Um thread representa um “thread de execução” dentro de um programa.

Threads têm seu próprio estado e podem ser executados independentemente de outros threads. Normalmente você usa threads para lidar com tarefas de longa execução que podem ser colocadas em segundo plano enquanto tarefas mais importantes podem continuar a fazer seu trabalho. Java tem threads de plataforma (ligados individualmente aos threads nativos fornecidos pelo sistema operacional) e threads virtuais (construções Java puras que retêm a semântica e os benefícios dos threads nativos sem a sobrecarga do sistema operacional).

2. Que palavra-chave você pode adicionar a um método se quiser que os threads

“se revezem” ao chamá-lo? (O que significa que dois threads não devem executar o método ao mesmo tempo para evitar corromper os dados

compartilhados.)

Você pode usar o modificador sincronizado em qualquer método que leia ou grave dados compartilhados. Se dois threads precisarem usar o mesmo

método, o primeiro thread define um bloqueio que impede o segundo thread de chamar o método. Assim

que o primeiro thread terminar, o bloqueio será liberado e o segundo thread poderá prosseguir.

3. Quais sinalizadores permitem compilar um programa Java que inclui código de recurso de visualização?

Ao compilar uma classe Java que depende de um recurso de visualização, você deve fornecer os sinalizadores --enable-preview e -source ou --release para javac.

4. Quais sinalizadores permitem executar um programa Java que inclui código de recurso de visualização?

Ao executar uma classe de compilação que inclui um recurso de visualização, você só precisa fornecer o sinalizador --enable-preview.

5. Quantos threads de plataforma um thread nativo pode suportar?

Apenas um. Criar um thread de plataforma usando a classe Thread com um destino Runnable ou usar algo como ExecutorService no pacote

java.util.concurrent requer que o sistema operacional forneça um thread também.

6. Quantos threads virtuais um thread nativo pode suportar?

Um único thread nativo suportará muitos threads virtuais. O Projeto Loom teve como objetivo separar os threads usados em programas Java dos threads gerenciados pelo sistema operacional. Para determinados cenários, threads virtuais leves apresentam desempenho muito melhor quando Java é

responsável por seu agendamento. Não existe uma proporção fixa de virtual para nativo, mas o principal insight com threads virtuais é que o número de threads virtuais não está vinculado ao número de threads nativos.

7. A afirmação é x = x + 1; uma ação atômica para a variável int x?

Não. Embora pareça uma operação tão pequena, há várias etapas de baixo nível envolvidas. Qualquer uma dessas etapas de baixo nível pode ser interrompida e o valor de x pode ser afetado negativamente. Você pode usar um

AtomicInteger ou agrupar a instrução em um bloco sincronizado se precisar garantir um incremento seguro para threads.

8. Qual pacote inclui versões thread-safe de classes de coleção populares como Queue e Map?

O pacote java.util.concurrent contém diversas classes de coleção que Java define como “simultâneas”, como ConcurrentLinkedQueue e

ConcurrentHashMap. A simultaneidade implica alguns outros comportamentos além de leituras e gravações seguras de thread, mas a segurança de thread é garantida.

Exercícios de código

1. Você fará a maior parte do seu trabalho dentro do método startClock(). (Você ainda terá que importar qualquer coisa além dos pacotes AWT e Swing que você usa, é claro.) Você pode criar uma classe separada, uma

classe interna ou uma classe interna anônima para lidar com o loop de atualização do relógio.

Lembre-se de que você pode solicitar uma atualização de elemento GUI chamando seu método repaint(). Java suporta alguns mecanismos para loops

“infinitos”. Você pode usar algo como while (true) { … } ou o loop “forever”

habilmente chamado: for (;;) { … }. Não se esqueça de iniciar seu tópico quando tudo estiver no lugar!

2. Esperamos que este exercício seja bastante simples para você. Como prova da compatibilidade geral dos threads virtuais com a base de código existente em Java, você deve alterar apenas algumas linhas onde a animação de

demonstração do lançamento da maçã começa. Nesta iteração do jogo, todo o código de configuração e início acontece na classe Field. Procure códigos como new Thread() ou new Runnable(). Você poderá reutilizar a lógica de animação real sem alterações.

Capítulo 10: Entrada e Saída de Arquivo

1. Como você poderia verificar se um determinado arquivo já existe?

Existem várias maneiras de verificar se um arquivo existe, mas duas das mais simples dependem de métodos auxiliares do pacote java.io ou java.nio. Uma instância de java.io.File pode usar o método exist(). O método estático java.nio.file.Files.exists() pode testar um objeto Path para ver se existe um arquivo representado.

2. Se você tiver que trabalhar com um arquivo de texto legado usando um esquema de codificação antigo, como ISO 8859, como configurar um leitor para converter adequadamente esse conteúdo para UTF-8?

Você pode fornecer um conjunto de caracteres apropriado

(java.nio.charset.Charset) ao construtor do FileReader para converter com segurança o arquivo em strings Java.

3. Qual pacote tem as melhores classes para E/S de arquivos sem bloqueio?

Um dos principais recursos do pacote java.nio e seus subpacotes é o suporte para E/S sem bloqueio.

4. Que tipo de fluxo de entrada você pode usar para analisar um arquivo binário, como uma imagem compactada em JPEG?

Em java.io, você poderia usar a classe DataInputStream. Para NIO, canais e buffers (como ByteBuffer) suportam naturalmente dados binários.

5. Quais são os três fluxos de texto padrão integrados à classe System?

A classe System dá acesso a dois fluxos de saída, System.out e System.err, e um fluxo de entrada, System.in. Esses fluxos são anexados aos identificadores do sistema operacional para stdout, stderr e stdin, respectivamente.

6. Os caminhos absolutos começam em uma raiz (/ ou C:\, por exemplo). Onde começam os caminhos relativos?

Mais especificamente, para onde estão os caminhos relativos?

Os caminhos relativos são relativos ao “diretório de trabalho”, que

normalmente é onde você iniciou o programa se estiver usando a linha de comando para iniciar seu aplicativo. Na maioria dos IDEs, o diretório de trabalho é algo que pode ser configurado.

7. Como você recupera um canal NIO de um FileInputStream existente?

Se você já possui uma instância de FileInputStream, pode usar seu método getChannel() para retornar um FileChannel associado ao fluxo de entrada.

Exercícios de código

1. A primeira iteração do nosso Count precisa apenas de um dos utilitários discutidos no capítulo. Você poderá usar a classe File com o caminho fornecido como argumento de linha de comando. A partir daí, o método existe()

informará se você pode prosseguir ou se deve imprimir uma mensagem de erro amigável, e o método length() fornecerá o tamanho do arquivo, em bytes.

(A solução para este exemplo é Count1.java na pasta ch10/solutions.)

2. Para a segunda iteração que mostra a contagem de linhas e palavras no arquivo fornecido, você precisará ler e analisar o conteúdo do arquivo. Uma das classes

Reader seria ótima, mas existem várias maneiras de ler arquivos de texto.

Independentemente de como você abre o arquivo, você pode contar cada linha e depois dividi-la em palavras com algo como String.split() ou uma expressão regular. (A solução para este exercício é Count2.java.)

3. Não há nenhuma funcionalidade nova nesta terceira versão, mas esperamos que você aproveite esta oportunidade para experimentar algumas das classes e métodos NIO. Dê uma olhada nos métodos da classe java.nio.file.Files. Você ficará surpreso com o quanto esta classe auxiliar ajuda! (A solução para este exercício é Count3.java.)

Exercícios Avançados

1. Para esta atualização final, você pode gravar em um arquivo ou canal.

Dependendo de como você escolheu ler o conteúdo da versão 2 ou 3, isso pode representar um acréscimo bastante significativo à nossa aula. Você precisará verificar se o segundo argumento (se foi fornecido!) É gravável. Em seguida, use uma das classes que permite anexar, como RandomAccessFile, ou inclua a opção APPEND para um FileChannel. (A solução para este exercício é

Count4.java. Usamos o Count3 anterior com NIO, mas você pode começar com Count2 e usar as classes de E/S padrão.)

Capítulo 11: Abordagens Funcionais em Java

1. Qual pacote contém a maioria das interfaces funcionais?

Embora as interfaces funcionais estejam espalhadas por todo o JDK, você encontrará a maior parte das interfaces “oficiais” definidas no pacote java.util.function. Usamos aspas qualificativas em “oficial” porque qualquer interface com um único método abstrato (SAM) pode ser tratada como uma interface funcional.

2. Você precisa usar algum sinalizador especial ao compilar ou executar aplicativos Java que usam recursos funcionais como lambdas?

Não. Os muitos recursos de FP atualmente disponíveis em Java são membros plenos do JDK. Não há visualização ou sinalizadores de recurso necessários para compilar ou executar o código Java que os utiliza.

3. Como você cria expressões lambda com múltiplas instruções no corpo?

O corpo de uma expressão lambda segue as mesmas regras que o corpo de algo como um loop while: instruções únicas não exigem o uso de chaves, mas instruções múltiplas, sim. Você pode usar um bloco de chaves no lado direito do lambda se tiver várias instruções. Se o seu lambda retornar um valor, você também poderá usar a instrução return padrão.

4. As expressões lambda podem ser nulas? Eles podem retornar valores?

Sim, em ambos os aspectos. As expressões lambda executam a mesma gama de opções que os métodos. Você pode ter lambdas que não aceitam argumentos e não retornam valores. Você pode ter lambdas que

consomem argumentos, mas não produzem resultados. Você pode ter geradores lambda que não possuem argumentos, mas retornam valores. Finalmente, você pode ter lambdas que aceitam um ou mais argumentos e retornam um valor.

5. Você pode reutilizar um fluxo depois de processá-lo?

Não. Depois que você começar a processar um stream, pronto. A tentativa de reutilizar um fluxo resultará em uma exceção. Muitas vezes você pode

reutilizar a fonte original para criar um fluxo totalmente novo, mas idêntico, se necessário.

6. Como você poderia pegar um fluxo de objetos e convertê-lo em um fluxo de números inteiros?

Você pode usar uma das variações mapToInt() da classe Stream: mapToInt(), flatMapToInt() ou mapMultiToInt(). A classe IntStream, por sua vez, possui um método mapToObj() para converter na direção oposta.

7. Se você tiver um fluxo que filtra linhas vazias de um arquivo, que operação você poderia usar para informar quantas linhas tinham algum conteúdo?

A maneira mais fácil de contar as linhas restantes seria usar a operação de terminal count(). Você também pode criar seu próprio redutor ou usar um coletor e consultar o comprimento da lista resultante.

Exercícios de código

1. Esperamos que o uso de uma fórmula mais interessante para o nosso ajuste pareça simples. Não precisamos de nenhuma sintaxe alternativa ou métodos

extras; apenas colocamos a conversão Celsius, C = (F – 32) * 5/9, no corpo do nosso lambda, assim:

System.out.print("Convertendo para Celsius: ");

System.out.println(ajustar(amostra, s -> (s - 32) * 5/9));

Não é muito dramático, mas queremos ressaltar que lambdas podem abrir algumas possibilidades realmente inteligentes que vão além dos seus planos iniciais.

2. Você tem várias opções disponíveis para realizar essa tarefa de cálculo da média. Você poderia escrever um redutor de média. Você poderia coletar os salários em um contêiner mais simples e depois escrever seu próprio código de média. Mas se você olhar a documentação dos diferentes fluxos, notará que os fluxos numéricos já possuem a operação perfeita: média(). Ele retorna um objeto OpcionalDouble. Você ainda precisará iniciar o stream e usar algo como mapToInt() para obter seu stream de valores numéricos.

Exercícios Avançados

1. O coletor groupingBy() precisa de uma função que extraia uma chave de cada elemento do fluxo e retorne um mapa das chaves emparelhadas com uma lista de todos os elementos que possuem chaves correspondentes. Para nosso

exemplo PaidEmployee, você provavelmente terá algo assim:

Map<String, List<PaidEmployee>> byRoles =

funcionários.stream().collect(

Collectors.groupingBy(PaidEmployee::getRole));

O tipo de chave em nosso mapa deve corresponder ao tipo de objeto que extraímos em nossa operação groupingBy(). Usamos uma referência de método aqui, mas qualquer lambda que retorne a função do funcionário também

funcionaria.

Não queríamos complicar a solução anterior, por isso fizemos cópias do relatório e das classes de funcionários denominadas Report2 e PaidEmployee2, respectivamente.

Capítulo 12: Aplicativos de Desktop

1. Qual componente você usaria para exibir algum texto ao usuário?

Embora você possa usar uma variedade de componentes baseados em texto, JLabel é a maneira mais simples de mostrar ao usuário algumas informações textuais (somente leitura).

2. Quais componentes você usaria para permitir que o usuário insira texto?

Dependendo da quantidade de informações que você espera do usuário, você pode usar JTextField ou JTextArea. (Existem outros componentes de texto, mas servem a propósitos mais especializados.)

3. Que evento o clique em um botão gera?

Clicar em um botão ou em qualquer componente semelhante a um botão, como JMenuItem, gera um

ActionEvent.

4. Qual ouvinte você deve anexar ao JList se quiser saber quando o usuário altera o item selecionado?

Você pode implementar o ListSelectionListener do pacote javax.swing.event para receber eventos de seleção (e desmarcação) de lista de um objeto JList.

5. Qual é o gerenciador de layout padrão para JPanel?

Por padrão, o JPanel usa o gerenciador FlowLayout. Uma exceção notável a esse padrão é o painel de conteúdo do JFrame. Esse painel é um JPanel, mas o quadro altera automaticamente o gerenciador do painel para BorderLayout.

6. Qual thread é responsável pelo processamento de eventos em Java?

O thread de despacho de eventos, às vezes chamado de fila de despacho de eventos, gerencia a entrega de eventos e atualizações de componentes na tela.

7. Qual método você usaria para atualizar um componente como JLabel após a conclusão de uma tarefa em segundo plano?

Você pode usar SwingUtilities.invokeAndWait() se quiser esperar a atualização do rótulo antes de processar qualquer outro evento. Você poderia usar Swing Utilities.invokeLater() se não importa exatamente quando o rótulo é

atualizado.

8. Qual contêiner contém objetos JMenuItem?

Um objeto JMenu pode conter objetos JMenuItem, bem como objetos JMenu aninhados. Os próprios menus estão contidos em um JMenuBar.

Exercícios de código

1. Você pode lidar com o layout da calculadora de duas maneiras: você pode usar painéis aninhados ou usar o GridBagLayout. (Nossa solução em

ch12/solutions/Calculator.java usa um painel aninhado para os botões.) Comece de forma simples. Adicione o campo de texto ao topo do quadro. Em

seguida, adicione um botão ao centro. Agora decida como você deseja adicionar os botões restantes. Se seus botões usam a instância Calculadora (usando a palavra-chave this que discutimos em“Sombreamento”) como ouvinte, você deverá ver o rótulo de qualquer botão em que clicar impresso no terminal.

2. Este exercício não requer muitos códigos gráficos novos. Mas você precisa trabalhar no thread de eventos da IU para alterar com segurança os obstáculos mostrados no campo. Você pode começar devagar simplesmente imprimindo uma mensagem ou usando um JOptionPane para mostrar um alerta sempre

que uma maçã tocar uma árvore ou cerca viva. Quando estiver confiante em sua medição de distância, analise como remover um objeto de uma lista. Depois de remover um obstáculo, certifique-se de repintar o campo.

Exercícios Avançados

1. A lógica de uma calculadora é relativamente simples, mas certamente não é trivial. Comece conectando os

vários botões numéricos (1, 2, 3 e assim por diante) ao display. Você precisará acrescentar dígitos para criar números completos. Clicar no botão 1 seguido do botão 2 deve exibir um 12. Quando o usuário clica em um botão de operação como “-”, armazene qualquer número exibido no momento, bem como a operação a ser usada posteriormente. Deixe o usuário digitar um segundo número. Clicar em “=” deve armazenar este segundo número e então realizar o cálculo real. Coloque o resultado no display e deixe o usuário recomeçar.

Existem muitas (muitas!) sutilezas em um aplicativo de calculadora

profissional completo. Não se preocupe se suas primeiras tentativas

impuserem restrições, como trabalhar apenas com números de um único

dígito. O objetivo deste exercício é praticar a resposta aos eventos. Até mesmo conseguir que um dígito seja exibido no campo de exibição depois que o usuário clica em um botão já vale a pena comemorar!

Capítulo 13: Programação de Rede em Java

1. Quais protocolos de rede a classe URL suporta por padrão?

A classe URL do Java inclui suporte para os protocolos HTTP, HTTPS e FTP.

Esses três protocolos cobrem uma grande parte dos recursos disponíveis online, mas você pode criar seu próprio manipulador de protocolo se lidar com sistemas que não sejam servidores da Web ou de arquivos.

2. Você pode usar Java para baixar dados binários de uma fonte online?

Sim. Os fluxos de bytes estão no centro de todos os dados de rede em Java. Você pode ler os bytes brutos ou encadear algum outro fluxo de nível superior. Por exemplo, InputStreamReader e BufferedReader funcionam muito bem para

texto. DataInputStream pode lidar com dados binários.

3. Como você envia dados de formulário para um servidor web usando Java? (Não há necessidade de um aplicativo totalmente funcional, só queremos que você pense nas etapas de alto nível executadas e nas classes Java envolvidas.) Você pode usar a classe URL para abrir uma conexão com um servidor web.

Antes de fazer qualquer solicitação, você pode configurar a conexão para comunicação bidirecional. O comando HTTP POST permite enviar dados ao servidor no corpo da sua solicitação.

4. Que classe você usa para escutar conexões de rede de entrada?

Você usa a classe java.net.ServerSocket para criar um ouvinte de rede.

5. Ao criar seu próprio servidor como fizemos para nosso jogo, há alguma regra para escolher um número de porta?

Sim. Os números de porta devem estar entre 0 e 65.535, com portas abaixo de 1.024 normalmente reservadas para serviços conhecidos que geralmente

exigem permissões especiais para uso.

6. Um aplicativo de servidor escrito em Java pode suportar vários clientes simultâneos?

Sim. Embora seja possível criar apenas um ServerSocket em uma determinada porta, você pode aceitar centenas ou até milhares de clientes e processar suas solicitações em um thread.

7. Com quantos servidores simultâneos uma determinada instância de cliente Socket pode se comunicar?

Um. Um soquete cliente se comunica com um host em uma porta. Um aplicativo cliente pode permitir vários soquetes distintos para comunicação com vários servidores ao mesmo tempo, mas cada soquete ainda se comunicará com um único servidor.

Exercícios de código

1. Adicionar um recurso ao protocolo do nosso jogo requer a atualização do código do servidor e do cliente. Felizmente podemos reutilizar grande parte da lógica da entrada TREE em ambos os lados. Ainda mais felizmente, todo o nosso código de comunicação de rede está na classe Multiplayer.

O cliente e o servidor são classes internas, denominadas criativamente Cliente e Servidor, respectivamente. Para o servidor, adicione um loop no método run() para enviar dados de hedge logo após enviar os dados da árvore. Para o cliente, adicione um segmento HEDGE ao método run() que aceita a localização da cobertura e a adiciona ao campo.

Depois que os campos são configurados para os dois jogadores, a parte do protocolo no jogo apenas relata pontuações e desconexões. Não precisamos

modificar nada desse código. Cada jogador terá os mesmos obstáculos e a mesma oportunidade de removê-los com uma maçã lançada.

2. Um servidor de data/hora legível deve ser bastante simples, mas queremos que você pratique a configuração de seus próprios soquetes do zero. Você precisará decidir sobre um número de porta para o servidor. 3283 soletra

“DATA” no teclado do telefone se você precisar de um pouco de inspiração.

Recomendamos processar a solicitação do cliente imediatamente após aceitar a conexão. (Os exercícios avançados oferecem a oportunidade de experimentar uma abordagem mais sofisticada de uso de threads.)

Para o cliente, o único dado real configurável é o nome do servidor. Você está livre para codificar “localhost” se planeja testar sua solução usando duas janelas de terminal em sua máquina local. Nossa solução usa um argumento de linha de comando opcional, padronizando também “localhost” se você não fornecer um argumento.

Exercícios Avançados

1. Para lidar com clientes usando threads, você precisa isolar o código responsável pela comunicação com o cliente. Uma classe auxiliar (interna, anônima ou separada são boas opções) ou um lambda funcionarão. Você ainda precisa deixar o ServerSocket fazer seu trabalho e aceitar() uma nova conexão, mas você pode

entregar o objeto Socket aceito ao seu ajudante assim que o obtiver.

Será difícil realmente testar esta classe, pois você precisaria de muitos clientes solicitando a data atual ao mesmo tempo. Porém, no mínimo, sua classe FDClient atual deverá funcionar sem alterações e você ainda deverá receber a data correta.

2. Trabalhar com APIs online pode ser divertido, mas também exige atenção aos detalhes. Normalmente, você precisa responder a algumas perguntas ao

começar a criar seu cliente:

–

Qual é o URL base da API?

–

A API usa codificação padrão de formulário da web ou JSON? Caso

contrário, existe uma biblioteca com suporte para codificação e

decodificação?

–

Existem limites para quantas solicitações você pode fazer ou quantos

dados você pode baixar?

–

O site possui boa documentação com exemplos comuns de envio e

recuperação de dados?

Ao praticar, você desenvolverá seu próprio senso de quais informações são necessárias para começar a usar novas APIs. Mas você precisa praticar. Depois de

construir seu primeiro cliente, procure outro serviço. Escreva um cliente para essa API e veja se você já consegue identificar problemas comuns, ou melhor, código reutilizável de seu primeiro cliente.

1Você pode visualizar qualquer arquivo JAR ou ZIP com jar tvf <jarfile>.

2Você poderia, no entanto, usar um SortedSet ou um TreeMap, que mantêm suas entradas classificadas. Para TreeMap, as chaves são mantidas em ordem.

Glossário

abstrato

A palavra-chave abstract é usada para declarar métodos e classes abstratos. Um método abstrato não possui implementação definida; ele é declarado com argumentos e um tipo de retorno como de costume, mas o corpo entre chaves é substituído por um ponto e vírgula. A implementação de um método abstrato é fornecida por uma subclasse da classe na qual ele está definido. Se um método abstrato aparecer em uma classe, a classe também será abstrata. A tentativa de instanciar uma classe abstrata falhará em tempo de compilação.

anotações

Metadados adicionados ao código-fonte Java usando a sintaxe da tag @. As anotações podem ser usadas pelo compilador ou em tempo de execução para aumentar classes, fornecer dados ou mapeamentos ou sinalizar serviços adicionais.

Formiga

Uma ferramenta de construção mais antiga baseada em XML para aplicativos Java. As compilações Ant podem compilar, empacotar e implantar código-fonte Java, bem como gerar documentação e executar outras atividades por meio de “destinos”

conectáveis.

Interface de programação de aplicativos (API)

Uma API consiste nos métodos e variáveis que os programadores usam para trabalhar com um componente ou ferramenta em suas aplicações. As APIs da linguagem Java consistem nas classes e métodos dos pacotes java.lang, java.util, java.io, java.text e java

.net, e muitos outros.

aplicativo

Um programa Java que é executado de forma independente, em comparação com um miniaplicativo.

Ferramenta de processamento de anotações (APT)

Um frontend para o compilador Java que processa anotações por meio de uma arquitetura de fábrica

conectável, permitindo aos usuários implementar anotações personalizadas em tempo de compilação.

afirmação

Um recurso de linguagem usado para testar condições que devem ser garantidas pela lógica do programa. Se uma condição verificada por uma asserção for considerada falsa, um erro fatal será lançado. Para maior desempenho, as asserções podem ser desabilitadas quando um aplicativo é implementado.

atômico

Discreta ou transacional no sentido de que uma operação acontece como uma unidade, do tipo tudo ou nada. Certas operações na máquina virtual Java (VM) e fornecidas pela API de simultaneidade Java são atômicas.

Kit de ferramentas de janela abstrata (AWT)

Kit de ferramentas de UI, gráficos e janelas independentes de plataforma original do Java.

Boojum

O alter ego místico e espectral de um Snark. Do poema de Lewis Carroll de 1876 “A Caça ao Snark”.

boleano

Um tipo de dados Java primitivo que contém um valor verdadeiro ou falso.

limites

Nos genéricos Java, uma limitação no tipo de um parâmetro de tipo. Um limite superior especifica que um

tipo deve estender (ou pode ser atribuído a) uma classe Java específica. Um limite inferior é usado para indicar que um tipo deve ser um supertipo (ou pode ser atribuído) do tipo especificado.

boxe

Envolvimento de tipos primitivos em Java por seus tipos de wrapper de objeto. Veja também *desembalar.*

byte

Um tipo de dados Java primitivo que é um número assinado em complemento de dois de 8 bits.

ligar de volta

Um comportamento que é definido por um objeto e posteriormente invocado por outro objeto quando ocorre um evento específico. O mecanismo de eventos Java é uma espécie de retorno de chamada.

elenco

A mudança do tipo aparente de um objeto Java de um tipo para outro, tipo especificado. As conversões Java são verificadas estaticamente pelo compilador Java e em tempo de execução.

pegar

A instrução Java catch introduz um bloco de código de tratamento de exceções após uma instrução try. A palavra-chave catch é seguida por um ou mais pares de tipo de exceção e nome de argumento entre parênteses e um bloco de código entre chaves.

certificado

Um documento eletrônico que utiliza uma assinatura digital para afirmar a identidade de uma pessoa, grupo ou organização. Os certificados atestam a identidade de uma pessoa ou grupo e contêm a chave pública dessa organização. Um certificado é assinado por uma autoridade de certificação com sua assinatura digital.

autoridade de certificação (CA)

Uma organização encarregada de emitir certificados, tomando todas as medidas necessárias para verificar a identidade real para a qual está emitindo o certificado.

Caracteres

Um tipo de dados Java primitivo; uma variável do tipo char contém um único caractere Unicode de 16 bits.

aula

1. A unidade fundamental que define um objeto na maioria das linguagens de programação orientadas a objetos. Uma classe é uma coleção encapsulada de variáveis e métodos que podem ter acesso privilegiado uns aos outros.

Normalmente, uma classe pode ser instanciada para produzir um objeto que seja uma instância da classe, com seu próprio conjunto exclusivo de dados.

2. A palavra-chave class é usada para declarar uma classe, definindo assim um novo tipo de objeto.

carregador de classe

Uma instância da classe java.lang.ClassLoader, responsável por carregar classes binárias Java no Java VM. Os carregadores de classe ajudam a particionar classes com base em sua origem para fins estruturais e de segurança, e também podem ser encadeados em uma hierarquia pai-filho.

método de classe

Ver *método estático.*

caminho de classe

A sequência de locais de caminho especificando diretórios e arquivos contendo recursos e arquivos de classe Java compilados, que são pesquisados para localizar componentes de um aplicativo Java.

variável de classe

Ver *variável estática.*

cliente

O consumidor de um recurso ou a parte que inicia uma conversa no caso de uma aplicação cliente/servidor em rede. Veja também *servidor.*

API de coleções

Classes no pacote java.util principal para trabalhar e classificar coleções estruturadas ou mapas de itens. Esta API inclui as classes Vector e Hashtable, bem como itens mais recentes, como List, Map e Queue.

unidade de compilação

A unidade de código-fonte de uma classe Java. Uma unidade de compilação normalmente contém uma única definição de classe e, na maioria dos ambientes de desenvolvimento atuais, é simplesmente um arquivo com extensão .java.

compilador

Um programa que traduz o código-fonte em código executável.

arquitetura de componentes

Uma metodologia para construir partes de um aplicativo. É uma forma de construir objetos reutilizáveis que podem ser facilmente montados para formar aplicações.

composição

Combinar objetos existentes para criar outro objeto mais complexo. Ao compor um novo objeto, você cria um comportamento complexo delegando tarefas aos objetos internos. A composição é diferente da herança, que define um novo objeto alterando ou refinando o comportamento de um objeto antigo. Veja tambémém *herança.*

constructor

Um método especial que é invocado automaticamente quando uma nova instância de uma classe é criada. Construtores são usados para inicializar as variáveis do objeto recém-criado. O método construtor tem o mesmo nome da classe e nenhum valor de retorno explícito.

manipulador de conteúdo

Uma classe chamada para analisar um tipo específico de dados e convertê-lo em um objeto apropriado.

datagrama

Um pacote de dados normalmente enviado usando um protocolo sem conexão como o UDP, que não oferece garantias sobre entrega ou verificação de erros e não fornece informações de controle.

ocultação de dados

See *encapsulamento.*

cópia profunda

Uma duplicata de um objeto junto com todos os objetos aos quais ele faz referência, transitivamente. Uma cópia profunda duplica todo o “gráfico” de objetos, em vez de apenas duplicar referências. Veja também *cópia superficial.*

Modelo de objeto de documento (DOM)

Uma representação na memória de um documento XML totalmente analisado usando objetos com nomes como Elemento, Atributo e Texto. A ligação Java XML DOM API é padronizada pelo World Wide Web Consortium (W3C).

double

Um tipo de dados primitivo Java; um valor duplo é um número de ponto flutuante de 64 bits (precisão dupla) no formato binário IEEE-754 (binary64).

Definição de tipo de documento (DTD)

Um documento contendo linguagem especializada que expressa restrições na estrutura de tags XML e atributos de tags. DTDs são usados para validar um documento XML e podem restringir a ordem e o aninhamento de tags, bem como os valores permitidos de atributos.

Enterprise JavaBeans (EJBs)

Uma arquitetura de componentes de negócios do lado do servidor nomeada, mas não significativamente relacionada, à arquitetura de componentes JavaBeans. EJBs representam serviços de negócios e componentes de banco de dados e fornecem segurança declarativa e transações.

encapsulamento

A técnica de programação orientada a objetos que limita a exposição de variáveis e métodos para simplificar a API de uma classe ou pacote. Usando as palavras-chave private e protected, um programador pode limitar a exposição de partes internas (“caixa preta”) de uma classe. O encapsulamento reduz bugs e promove a reutilização e modularidade das classes. Essa técnica também é conhecida como ocultação de dados.

enumeração

A palavra-chave Java para declarar um tipo enumerado. Um enum contém uma lista de identificadores de objetos constantes que podem ser usados como uma alternativa de tipo seguro para constantes numéricas que servem como identificadores ou rótulos.

enumeração

Ver *enumeração.*

apagamento

A técnica de implementação usada pelos genéricos Java na qual as informações de tipo genérico são removidas (apagadas) e destiladas em tipos Java brutos na compilação.

Erasure fornece compatibilidade retroativa com código Java não genérico, mas introduz algumas dificuldades na linguagem.

evento

1. A ação de um usuário, como um clique do mouse ou um pressionamento de tecla.

2. O objeto Java entregue a um ouvinte de eventos registrado em resposta a uma ação do usuário ou outra atividade no sistema.

exceção

Um sinal de que ocorreu alguma condição inesperada no programa. Em Java, exceções são objetos que são subclasses de Exception ou Error (que são subclasses de Throwable). Exceções em Java são “criadas” com a palavra-chave throw e tratadas com a palavra-chave catch. Veja também *pegar*, *lançar*, e *lança*.

encadeamento de exceções

O padrão de design de capturar uma exceção e lançar uma exceção nova, de nível superior ou mais apropriada que contém a exceção subjacente como sua causa. A exceção “causa” pode ser recuperada se necessário.

estende

Uma palavra-chave usada em uma declaração de classe para especificar a superclasse da classe que está sendo definida. A classe que está sendo definida tem acesso a todas as variáveis e métodos públicos e protegidos da superclasse (ou, se a classe que está sendo definida estiver no mesmo pacote, tem acesso a todas as variáveis e métodos não privados). Se uma definição de classe omitir a cláusula extends, sua superclasse será considerada java.lang.Object.

final

Um modificador de palavra-chave que pode ser aplicado a classes, métodos e variáveis. Tem um significado semelhante, mas não idêntico em cada caso. Quando final é aplicado a uma classe, significa que a classe nunca pode ser subclassificada.

java.lang.System é um exemplo de classe final. Um método final não pode ser substituído em uma subclasse. Quando final é aplicado a uma variável, a variável é uma constante — ou seja, não pode ser modificada. (O conteúdo de um objeto mutável ainda pode ser alterado; a variável final sempre aponta para o mesmo objeto.)
finalizar

Um nome de método reservado. O método finalize() é chamado pelo Java VM quando um objeto não está mais sendo usado (ou seja, quando não há mais referências a ele), mas antes que a memória do objeto seja realmente recuperada pelo sistema. Em grande parte desfavorecido à luz das abordagens mais recentes, como a interface Closeable e try-with-resources.

finalmente

Uma palavra-chave que introduz o bloco final de uma construção

try/catch/finalmente. Os blocos catch e finalmente fornecem tratamento de exceções e limpeza de rotina para código em um bloco try. O bloco final é opcional e aparece após o bloco try e após zero ou mais blocos catch. O código em um bloco finalmente é executado uma vez, independentemente de como o código no bloco try é executado.

Na execução normal, o controle chega ao final do bloco try e segue para o bloco final, que geralmente executa qualquer limpeza necessária.

flutuador

Um tipo de dados primitivo Java; um valor flutuante é um número de ponto flutuante de 32 bits (precisão simples) representado no formato IEEE 754.

coleta de lixo

O processo de recuperação da memória de objetos que não estão mais em uso. Um objeto não está mais em uso quando não há referências a ele de outros objetos no sistema e nenhuma referência em quaisquer variáveis locais na pilha de chamadas de método de qualquer thread.

genéricos

A sintaxe e implementação de tipos parametrizados na linguagem Java, adicionada no Java 5.0. Tipos genéricos são classes Java parametrizadas pelo usuário em um ou mais tipos Java adicionais para especializar o

comportamento da classe. Os genéricos às vezes são chamados de modelos em outros idiomas.

classe genérica

Uma classe que utiliza a sintaxe genérica Java e é parametrizada por uma ou mais variáveis de tipo, que representam tipos de classe a serem substituídos pelo usuário da classe. Classes genéricas são particularmente úteis para objetos contêineres e coleções que podem ser especializadas para operar em um tipo específico de elemento.

método genérico

Um método que usa a sintaxe genérica Java e possui um ou mais argumentos ou tipos de retorno que se referem a variáveis de tipo que representam o tipo real de elemento de dados que o método usará. O compilador Java muitas vezes pode inferir os tipos das variáveis de tipo a partir do contexto de uso do método.

contexto gráfico

Uma superfície desenhável representada pela classe java.awt.Graphics. Um contexto gráfico contém informações contextuais sobre a área de desenho e fornece métodos para executar operações de desenho nela.

interface gráfica do usuário (GUI)

Uma interface de usuário visual tradicional que consiste em uma janela contendo itens gráficos como botões, campos de texto, menus suspensos, caixas de diálogo e outros componentes de interface padrão.

código hash

Um número de identificação de aparência aleatória, baseado no conteúdo dos dados de um objeto, usado como uma espécie de assinatura do objeto. Um código hash é usado para armazenar um objeto em uma tabela hash (ou mapa hash). Veja também *tabela hash*.

tabela hash

Um objeto que é como um dicionário ou uma matriz associativa. Uma tabela hash armazena e recupera elementos usando valores-chave chamados códigos hash. Veja também *código hash*.

nome de anfitrião

O nome legível dado a um computador individual conectado à Internet.

Protocolo de transferência de hipertexto (HTTP)

O protocolo usado por navegadores da web ou outros clientes para se comunicar com servidores da web. A forma mais simples do protocolo usa os comandos GET para solicitar um arquivo e POST para enviar dados.

Ambiente de Desenvolvimento Integrado (IDE)

Uma ferramenta GUI, como IntelliJ IDEA ou Eclipse, que fornece funcionalidade de edição, compilação, execução, depuração e implementação de código-fonte para o desenvolvimento de aplicativos Java.

implementos

Uma palavra-chave usada em declarações de classe para indicar que a classe implementa a interface ou interfaces nomeadas. A cláusula implements é opcional nas declarações de classe; se aparecer, deve seguir a cláusula extends (se houver). Se uma cláusula implements aparecer na declaração de uma classe não abstrata, cada método de cada interface especificada deverá ser implementado pela classe ou por uma de suas superclasses.

importar

A instrução import disponibiliza classes Java para a classe atual sob um nome abreviado ou desambigua classes importadas em massa por outras instruções import.

(As classes Java estão sempre disponíveis por seu nome totalmente qualificado, assumindo que o arquivo de classe apropriado possa ser encontrado em relação à variável de ambiente CLASSPATH e que o arquivo de classe seja legível. import não disponibiliza a classe; apenas salva a digitação e torna seu código mais legível.) Qualquer número de instruções de importação pode aparecer em um programa Java.

Eles devem aparecer, entretanto, após a instrução package opcional na parte superior do arquivo e antes da primeira classe ou definição de interface no arquivo.

herança

Um recurso importante da programação orientada a objetos que envolve a definição de um novo objeto alterando ou refinando o comportamento de um objeto existente.

Através da herança, um objeto contém implicitamente todas as variáveis e métodos não privados de sua superclasse. Java suporta herança única de classes e herança múltipla de interfaces.

classe interna

Uma definição de classe aninhada em outra classe ou método. Uma classe interna funciona dentro do escopo léxico de outra classe.

instância

Uma ocorrência de algo, geralmente um objeto. Quando uma classe é instanciada para produzir um objeto, dizemos que o objeto é uma instância da classe.

método de instância

Um método não estático de uma classe. Tal método recebe uma referência this implícita ao objeto que o invocou. Veja também *estático*, *método estático*.

instancia de

Um operador Java que retorna verdadeiro se o objeto no lado esquerdo for uma instância da classe (ou implementar a interface) especificada no lado direito.

instanceof retorna falso se o objeto não for uma instância da classe especificada ou não implementar a interface especificada. Também retorna falso se o objeto especificado for nulo.

variável de instância

Uma variável não estática de uma classe. Cada instância de uma classe possui uma cópia independente de todas as variáveis de instância da classe. Veja também *variável de classe*, *estático*.

interno

Um tipo de dados Java primitivo que é um número assinado em complemento de dois de 32 bits.

interface

1. Uma palavra-chave usada para declarar uma interface.

2. Uma coleção de métodos abstratos que definem coletivamente um tipo na linguagem Java. As classes que implementam os métodos podem declarar que implementam o tipo de interface e suas instâncias podem ser tratadas como esse tipo.

internacionalização

O processo de tornar um aplicativo acessível a pessoas que falam vários idiomas. Às vezes abreviado como I18N.

intérprete

O módulo que decodifica e executa bytecode Java. A maior parte do bytecode Java não é mais interpretado, estritamente falando, mas é compilado em código nativo dinamicamente pelo Java VM.

introspecção

O processo pelo qual um JavaBean fornece informações adicionais sobre si mesmo, complementando as informações aprendidas por reflexão.

ISO 8859-1

Uma codificação de caracteres de 8 bits padronizada pela ISO. Esta codificação também é conhecida como Latin-1 e contém caracteres do alfabeto latino adequados para o inglês e para a maioria dos idiomas da Europa Ocidental.

JavaBeans

Uma arquitetura de componentes para Java. É uma maneira de construir objetos Java interoperáveis que podem ser facilmente manipulados em um ambiente visual de construção de aplicativos.

Feijões Java

Classes Java que são construídas seguindo os padrões e convenções de design JavaBeans.

JavaScript

Uma linguagem desenvolvida no início da história da web pela Netscape para a criação de páginas web dinâmicas. Do ponto de vista do programador, não tem relação com Java, embora parte de sua sintaxe seja semelhante.

API Java para ligação XML (JAXB)

Uma API Java que permite a geração de classes Java a partir de descrições XML DTD

ou de esquema e a geração de XML a partir de classes Java.

API Java para analisadores XML (JAXP)

A API Java que permite implementações conectáveis de mecanismos XML e XSL. Esta API fornece uma maneira

neutra em termos de implementação para construir analisadores e transformações.

JAX-RPC

A API Java para chamadas de procedimento remoto XML, usadas por serviços da web.

Conectividade de banco de dados Java (JDBC)

A API Java padrão para comunicação com um banco de dados SQL (Structured Query Language).

JDOM

Um Java XML DOM nativo criado por Jason Hunter e Brett McLaughlin. JDOM é mais fácil de usar do que a API DOM padrão para Java. Ele usa a API Java Collections e convenções Java padrão. Disponível noSite do projeto JDOM.

Pacote de desenvolvedor de serviços da Web Java (JDSDP)

Um pacote de APIs de extensão padrão empacotadas como um grupo com um instalador da Sun. O JWSDP inclui JAXB, JAX-RPC e outros pacotes relacionados a XML

e serviços da web.

lambda (ou expressão lambda)

Uma maneira compacta de colocar toda a definição de uma função pequena e anônima exatamente onde você a está usando no código.

Latim-1

Um apelido para ISO 8859-1.

gerenciador de layout

Um objeto que controla a disposição dos componentes na área de exibição de um contêiner Swing ou AWT.

componente leve

Um componente Java GUI puro que não possui peer nativo no AWT.

variável local

Uma variável que é declarada dentro de um método. Uma variável local pode ser vista apenas pelo código desse método.

API de registro

A API Java para registro estruturado e relatórios de mensagens de componentes do aplicativo. A API Logging oferece suporte a níveis de registro que indicam a importância das mensagens, bem como recursos de filtragem e saída.

longo

Um tipo de dados Java primitivo que é um número assinado em complemento de dois de 64 bits.

resumo da mensagem

Um número computado criptograficamente com base no conteúdo de uma mensagem, usado para determinar se o conteúdo da mensagem foi alterado de alguma forma.

Uma alteração no conteúdo de uma mensagem alterará o resumo da mensagem.

Quando implementado corretamente, é quase impossível criar duas mensagens semelhantes com o mesmo resumo.

método

O termo de programação orientada a objetos para uma função ou procedimento.

sobrecarga de método

Fornece definições de mais de um método com o mesmo nome, mas com listas de argumentos diferentes. Quando um método sobrecarregado é chamado, o compilador determina qual deles se destina examinando os tipos de argumentos fornecidos.

substituição de método

Define um método que corresponde ao nome e aos tipos de argumento de um método definido em uma superclasse. Quando um método substituído é invocado, o interpretador usa a pesquisa dinâmica de métodos para determinar qual definição de método é aplicável ao objeto atual. A partir do Java 5.0, os métodos substituídos podem ter diferentes tipos de retorno, com restrições.

MIME (ou tipo MIME)

Um sistema de classificação de tipo de mídia frequentemente associado a anexos de email ou conteúdo de páginas da web.

Estrutura Model-View-Controller (MVC)

Um design de UI originado em Smalltalk. No MVC, os dados de um item de exibição são chamados de modelo. Uma visualização exibe uma representação específica do

modelo e um controlador fornece interação do usuário com ambos. Java incorpora muitos conceitos MVC.

modificador

Uma palavra-chave colocada antes de uma classe, variável ou método que altera a acessibilidade, o comportamento ou a semântica do item. Veja

também *abstrato*, *final*, *método nativo*, *privado*, *protegido*, *público*, *estático*, *sincronizado*.

NaN (não é um número)

Este é um valor especial dos tipos de dados double e float que representa um resultado indefinido de uma operação matemática, como zero dividido por zero.

método nativo

Um método que é implementado em uma linguagem nativa em uma plataforma host, em vez de ser implementado em Java. Os métodos nativos fornecem acesso a recursos como a rede, o sistema de janelas e o sistema de arquivos host.

novo

Um operador unário que cria um novo objeto ou matriz (ou gera uma

OutOfMemoryException se não houver memória suficiente disponível).

NIO

O “novo” pacote de E/S Java. Um pacote principal introduzido no Java 1.4 para suportar operações de E/S assíncronas, interrompíveis e escaláveis. A API NIO

suporta manipulação de E/S no estilo “select” sem thread.

nulo

nuloé um valor especial que indica que uma variável de tipo de referência não se refere a nenhuma instância de objeto. Variáveis estáticas e de instância de classes têm como padrão o valor null se não forem atribuídas de outra forma.

objeto

1. A unidade estrutural fundamental de uma linguagem de programação

orientada a objetos, encapsulando um conjunto de dados e comportamento que opera nesses dados.

2. Uma instância de uma classe, tendo a estrutura da classe, mas sua própria cópia dos elementos de dados. Veja tambémém *instância*.

pacote

A instrução package especifica o pacote Java para uma classe Java. O código Java que faz parte de um pacote específico tem acesso a todas as classes (públicas e não

públicas) do pacote e a todos os métodos e campos não privados em todas essas classes. Quando o código Java faz parte de um pacote nomeado, o arquivo de classe compilado deve ser colocado na posição apropriada na hierarquia de diretórios CLASSPATH antes de poder ser acessado pelo interpretador Java ou por outros utilitários. Se a instrução package for omitida de um arquivo, o código desse arquivo fará parte de um pacote padrão sem nome. Isso é conveniente para pequenos

programas de teste executados na linha de comando ou durante o desenvolvimento porque significa que o código pode ser interpretado no diretório atual.

tipo parametrizado

Uma classe, usando sintaxe genérica Java, que depende de um ou mais tipos a serem especificados pelo usuário. Os tipos de parâmetros fornecidos pelo usuário preenchem os valores de tipo na classe e os adaptam para uso com os tipos especificados.

polimorfismo

Um dos princípios fundamentais de uma linguagem orientada a objetos. O

polimorfismo afirma que um tipo que estende outro tipo é um “tipo” do tipo pai e pode ser usado de forma intercambiável com o tipo original, aumentando ou refinando suas capacidades.

API de preferências

A API Java para armazenar pequenas quantidades de informações por usuário ou em todo o sistema em

execuções do Java VM. A API de Preferências é análoga a um pequeno banco de dados ou ao registro do Windows.

tipo primitivo

Um dos tipos de dados Java: boolean, char, byte, short, int, long, float e double. Os tipos primitivos são manipulados, atribuídos e passados para métodos “por valor” (ou seja, os bytes reais dos dados são copiados). Veja também *tipo de referência*.

imprimir

Um estilo de formatação de texto originado na linguagem C, que conta com uma sintaxe de identificador incorporada e listas de argumentos de comprimento variável para fornecer parâmetros.

privado

A palavra-chave private é um modificador de visibilidade que pode ser aplicado a variáveis de método e campo de classes. Um método ou campo privado não é visível fora de sua definição de classe e não pode ser acessado por subclasses.

protegido

Uma palavra-chave que é um modificador de visibilidade; pode ser aplicado a variáveis de método e campo de classes. Um campo protegido é visível apenas dentro de sua classe, dentro de subclasses e dentro do pacote do qual sua classe faz parte.

Observe que subclasses em pacotes diferentes podem acessar apenas campos protegidos dentro delas mesmas ou dentro de outros objetos que sejam subclasses; eles

não podem acessar campos protegidos nas instâncias da superclasse.

manipulador de protocolo

Um componente de URL que implementa a conexão de rede necessária para acessar um recurso para um tipo de esquema de URL (como HTTP ou FTP). Um manipulador de protocolo Java consiste em duas classes: StreamHandler e URLConnection.

público

Uma palavra-chave que é um modificador de visibilidade; pode ser aplicado a classes e interfaces e às variáveis de método e campo de classes e interfaces. Uma classe ou interface pública é visível em qualquer lugar. Uma classe ou interface não pública é visível apenas dentro de seu pacote. Um método ou variável pública é visível em todos os lugares em que sua classe está visível. Quando nenhum dos modificadores private, protected ou public é especificado, um campo fica visível apenas dentro do pacote do qual sua classe faz parte.

criptografia de chave pública

Um sistema criptográfico que requer chaves públicas e privadas. A chave privada pode descriptografar mensagens criptografadas com a chave pública correspondente e vice-versa. A chave pública pode ser disponibilizada ao público sem comprometer a segurança e utilizada para verificar se as mensagens enviadas pelo titular da chave privada devem ser genuínas.

fila

Uma estrutura de dados semelhante a uma lista normalmente usada no estilo primeiro a entrar, primeiro a sair para armazenar itens de trabalho em buffer.

tipo bruto

Em genéricos Java, o tipo Java simples de uma classe sem nenhuma informação de parâmetro de tipo genérico. Este é o verdadeiro tipo de todas as classes Java depois de compiladas. Veja também *apagamento*.

tipo de referência

Qualquer objeto ou array. Os tipos de referência são manipulados, atribuídos e passados para métodos “por referência”. Em outras palavras, o valor subjacente não é copiado; apenas uma referência a ele é. Veja também *tipo primitivo*.

reflexão

A capacidade de uma linguagem de programação interagir com estruturas da própria linguagem em tempo de execução. O Reflection em Java permite que um programa Java examine arquivos de classe em tempo de execução para descobrir seus métodos e variáveis, e para invocar métodos ou modificar variáveis dinamicamente.

expressão regular

Uma sintaxe compacta, porém poderosa, para descrever um padrão em texto.

Expressões regulares podem ser usadas para reconhecer e analisar a maioria dos tipos de construções textuais, permitindo uma ampla variação em sua forma.

API de expressão regular

O pacote java.util.regex principal para usar expressões regulares. O pacote regex pode ser usado para pesquisar e substituir texto com base em padrões sofisticados.

Esquema

Os esquemas XML substituem os DTDs. Introduzida pelo W3C, XML Schema é uma linguagem baseada em XML para expressar restrições na estrutura de tags XML e

atributos de tags, bem como na estrutura e tipo do conteúdo dos dados. Outros tipos de linguagens de esquema XML possuem sintaxes diferentes.

Kit de desenvolvimento de software (SDK)

Um pacote de software distribuído pela Oracle para desenvolvedores Java. Inclui o interpretador Java, classes Java e ferramentas de desenvolvimento Java: compilador, depurador, desmontador, visualizador de miniaplicativos, gerador de arquivo stub e gerador de documentação. Também chamado de JDK.

Gerente de segurança

A classe Java que define os métodos que o sistema chama para verificar se uma determinada operação é permitida no ambiente atual.

serializar

Serializar significa ordenar ou tornar sequencial. Métodos serializados são métodos que foram sincronizados em relação aos threads para que apenas um possa estar em execução em um determinado momento.

servidor

A parte que fornece um recurso ou aceita uma solicitação de conversa no caso de uma aplicação cliente/servidor em rede. Veja também *cliente*.

servlet

Um componente de aplicativo Java que implementa a API javax.servlet.Servlet, permitindo que ela seja executada dentro de um contêiner de servlet ou servidor web.

Servlets são amplamente utilizados em aplicações web para processar dados do usuário e gerar HTML ou outras formas de saída.

contexto de servlet

Na API Servlet, este é o ambiente de aplicação web de um servlet que fornece recursos de servidor e aplicação. O caminho da URL base do aplicativo Web também é frequentemente chamado de contexto de servlet.

sombra

Para declarar uma variável com o mesmo nome de uma variável definida em uma superclasse. Dizemos que a variável “sombra” a variável da superclasse. Use a palavra-chave super para se referir à variável sombreada ou consulte-a convertendo o objeto para o tipo da superclasse.

cópia superficial

Uma cópia de um objeto que duplica apenas os valores contidos no próprio objeto. As referências a outros

objetos são repetidas como referências e não são duplicadas. Veja também *cópia profunda.*

curto

Um tipo de dados Java primitivo que é um número assinado em complemento de dois de 16 bits.

assinatura

1. Referindo-se a uma assinatura digital. Uma combinação do resumo da mensagem, criptografado com a chave privada do signatário, e o certificado do signatário, atestando a identidade do signatário. Alguém que recebe uma mensagem assinada pode obter a chave pública do signatário do certificado, descriptografar o resumo da mensagem criptografada e comparar esse

resultado com o resumo da mensagem calculado a partir da mensagem

assinada. Se os dois resumos da mensagem concordarem, o destinatário saberá que a mensagem não foi modificada e que o signatário é quem afirma ser.

2. Referindo-se a um método Java. O nome do método e os tipos de argumento e possivelmente o tipo de retorno, identificando coletivamente o método de maneira exclusiva em algum contexto.

miniaplicativo assinado

Applet empacotado em arquivo JAR assinado com assinatura digital, permitindo autenticação de sua origem e validação da integridade de seu conteúdo.

classe assinada

Uma classe Java (ou arquivo Java) que possui uma assinatura anexada. A assinatura permite ao destinatário verificar a origem da classe e se ela não foi modificada. O

destinatário pode, portanto, conceder à classe maiores privilégios de tempo de execução.

tomadas

Uma API de rede originada no BSD Unix. Um par de soquetes fornece os pontos finais para comunicação entre duas partes na rede. Um soquete de servidor escuta conexões de clientes e cria soquetes individuais no lado do servidor para cada conversa.

girador

Um componente GUI que exibe um valor e um par de pequenos botões para cima e para baixo que aumentam ou diminuem o valor. O Swing JSpinner pode trabalhar com intervalos de números e datas, bem como enumerações arbitrárias.

estático

Uma palavra-chave que é um modificador aplicado a declarações de métodos e variáveis dentro de uma classe. Uma variável estática também é conhecida como variável de classe, em oposição a variáveis de instância não estáticas. Embora cada instância de uma classe tenha um conjunto completo de suas próprias variáveis de instância, há apenas uma cópia de cada variável de classe estática,

independentemente do número de instâncias da classe (talvez zero) que são criadas.

variáveis estáticas podem ser acessadas pelo nome da classe ou por meio de uma instância. Variáveis não estáticas podem ser acessadas somente por meio de uma instância.

importação estática

Uma instrução, semelhante à importação de classe e pacote, que importa os nomes de métodos estáticos e variáveis de uma classe para um escopo de classe. A importação estática é uma conveniência que fornece o efeito de métodos e constantes globais.

método estático

Um método declarado estático. Métodos deste tipo não são passados implícitamente nestas referências e podem referir-se apenas a variáveis de classe e invocar outros métodos de classe da classe atual. Um método de classe pode ser invocado através do nome da classe, em vez de através de uma instância da classe.

variável estática

Uma variável declarada estática. Variáveis desse tipo estão associadas à classe, e não a uma instância específica da classe. Existe apenas uma cópia de uma variável estática, independentemente do número de instâncias da classe que são criadas.

fluxo

Um fluxo de dados ou um canal de comunicação. Todas as E/S fundamentais em Java são baseadas em fluxos. O pacote NIO usa canais orientados a pacotes. Também é uma estrutura para programação funcional introduzida no Java 8.

corda

Uma sequência de dados de caracteres e a classe Java usada para representar esse tipo de dados de caracteres. A classe String inclui muitos métodos para operar em objetos string.

subclasse

Uma classe que estende outra. A subclasse herda os métodos e variáveis públicos e protegidos de sua superclasse. Veja também *estende*.

super

Uma palavra-chave usada por uma classe para se referir a variáveis e métodos de sua classe pai. A referência especial super é usada da mesma forma que a referência especial this é usada para qualificar referências ao contexto do objeto atual.

superclasse

Uma classe pai, estendida por alguma outra classe. Os métodos e variáveis públicos e protegidos da superclasse estão disponíveis para a subclasse. Veja também *estende*.

sincronizado

Uma palavra-chave usada de duas maneiras relacionadas em Java: como modificador e como instrução. Primeiro, é um modificador aplicado a métodos de classe ou instância. Indica que o método modifica o estado interno da classe ou o estado interno de uma instância da classe de uma forma que não é threadsafe. Antes de executar um método de classe sincronizado, Java obtém um

bloqueio na classe para garantir que nenhum outro thread possa modificar a classe simultaneamente. Antes de executar um método de instância sincronizada, Java obtém um bloqueio na instância que invocou o método, garantindo que nenhum outro thread possa modificar o objeto ao

mesmo tempo. A sincronização também garante que as alterações em um valor sejam propagadas entre threads e eventualmente visíveis em todos os núcleos do processador.

Java também suporta uma instrução sincronizada que serve para especificar uma

“seção crítica” do código. A palavra-chave sincronizada é seguida por uma expressão entre parênteses e uma instrução ou bloco de instruções. A expressão deve ser avaliada como um objeto ou matriz. Java obtém um bloqueio no objeto ou array especificado antes de executar as instruções.

TCP (protocolo de controle de transmissão)

Um protocolo confiável e orientado à conexão. Um dos protocolos em que a Internet se baseia.

esse

Dentro de um método de instância ou construtor de uma classe, this refere-se a “este objeto” – a instância que está sendo operada no momento. É útil referir-se a uma variável de instância da classe que foi ocultada por uma variável local ou argumento de método. Também é útil passar o objeto atual como argumento para métodos estáticos ou métodos de outras classes. Há um uso adicional para isso: quando aparece como a primeira

instrução em um método construtor, refere-se a um dos outros construtores da classe.

fio

Um fluxo independente de execução dentro de um programa. Como Java é uma linguagem de programação multithread, mais de um thread pode estar em execução no interpretador Java ao mesmo tempo. Threads em Java são representados e controlados por meio do objeto Thread.

Grupo de discussão

Um grupo de threads “recicláveis” usados para atender solicitações de trabalho. Um thread é alocado para manipular um item e depois retornado ao pool.

lançar

A instrução throw sinaliza que uma condição excepcional ocorreu lançando um objeto Throwable (exceção) especificado. Esta instrução interrompe a execução do programa e a passa para a instrução catch mais próxima que pode manipular o objeto de exceção especificado.

lança

A palavra-chave throws é usada em uma declaração de método para listar as exceções que o método pode lançar. Quaisquer exceções que um método possa levantar que não sejam subclasses de Error ou RuntimeException devem ser capturadas dentro do método ou declaradas na cláusula throws do método.

tentar

A palavra-chave try indica um bloco de código protegido ao qual as cláusulas catch e finalmente se aplicam. A instrução try em si não executa nenhuma ação especial. Veja

também *pegar e finalmente* para obter mais informações sobre a construção try/catch/finalmente.

tente com recursos

Um bloco try que também abre recursos que implementam a interface Closeable para limpeza automática.

tipo instanciação

Em genéricos Java, o ponto em que um tipo genérico é aplicado fornecendo tipos reais ou curinga como parâmetros de tipo. Um tipo genérico é instanciado pelo usuário do tipo, criando efetivamente um novo tipo na linguagem Java especializada para os tipos de parâmetro.

invocação de tipo

Ver *tipo instanciação*. O termo invocação de tipo às vezes é usado por analogia com a sintaxe de invocação de método.

Protocolo de datagrama de usuário (UDP)

Um protocolo sem conexão e não confiável. UDP descreve uma conexão de dados de rede baseada em datagramas com pouco controle de pacotes.

desembalar

Desempacotar um valor primitivo que é mantido em seu tipo de wrapper de objeto e recuperar o valor como um primitivo.

Unicode

Um padrão universal para codificação de caracteres de texto, acomodando formas escritas de quase todos os idiomas. Unicode é padronizado pelo Unicode Consortium.

Java usa Unicode para seus tipos char e String.

UTF-8 (formato de transformação UCS de 8 bits)

Uma codificação para caracteres Unicode (e, mais geralmente, caracteres UCS) comumente usada para transmissão e armazenamento. É um formato multibyte no qual caracteres diferentes requerem diferentes números de bytes para serem representados.

lista de argumentos de comprimento variável

Um método em Java pode indicar que pode aceitar qualquer número de um tipo especificado de argumento após sua lista inicial fixa de argumentos. Os argumentos são tratados empacotando-os como um array.

varargs

Ver *lista de argumentos de comprimento variável.*

vetor

Uma matriz dinâmica de elementos.

verificador

Uma espécie de provador de teoremas que percorre o bytecode Java antes de ser executado e garante que ele se comporte bem e não viole o modelo de segurança Java.

O verificador de bytecode é a primeira linha de defesa no modelo de segurança Java.

Arquivo de recursos de aplicativos da Web (arquivo WAR)

Um arquivo JAR com estrutura adicional para armazenar classes e recursos para aplicações web. Um arquivo WAR inclui um diretório WEB-INF para classes, bibliotecas e o arquivo de implementação web.xml.

aplicação web

Um aplicativo executado em um servidor web ou servidor de aplicativos, normalmente usando um navegador web como cliente.

serviço de internet

Um serviço em nível de aplicativo executado em um servidor e acessado de maneira padrão usando XML para empacotamento de dados e HTTP como transporte de rede.

tipo curinga

Em genéricos Java, uma sintaxe “*” usada no lugar de um tipo de parâmetro real para instanciação de tipo para indicar que o tipo genérico representa um conjunto ou supertipo de muitas instanciações de tipo concreto.

XIncluir

Um padrão XML e API Java para inclusão de documentos XML.

Linguagem de marcação extensível (XML)

Uma linguagem de marcação universal para texto e dados, usando tags aninhadas para adicionar estrutura e metainformações ao conteúdo.

XPath

Um padrão XML e API Java para combinar elementos e atributos em XML usando uma linguagem de expressão hierárquica semelhante a regex.

Linguagem de folha de estilo extensível/XSLTransformações (XSL/XSLT) Uma linguagem baseada em XML para descrever estilos e transformação de documentos XML. O estilo envolve a simples adição de marcação, geralmente para apresentação. XSLT permite reestruturação completa de documentos, além de estilização.

Índice

Símbolos

•

$ (cifrão), marcador de posição,Marcadores de posição

•

Operador % (por cento),Operadores

•

. (ponto) (ver notação/operador de ponto (.))

•

Notação .* (ponto estrela),Pacotes e Importações

•

/** (comentários do documento),Comentários Javadoc

•

// ou /* (sintaxe de comentário),Comentários

•

<> (colchetes angulares), parâmetro de tipo,Insira genéricos

•

<String> objeto,Insira genéricos, JList

•

= (operador de atribuição),Atribuição

•

== (operador igual),Comparando Strings

•

> (operador de comparação),fazer/enquanto loops

•

>>, operador de deslocamento aritmético para a direita,Operadores

\ (barra invertida), precedendo sequências de escape,Personagens

escapados, Localização de caminho

•

\b ou \B (marcador de posição de limite de palavra),Marcadores de posição

•

\d ou \D (caractere numérico ou não numérico),Personagens e classes de

personagens

•

\s ou \S (caractere de espaço em branco ou não-espaço em

branco),Personagens e classes de personagens

•

\s ou \s* (espaço em branco único ou múltiplo),Texto de tokenização

•

Caractere \w ou \W (palavra ou não),Personagens e classes de personagens

•

^ (quilate)

–

invertendo classe de personagem,Classes de personagens

personalizadas

–

marcador de posição,Marcadores de posição

•

{x,} (intervalo), pelo menos x ou mais iterações,Iteração (multiplicidade)

•

{} (chaves), blocos de código,Declarações

•

| (Barra vertical)

–

alternância,Alternância

–

ou sintaxe,Manipulação de exceção

•

' ' (aspas simples), delimitando literais de caracteres,Literais de caracteres

•

–

JFrame,Molduras e Janelas

–

JPanel,JPanel

–

Lista,Lista, Contêineres: Construindo uma Ratoeira Melhor

–

Fila,Fila

•

Método addAll(), Coleção,A interface da coleção

•

método addMouseListener(),Eventos de mouse

•

método addMouseMotionListener(),Construtores

•

compilação antecipada (AOT),Uma máquina virtual

•

algoritmo, montagem,Declarações, expressões e algoritmos

•

–

col[eções,Aplicação: Árvores no Campo-Aplicação: Árvores no Campo]

–

Olá[Java,OláJava-O método paintComponent()]

–

Olá[Java2,HelloJava2: a sequência-Interfaces]

–

Ferramentas Java e ambiente p[ara,Ferramentas e ambiente Java-

Pegando os exemplos]

–

corrend[o,Executando o Projeto-Executando o Projeto, Executando

aplicativos Java-Executando aplicativos Java]

•

método apply(), IntFunction,[Interfaces Funcionais]

•

APT (Ferramenta de Processamento de Anotaç[ões),Glossário]

•

B

C

D

E

F

–

ByteBuffer, Buffer operations

–

List, List

–

Map, The Map Interface

–

Stream, Stream generators

•

getAbsolutePath() method, File, File operations

•

getCanonicalPath() method, File, File operations

•

getCause() method, Exception, Chaining and re-throwing exceptions

•

getContent() method

–

try/catch, Bubbling Up

–

H

I

J

•

java.util.function package, Functional Approaches in Java-Advanced Exercises

–

BinaryOperator, Creating a custom reducer

–

DoubleUnaryOperator, Using Lambdas Directly

–

IntFunction interface, Functional Interfaces

–

Predicate interface, Filtering Streams

•

java.util.regext API, The java.util.regex API-Splitting strings

•

java.util.stream package, Streams-Collectors

–

Collector interface, Collectors

–

filtering, Filtering Streams

–

flatmaps, Flatmaps

K

•

M

N

–

O

P

•

•

–

JPanel, JPanel

–

List, List

–

Map, The Map Interface

–

Queue, Queue

•

removeAll() method, Collection, The Collection Interface

•

renameTo() method, File, File operations

•

repaint() method

–

JComponent, The repaint() Method

–

JPanel, JPanel

•

REPL (Read Eval Print Loop), Trying Java-Trying Java

S

T

U

V

W

•

X

Z